INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Sciences Cognitives

préparée au Laboratoire Leibniz-IMAG
dans le cadre de l'Ecole Doctorale Ingénierie pour le Vivant : Santé, Cognition, Environnement

présentée et soutenue publiquement

par

M. Rémi Coulom

le 19 juin 2002

Titre :

Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur

Directeur de Thèse : M. Philippe Jorrand

JURY

M. Jean Della Dora	Président
M. Kenji Doya	Rapporteur
M. Manuel Samuelides	Rapporteur
M. Stéphane Canu	Rapporteur
M. Philippe Jorrand	Directeur de thèse
Mme. Mirta B. Gordon	Examinateur

Remerciements

Je remercie Monsieur Philippe Jorrand pour avoir été mon directeur de thèse. Je remercie les membres du jury, Mme Mirta Gordon, Messieurs Kenji Doya, Manuel Samuelides, Stéphane Canu et Jean Della Dora pour avoir accepté d'évaluer mon travail, et pour leurs remarques pertinentes qui ont permis d'améliorer ce texte. Je remercie les chercheurs du laboratoire Leibniz pour leur accueil, en particulier son directeur, Monsieur Nicolas Balacheff, et les membres des équipes "Apprentissage et Cognition" et "Réseaux de Neurones", Messieurs Gilles Bisson, Daniel Memmi et Bernard Amy, ainsi que tous les étudiants avec lesquels j'ai travaillé. Je remercie enfin le responsable de la Formation Doctorale en Sciences Cognitives, Monsieur Pierre Escudier, pour ses conseils.

Table des matières

Résumé (Summary in French)

Ce résumé est composé d'une traduction de l'introduction et de la conclusion de la thèse, ainsi que d'une synthèse des résultats présentés dans le développement. La traduction est assez grossière, et les lecteurs anglophones sont vivement encouragés à lire la version originale.

Introduction

Construire des contrôleurs automatiques pour des robots ou des mécanismes de toutes sortes a toujours représenté un grand défi pour les scientifiques et les ingénieurs. Les performances des animaux dans les tâches motrices les plus simples, telles que la marche ou la natation, s'avèrent extrêment difficiles à reproduire dans des systèmes artificiels, qu'ils soient simulés ou réels. Cette thèse explore comment des techniques inspirées par la Nature, les réseaux de neurones artificiels et l'apprentissage par renforcement, peuvent aider à résoudre de tels problèmes.

Contexte

Trouver des actions optimales pour contrôler le comportement d'un système dynamique est crucial dans de nombreuses applications, telles que la robotique, les procédés industriels, ou le pilotage de véhicules spatiaux. Des efforts de recherche de grande ampleur ont été produits pour traiter les questions théoriques soulevées par ces problèmes, et pour fournir des méthodes pratiques permettant de construire des contrôleurs efficaces.

L'approche classique de la commande optimale numérique consiste à calculer une trajectoire optimale en premier. Ensuite, un contrôleur peut être construit pour suivre cette trajectoire. Ce type de méthode est souvent utilisé dans l'astronautique, ou pour l'animation de personnages artificiels dans des films. Les algorithmes modernes peuvent résoudre des problèmes très complexes, tels que la démarche simulée optimale de Hardt *et al* [30].

Bien que ces méthodes peuvent traiter avec précision des systèmes très complexes, elles ont des limitations. En particulier, calculer une trajectoire optimale est souvent trop coûteux pour être fait en ligne. Ce n'est pas un problème pour les sondes spatiales ou l'animation, car connaître une seule trajectoire optimale en avance suffit. Dans d'autres situations, cependant, la dynamique du système peut ne pas être complètement prévisible et il peut être nécessaire de trouver de nouvelles actions optimales rapidement. Par exemple, si un robot marcheur trébuche sur un obstacle imprévu, il doit réagir rapidement pour retrouver son équilibre.

Pour traiter ce problème, d'autres méthodes ont été mises au point. Elles permettent de construire des contrôleurs qui produisent des actions optimales quelle que soit la situation, pas seulement dans le voisinage d'une trajectoire pré-calculée. Bien sûr, c'est une tâche beaucoup plus difficile que trouver une seule trajectoire optimale, et donc, ces techniques ont des performances qui, en général, sont inférieures à celles des méthodes classiques de la commande optimale lorsqu'elles sont appliquées à des problèmes où les deux peuvent être utilisées.

Une première possibilité consiste à utiliser un réseau de neurones (ou n'importe quel type d'approximateur de fonctions) avec un algorithme d'apprentissage supervisé pour généraliser la commande à partir d'un ensemble de trajectoires. Ces trajectoires peuvent être obtenues en enregistrant les actions d'experts humains, ou en les générant avec des méthodes de commande optimale numérique. Cette dernière technique est utilisée dans l'algorithme d'évitement d'obstacles mobiles de Lachner *et al.* [35], par exemple.

Une autre solution consiste à chercher directement dans un ensemble de contrôleurs avec un algorithme d'optimization. Van de Panne [50] a combiné une recherche stochastique avec une descente de gradient pour optimiser des contrôleurs. Les algorithmes génétiques sont aussi bien adaptés pour effectuer cette optimisation, car l'espace des contrôleurs a une structure complexe. Sims [63, 62] a utilisé cette technique pour faire évoluer des créatures virtuelles très spectaculaires qui marchent, combattent ou suivent des sources de lumière. De nombreux autres travaux de recherche ont obtenus des contrôleurs grâce aux algorithmes génétiques, comme, par exemple ceux de Meyer *et al.* [38].

Enfin, une large famille de techniques pour construire de tels contrôleurs est basée sur les principes de la programmation dynamique, qui ont été introduits par Bellman dans les premiers jours de la théorie du contrôle [13]. En particulier, la théorie de l'apprentissage par renforcement (ou programmation neuro-dynamique, qui est souvent considérée comme un synonyme) a été appliquée avec succès à un grande variété de problèmes de commande. C'est cette approche qui sera développée dans cette thèse.

Apprentissage par renforcement et réseaux de neurones

L'apprentissage par renforcement, c'est apprendre à agir par essai et erreur. Dans ce paradigme, un agent peut percevoir sont état et effectuer des actions. Après chaque action, une récompense numérique est donnée. Le but de l'agent est de maximiser la récompense totale qu'il reçoit au cours du temps.

Une grande variété d'algorithmes ont été proposés, qui selectionnent les actions de façon à explorer l'environnement et à graduellement construire une stratégie qui tend à obtenir une récompense cumulée maximale [68, 33]. Ces algorithmes ont été appliqués avec succès à des problèmes complexes, tels que les jeux de plateau [70], l'ordonnancement de tâches [81], le contrôle d'ascenceurs [20] et, bien sûr, des tâches de contrôle moteur, simulées [67, 24] ou réelles [41, 5].

Model-based et model-free

Ces algorithmes d'apprentissage par renforcement peuvent être divisés en deux catégories : les algorithmes dits *model-based* (ou indirects), qui utilisent une estimation de la dynamique du système, et les algorithmes dits *model-free* (ou directs), qui n'en utilisent pas. La supériorité d'une approche sur l'autre n'est pas claire, et dépend beaucoup du problème particulier à résoudre. Les avantages principaux apportés par un modèle est que l'expérience réelle peut être complémentée par de l'expérience simulée («imaginaire»), et que connaître la valeur des états suffit pour trouver le contrôle optimal. Les inconvénients les plus importants des algorithmes model-based est qu'ils sont plus complexes (car il faut mettre en œuvre un mécanisme pour estimer le modèle), et que l'expérience simulée produite par le modèle peut ne pas être fidèle à la réalité (ce qui peut induire en erreur le processus d'apprentissage).

Bien que la supériorité d'une approche sur l'autre ne soit pas complètement évidente, certains résultats de la recherche tendent à indiquer que l'apprentissage par renforcement model-based peut résoudre des problèmes de contrôle moteur de manière plus efficace. Cela a été montré dans des simulations [5, 24] et aussi dans des expériences avec des robots réels. Morimoto et Doya [42] ont combiné l'expérience simulée avec l'expérience réelle pour apprendre à un robot à se mettre debout avec l'algorithme du Q-learning. Schaal et Atkeson ont aussi utilisé avec succès l'apprentissage par renforcement model-base dans leurs expériences de robot jongleur [59].

Réseaux de neurones

Quasiment tous les algorithmes d'apprentissage par renforcement font appel à l'estimation de «fonctions valeur» qui indiquent à quel point il est bon d'être dans un état donné (en termes de récompense totale attendue dans le long terme), ou à quel point il est bon d'effectuer une action donnée dans un état donné. La façon la plus élémentaire de construire cette fonction valeur consiste à mettre à jour une table qui contient une valeur pour chaque état (ou chaque paire état-action), mais cette approche ne peut pas fonctionner pour des problèmes à grande échelle. Pour pouvoir traiter des tâches qui ont un très grand nombre d'états, il est nécessaire de faire appel aux capacités de généralisation d'approximateurs de fonctions.

Les réseaux de neurones feedforward sont un cas particulier de tels approximateurs de fonctions, qui peuvent être utilisés en combinaison avec l'apprentissage par renforcement. Le succès le plus spectaculaire de cette technique est probablement le joueur de backgammon de Tesauro [70], qui a réussi à atteindre le niveau des maîtres humains après des mois de jeu contre lui-même. Dans le jeu de backgammon, le nombre estimé de positions possibles est de l'ordre de 10^{20}. Il est évident qu'il est impossible de stocker une table de valeurs sur un tel nombre d'états possibles.

Résumé et contributions

Le problème

L'objectif des travaux présentés dans cette thèse est de trouver des méthodes efficaces pour construire des contrôleurs pour des tâches de contrôle moteur simulées. Le fait de travailler sur des simulations implique qu'un modèle exact du système à contrôler est connu. De façon à ne pas imposer des contraintes artificielles, on supposera que les algorithmes d'apprentissage ont accès à ce modèle. Bien sûr, cette supposition est une limitation importante, mais elle laisse malgré tout de nombreux problèmes difficiles à résoudre, et les progrès effectués dans ce cadre limité peuvent être transposés dans le cas général où un modèle doit être appris.

L'approche

La technique employée pour aborder ce problème est l'algorithme TD(λ) continu de Doya [23]. Il s'agit d'une formulation continue du TD(λ) classique de Sutton [66] qui est bien adaptée aux problèmes de contrôle moteur. Son efficacité a été démontrée par l'apprentissage du balancement d'une tige en rotation montée sur un chariot mobile [24].

Dans de nombreux travaux d'apprentissage par renforcement appliqué au contrôle moteur, c'est un approximateur de fonctions linéaire qui est utilisé pour approximer la fonction valeur. Cette technique d'approximation a de nombreuses propriétés intéressantes, mais sa capacité à traiter un grand nombre de variables d'état indépendantes est assez limitée.

L'originalité principale de l'approche suivie dans cette thèse est que la fonction valeur est estimée avec des réseaux de neurones feedforward au lieu d'approximateurs de fonction linéaires. La non-linéarité de ces réseaux de neurones les rend difficiles à maîtriser, mais leurs excellentes capacités de généralisation dans des espaces d'entrée en dimension élevée leur permet de résoudre des problèmes dont la complexité est supérieure de plusieurs ordres de grandeur à ce que peut traiter un approximateur de fonctions linéaire.

Contributions

Ce travail explore les problèmes numériques qui doivent être résolus de façon à améliorer l'efficacité de l'algorithme TD(λ) continu lorsqu'il est utilisé en association avec des réseaux de neurones feedforward. Les contributions principales qu'il apporte sont :

- Une méthode pour traiter les discontinuités de la commande. Dans de nombreux problèmes, la commande est discontinue, ce qui rend difficile l'application de méthodes efficaces d'intégration numérique. Nous montrons que la commande de Filippov peut être obtenue en utilisant des informations de second ordre sur la fonction valeur.
- Une méthode pour traiter les discontinuités de l'état. Elle est nécessaire pour pouvoir appliquer l'algorithme TD(λ) continu à des problèmes avec des chocs ou des capteurs discontinus.
- L'algorithme Vario-η [47] est proposé comme une méthode efficace pour effectuer la descente de gradient dans l'apprentissage par renforcement.
- De nombreux résultats expérimentaux indiquent clairement le potentiel énorme de l'utilisation de réseaux de neurones feedforward dans les algorithmes d'apprentissage par renforcement appliqués au contrôle moteur. En particulier, un nageur articulé complexe, possédant 12 variables d'état indépendantes et 4 variables de contrôle a appris à nager grâce aux réseaux de neurones feedforward.

Plan de la thèse

- Partie I : Théorie
  - Chapitre 1 : La Programmation dynamique
  - Chapitre 2 : Les réseaux de neurones

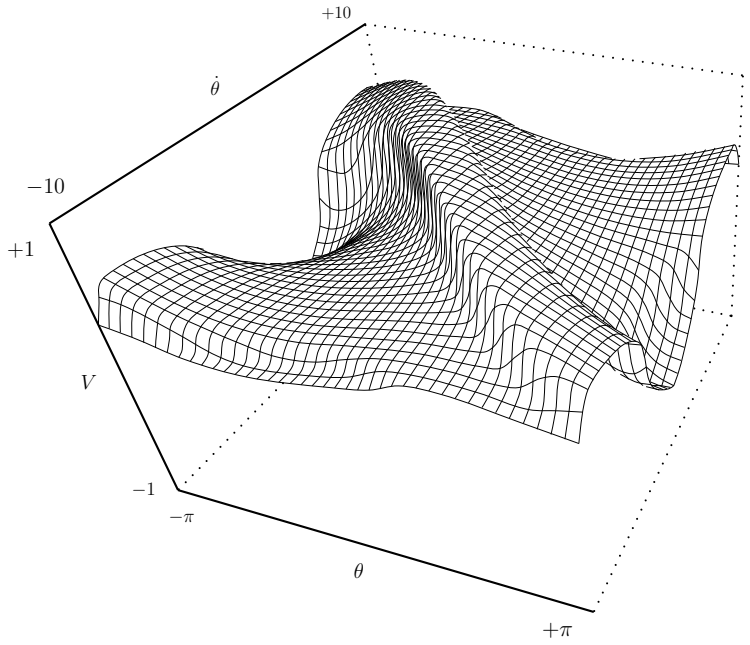


FIG. 1 – Fonction valeur pour le problème du pendule obtenue par programmation dynamique sur une grille de 1600×1600.

  - Chapitre 3 : La Programmation neuro-dynamique
  - Chapitre 4 : Le TD(λ) en pratique
- Partie II : Expériences
  - Chapitre 5 : Les Problèmes classiques
  - Chapitre 6 : Robot Auto Racing Simulator
  - Chapitre 7 : Les Nageurs

Théorie

Programmation dynamique

La programmation dynamique a été introduite par Bellman dans les années 50 [13] et est la base théorique des algorithmes d'apprentissage par renforcement. La grosse limitation de la programmation dynamique est qu'elle requiert une discrétisation de l'espace d'état, ce qui n'est pas raisonablement envisageable pour des problèmes en dimension supérieure à 6, du fait du coût exponentiel de la discrétisation avec la dimension de l'espace d'état. La figure 1 montre la fonction valeur obtenue par programmation dynamique sur le problème du pendule simple (voir Appendice B).

Réseaux de neurones

Les réseaux de neurones sont des approximateurs de fonctions dont les capacités de généralisation vont permettre de résoudre les problèmes d'explosion combinatoire.

Programmation neuro-dynamique

La programmation neuro-dynamique consiste à combiner les techniques de réseaux de neurones avec la programmation dynamique. Les algorithmes de différences temporelles, et en particulier, le TD(λ) continu inventé par Doya [24] sont particulièrement bien adaptés à la résolution de problèmes de contrôle moteur.

Le TD(λ) dans la pratique

Dans ce chapitre sont présentées des techniques pour permettre une utilisation efficace de l'algorithme TD(λ) avec des réseaux de neurones. Il s'agit des contributions théoriques les plus importantes de ce travail.

Expériences

Problèmes classiques

Dans ce chapitre sont présentées des expériences sur les problèmes classiques (pendule simple, tige-chariot et acrobot). Les figures 2 et 3 montrent les résultats obtenus avec un approximateur de fonctions linéaire et un réseau de neurones feedforward. Bien qu'il possède moins de paramètres, le réseau feedforward permet d'obtenir une approximation de la fonction valeur qui est beaucoup plus précise.

Robot Auto Racing Simulator

Le Robot Auto Racing Simulator est un simulateur de voiture de course où l'objectif est de construire un contrôleur pour conduire une voiture. La figure 4 montre les résultats obtenus sur un petit circuit.

Les Nageurs

Les nageurs sont formés de segments articulés, et plongés dans un liquide visqueux. L'objectif est de trouver une loi de commande leur permettant de nager dans une direction donnée. C'est un problème dont la dimensionalité

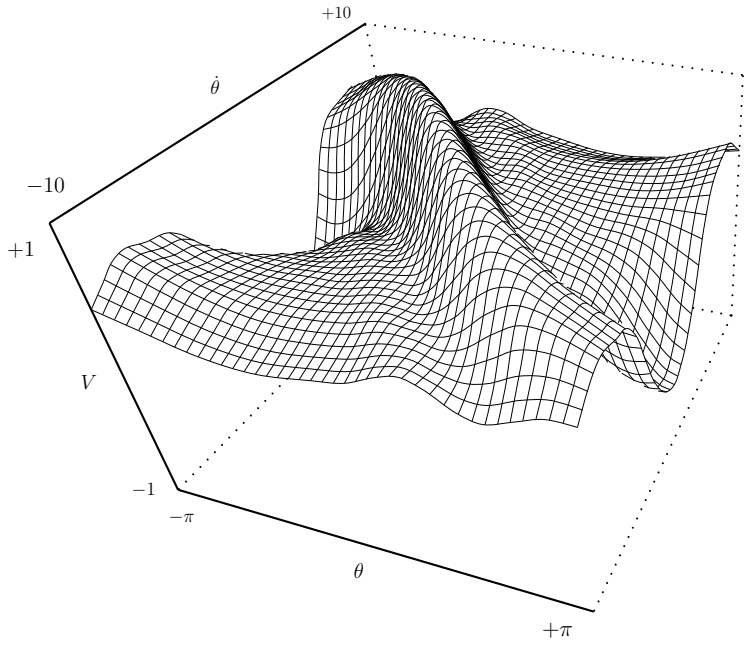


FIG. 2 – Fonction valeur obtenue avec un réseau gaussien normalisé (similaire à ceux utilisés dans les expériences de Doya [24])

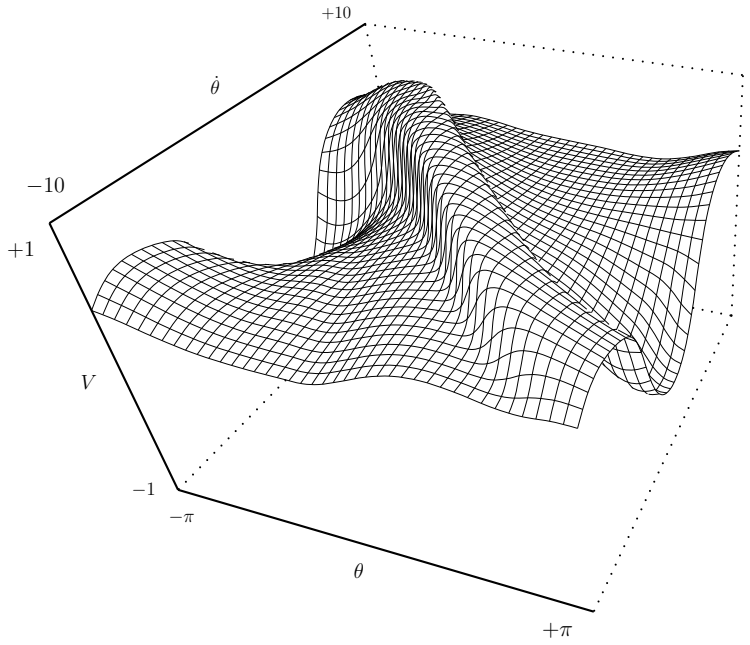


FIG. 3 – Fonction valeur apprise par un réseau à 12 neurones

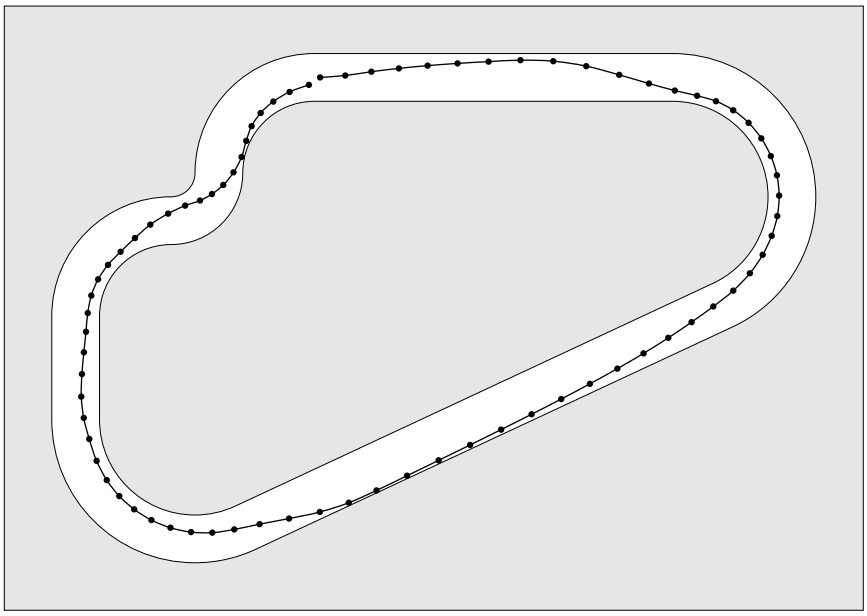


FIG. 4 – Une trajectoire obtenue par la voiture de course avec un réseaux de neurones à 30 neurones.

dépasse largement celle des problèmes classiquement traités dans la littérature sur l'apprentissage par renforcement. Les figures 5, 6, 7 and 8 montrent les résultats obtenus pour des nageurs à 3, 4 et 5 segements

Conclusion

Dans cette thèse, nous avons présenté une étude de l'apprentissage par renforcement utilisant des réseaux de neurones. Les techniques classiques de la programmation dynamique, des réseaux de neurones et de la programmation neuro-dynamique continue ont été présentées, et des perfectionnements de ces méthodes ont été proposées. Enfin, ces algorithmes ont été appliqués avec succès à des problèmes difficiles de contrôle moteur.

De nombreux résultats originaux ont été présentés dans ce mémore : la notation ∂^* et l'algorithme de rétropropagation différentielle (Appendice A), l'algorithme d'optimization de trajectoires K1999 (Appendice C), et l'équation du second ordre pour les méthodes aux différences finies (1.2.6). En plus de ces résultats, les contributions originales principales de ce travail sont les méthodes d'intégration numérique pour gérer les discontinuités des états et des actions dans le TD(λ), une technique de descente de gradient simple et efficace pour les réseaux de neurones feedforward, et de nombreux résul-

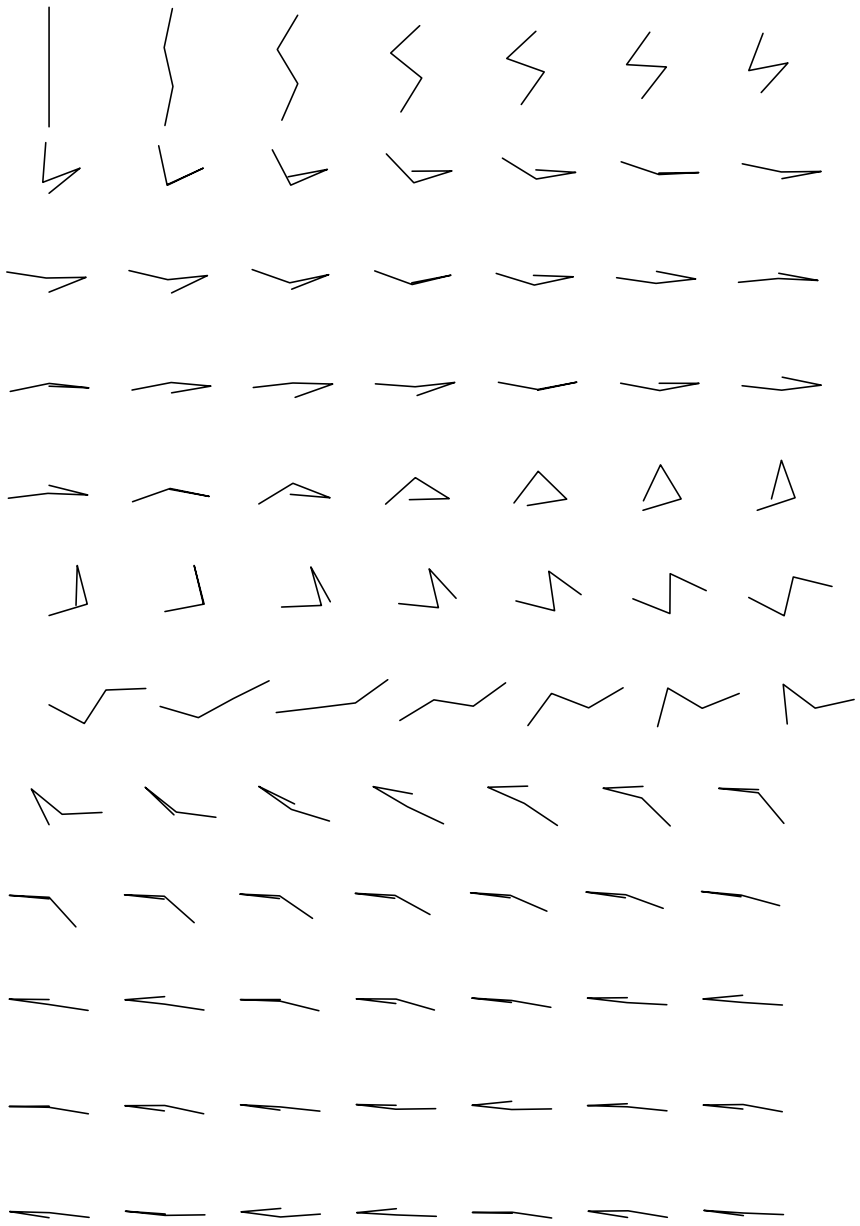


FIG. 5 – Un nageur à 3 segments qui utilise un réseau à 30 neurones. Dans les 4 premières lignes de cette animation, la direction cible est vers la droite. Dans les 8 dernières, elle est inversée vers la gauche. Le nageur est dessiné toutes les 0.1 secondes.

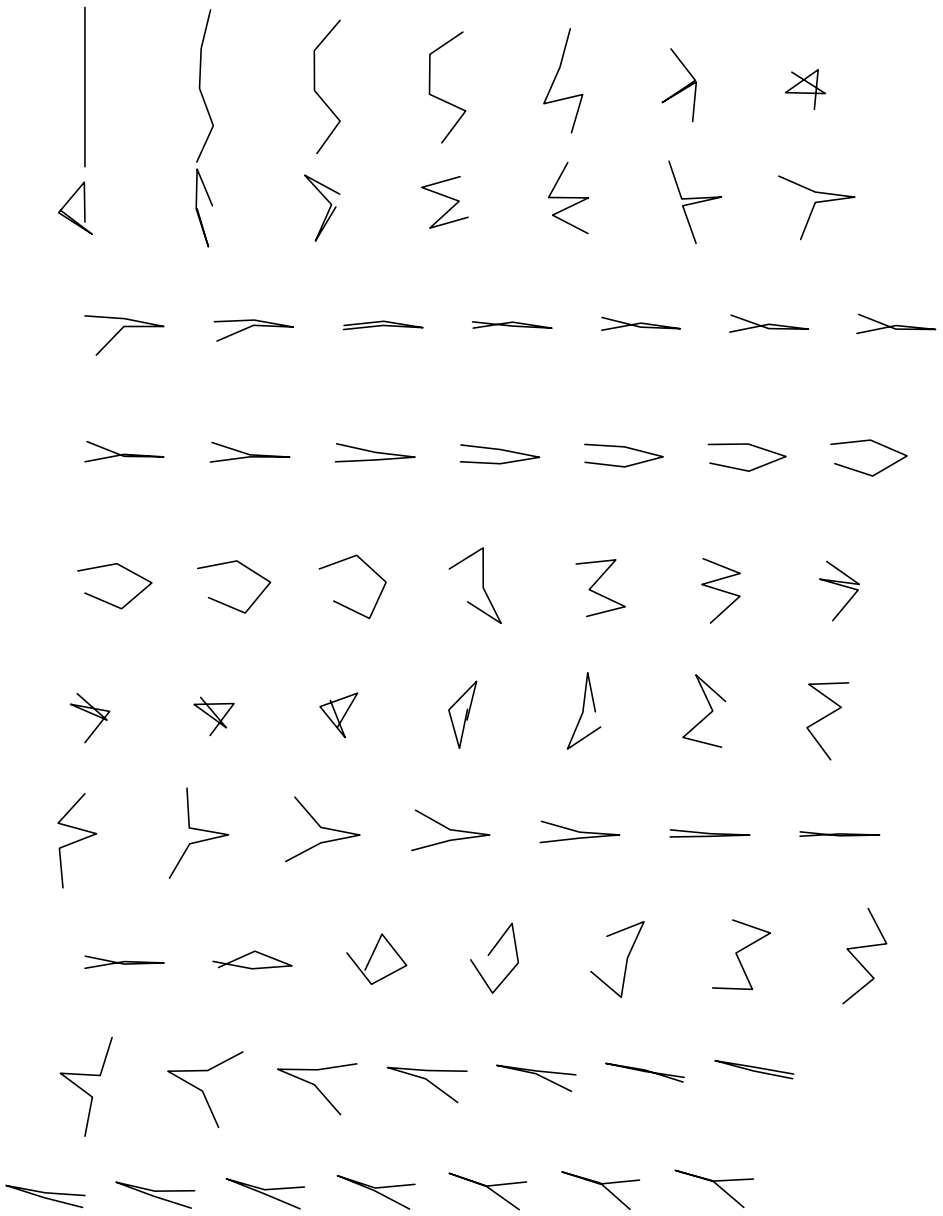


FIG. 6 – Un nageur à 4 segments qui utilise un réseau à 30 neurones. Dans les 7 premières lignes de cette animation, la direction cible est vers la droite. Dans les 3 dernières, elle est inversée vers la gauche. Le nageur est dessiné toutes les 0.2 secondes.

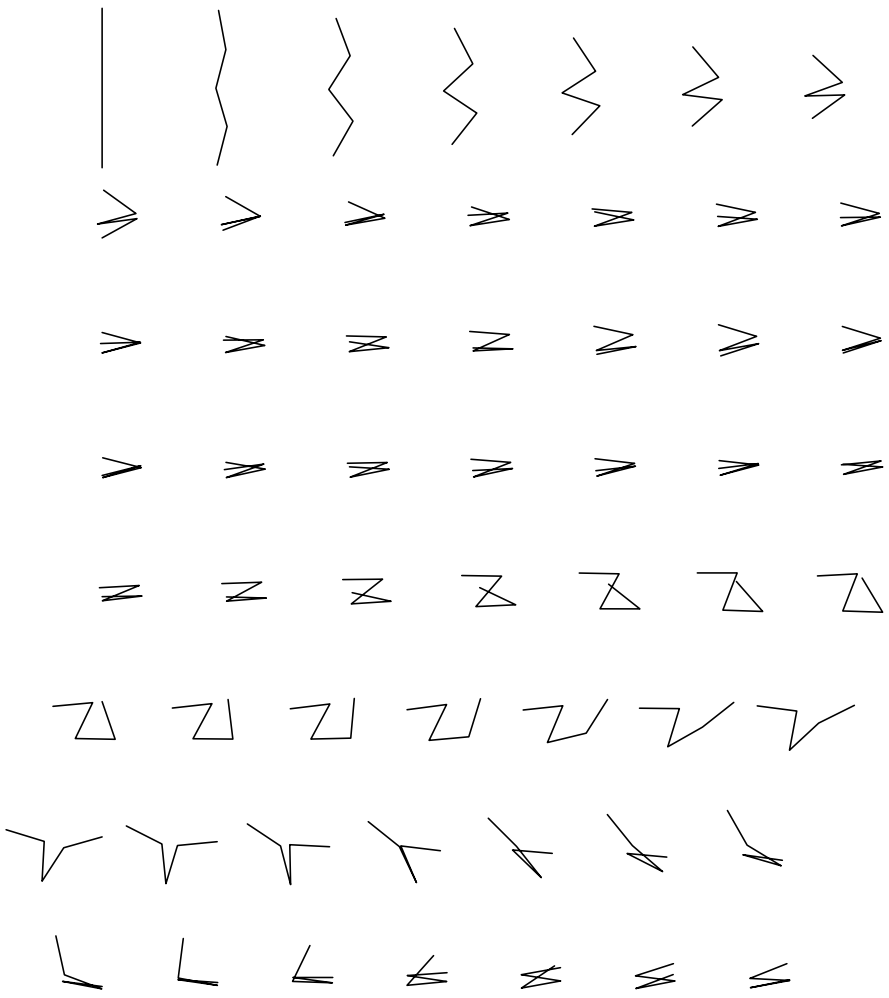


FIG. 7 – Un nageur à 4 segments qui utilise un réseau à 60 neurones. Dans les 4 premières lignes de cette animation, la direction cible est vers la droite. Dans les 4 dernières, elle est inversée vers la gauche. Le nageur est dessiné toutes les 0.1 secondes.

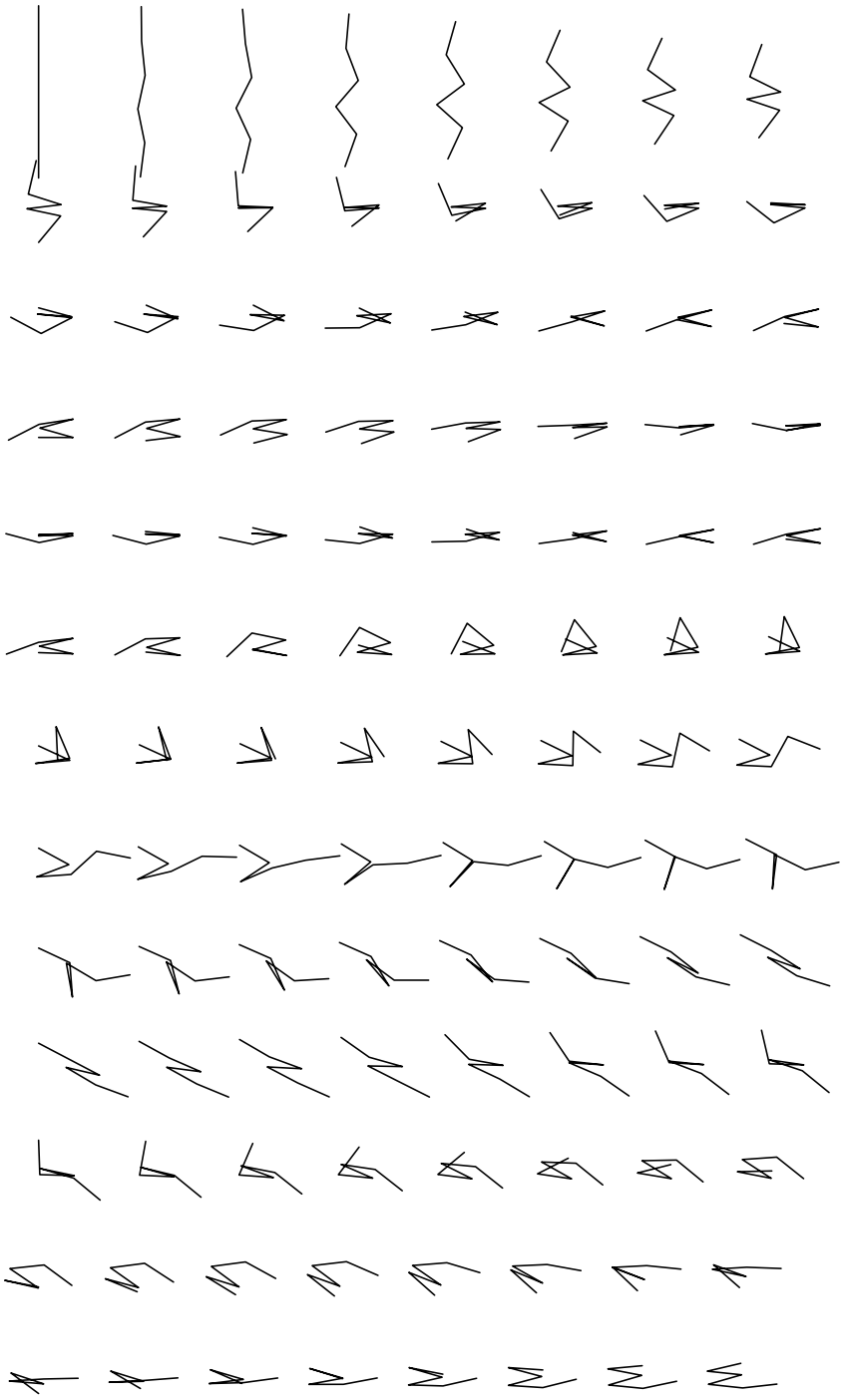


FIG. 8 – Un nageur à 5 segments qui utilise un réseau à 60 neurones. Dans les 4 premières lignes de cette animation, la direction cible est vers la droite. Dans les 8 dernières, elle est inversée vers la gauche. Le nageur est dessiné toutes les 0.1 secondes.

tats expérimentaux originaux sur une grand variété de tâches motrices. La contribution la plus significative est probablement le succès des expériences avec les nageurs (Chapitre 7), qui montre que combiner l'apprentissage par renforcement continu basé sur un modèle avec des réseaux de neurones feedforward peut traiter des problèmes de contrôle moteur qui sont beaucoup plus complexes que ceux habituellement résolus avec des méthodes similaires.

Chacune de ces contributions ouvre aussi des questions et des directions pour des travaux futurs :

- La méthode d'intégration numérique pourrait certainement être améliorée de manière significative. En particulier, l'idée d'utiliser l'information du second ordre sur la fonction valeur pour estimer le contrôle de Filippov pourrait être étendue aux espaces de contrôle à plus d'une dimension.
- Il faudrait comparer la méthode de descente de gradient efficace utilisée dans cette thèse aux méthodes classiques du second ordre dans des tâches d'apprentissage supervisé ou par renforcement.
- De nombreuses expériences nouvelles pourraient être réalisées avec les nageurs. En particulier, il faudrait étudier les raisons des instabilités observées, et des nageurs plus gros pourraient apprendre à nager. Au-delà des nageurs, la méthode utilisée pourrait aussi servir à construire des contrôleurs pour des problèmes beaucoup plus complexes.

En plus de ces extensions directes, une autres question très importante à explorer est la possibilité d'appliquer les réseaux de neurones feedforward hors du cadre restreint du contrôle moteur simulé basé sur la connaissance d'un modèle. En particulier, les expériences indiquent que les réseaux de neurones feedforward demandent beaucoup plus d'épisodes que les approximateurs de fonction linéaires. Cette demande pourrait être un obstacle majeur dans des situations où les données d'apprentissage sont coûteuses à obtenir, ce qui est le cas quand les expériences ont lieu en temps réel (comme dans les expériences de robotique), où quand la sélection des actions implique beaucoup de calculs (comme dans le jeu d'échecs [12]). Ce n'était pas le cas dans les expériences des nageurs, ou avec le joueur de Backgammon de Tesauro, car il était possible de produire, sans coût, autant de données d'apprentissage que nécessaire.

Le problème clé, ici, est la localité. Très souvent, les approximateurs de fonction linéaires sont préférés, parce que leur bonne localité leur permet de faire de l'apprentissage incrémental efficacement, alors que les réseaux de neurones feedforward ont tendance à désapprendre l'expérience passée quand de nouvelles données d'apprentissage sont traitées. Cependant, Les performances des nageurs obtenues dans cette thèse indiquent clairement que les réseaux de neurones feedforward peuvent résoudre des problèmes qui sont

plus complexes que ce que les approximateurs de fonctions linéaires peuvent traiter. Il serait donc naturel d'essayer de combiner les qualités de ces deux schémas d'approximation.

Créer un approximateur de fonction qui aurait à la fois la localité des approximateurs de fonctions linéaires, et les capacités de généralisation des réseaux de neurones feedforward semble très difficile. Weaver *et al.* [78] ont proposé un algorithme d'apprentissage spécial qui permet d'éviter le désapprentissage. Son efficacité dans l'apprentissage par renforcement en dimension élevée reste à démontrer, mais cela pourrait être une direction de recherche intéressante.

Une autre possibilité pour faire un meilleur usage de données d'apprentissage peu abondantes consisterait à utiliser, en complément de l'algorithme d'apprentissage par renforcement, une forme de mémoire à long terme qui stocke ces données. Après un certain temps, l'algorithme d'apprentissage pourrait rappeler ces données stockées pour vérifier qu'elles n'ont pas été oubliées par le réseau de neurones feedforward. Une difficulté majeure de cette approche est qu'elle demanderait un sorte de TD(λ) hors-stratégie, car l'algorithme d'apprentissage observerait des trajectoires qui ont été générées avec une fonction valeur différente.

Introduction

Introduction

Building automatic controllers for robots or mechanisms of all kinds has been a great challenge for scientists and engineers, ever since the early days of the computer era. The performance of animals in the simplest motor tasks, such as walking or swimming, turns out to be extremely difficult to reproduce in artificial mechanical devices, simulated or real. This thesis is an investigation of how some techniques inspired by Nature—artificial neural networks and reinforcement learning—can help to solve such problems.

Background

Finding optimal actions to control the behavior of a dynamical system is crucial in many important applications, such as robotics, industrial processes, or spacecraft flying. Some major research efforts have been conducted to address the theoretical issues raised by these problems, and to provide practical methods to build efficient controllers.

The classical approach of numerical optimal control consists in computing an optimal trajectory first. Then, a controller can be built to track this trajectory. This kind of method is often used in astronautics, or for the animation of artificial movie characters. Modern algorithms can solve complex problems such as, for instance, Hardt *et al.*'s optimal simulated human gait [30].

Although these methods can deal accurately with very complex systems, they have some limitations. In particular, computing an optimal trajectory is often too costly to be performed online. This is not a problem for space probes or animation, because knowing one single optimal trajectory in advance is enough. In some other situations, however, the dynamics of the system might not be completely predictable, and it might be necessary to find new optimal actions quickly. For instance, if a walking robot stumbles over an unforeseen obstacle, it must react rapidly to recover its balance.

In order to deal with this problem, some other methods have been designed. They allow to build controllers that produce optimal actions in any

situation, not only in the neighborhood of a pre-computed optimal trajectory. This is a much more difficult task than finding one single path, so these techniques usually do not perform as well as classical numerical optimal control on applications where both can be used.

A first possibility consists in using a neural network (or any kind of function approximator) with a supervised learning algorithm to generalize controls from a set of trajectories. These trajectories can be obtained by recording actions of human "experts" or by generating them with methods of numerical optimal control. The latter technique is used by Lachner *et al.*'s collision-avoidance algorithm [35], for instance.

Another solution consists in directly searching a set of controllers with an optimization algorithm. Van de Panne [50] combined stochastic search and gradient descent to optimize controllers. Genetic algorithms are also well suited to perform this optimization, because the space that is searched often has a complex structure. Sims [63, 62] used this technique to evolve very spectacular virtual creatures that can walk, swim, fight or follow a light source. Many other research works produced controllers thanks to genetic algorithms, such as, for instance Meyer *et al*'s [38].

Lastly, a wide family of techniques to build such controllers is based on the principles of dynamic programming, which were introduced by Bellman in the early days of control theory [13]. In particular, the theory of reinforcement learning (or neuro-dynamic programming, which is often considered as a synonym) has been successfully applied to a large variety of control problems. It is this approach that will be developed in this thesis.

Reinforcement Learning using Neural Networks

Reinforcement learning is learning to act by trial and error. In this paradigm, an agent can perceive its state and perform actions. After each action, a numerical reward is given. The goal of the agent is to maximize the total reward it receives over time.

A large variety of algorithms have been proposed that select actions in order to explore the environment, and gradually build a strategy that tends to obtain a maximum reward [68, 33]. These algorithms have been successfully applied to complex problems such as board games [70], job-shop scheduling [81], elevator dispatching [20], and motor control tasks, either simulated [67, 24], or real [41, 59].

Model-Based *versus* Model-Free

These reinforcement learning algorithms can be divided into two categories: model-based (or indirect) algorithms, which use an estimation of the system's dynamics, and model-free (or direct) algorithms, which do not. Whether one approach is better than the other is not clear, and depends a lot on the specific problem to be solved. The main advantages provided by a model is that actual experience can be complemented by simulated ("imaginary") experience, and that the knowledge of state values is enough to find the optimal control. The main drawbacks of model-based algorithms are that they are more complex (because a mechanism to estimate the model is required), and simulated experience produced by the model might not be accurate (which may mislead the learning process).

Although it is not obvious which is the best approach, some research results tend to indicate that model-based reinforcement learning can solve motor-control problems more efficiently. This has been shown in simulations [5, 24] and also in experiments with real robots. Morimoto and Doya combined simulated experience with real experience to teach a robot to stand up [42]. Schaal and Atkeson also used model-based reinforcement learning in their robot-juggling experiments [59].

Neural Networks

Almost all reinforcement learning algorithms involve estimating value functions that indicate how good it is to be in a given state (in terms of total expected reward in the long term), or how good it is to perform a given action in a given state. The most basic way to build this value function consists in updating a table that contains a value for each state (or each state-action pair), but this approach is not practical for large scale problems. In order to deal with tasks that have a very large number of states, it is necessary to use the generalization capabilities of function approximators.

Feedforward neural networks are a particular case of such function approximators that can be used in combination with reinforcement learning. The most spectacular success of this technique is probably Tesauro's backgammon player [70], which managed to reach the level of human masters after months of self-play. In backgammon, the estimated number of possible positions is about 10^{20}. A value function over such a number of states cannot be stored in a lookup-table.

Summary and Contributions

Problem

The aim of the research reported in this dissertation is to find efficient methods to build controllers for simulated motor control tasks. Simulation means that an exact model of the system to be controlled is available. In order to avoid imposing artificial constraints, learning algorithms will be supposed to have access to this model. This assumption is an important limitation, but it still provides a lot of challenges, and progress made within this limited framework can be transposed to the more general case where a model has to be learnt.

Approach

The technique used to tackle this problem is Doya's continuous TD(λ) reinforcement learning algorithm [23]. It is a continuous formulation of Sutton's classical discrete TD(λ) [66] that is well adapted to motor control problems. Its efficiency was demonstrated by successfully learning to swing up a rotating pole mounted on a moving cart [24].

In many of the reinforcement learning experiments in the domain of motor control, a linear function approximator has been used to approximate the value function. This approximation scheme has many interesting properties, but its ability to deal with a large number of independent state variables is not very good.

The main originality of the approach followed in this thesis is that the value function is estimated with feedforward neural networks instead of linear function approximators. The nonlinearity of these neural networks makes them difficult to harness, but their excellent ability to generalize in high-dimensional input spaces might allow them to solve problems that are orders of magnitude more complex than what linear function approximators can handle.

Contributions

This work explores the numerical issues that have to be solved in order to improve the efficiency of the continuous TD(λ) algorithm with feedforward neural networks. Some of its main contributions are:

- A method to deal with discontinuous control. In many problems, the optimal control is discontinuous, which makes it difficult to apply efficient numerical integration algorithms. We show how Filippov control

can be obtained by using second order information about the value function.

- A method to deal with discontinuous states, that is to say hybrid control problems. This is necessary to apply continuous TD(λ) to problems with shocks or discontinuous inputs.
- The Vario-η algorithm [47] is proposed as practical method to perform gradient descent in reinforcement learning.
- Many experimental results that clearly indicate the huge potential of feedforward neural networks in reinforcement learning applied to motor control. In particular, a complex articulated swimmer with 12 independent state variables and 4 control variables learnt to swim thanks to feedforward neural networks.

Outline

- Part I: Theory
  - Chapter 1: Dynamic Programming
  - Chapter 2: Neural Networks
  - Chapter 3: Neuro-Dynamic Programming
  - Chapter 4: TD(λ) in Practice
- Part II: Experiments
  - Chapter 5: Classical Problems
  - Chapter 6: Robot Auto Racing Simulator
  - Chapter 7: Swimmers

Part I

Theory

Chapter 1

Dynamic Programming

Dynamic programming is a fundamental tool in the theory of optimal control, which was developed by Bellman in the fifties [13, 14]. The basic principles of this method are presented in this chapter, in both the discrete and the continuous case.

1.1 Discrete Problems

The most basic category of problems that dynamic programming can solve are problems where the system to be controlled can only be in a finite number of states. Motor control problems do not belong to this category, because a mechanical system can be in a continuous infinity of states. Still, it is interesting to study discrete problems, since they are much simpler to analyze, and some concepts introduced in this analysis can be extended to the continuous case.

1.1.1 Finite Discrete Deterministic Decision Processes

A finite discrete deterministic decision process (or control problem) is formally defined by

- a finite set of states S.
- for each state x, a finite set of actions $U(x)$.
- a transition function ν that maps state-action pairs to states. $\nu(x, u)$ is the state into which the system jumps when action u is performed in state x.

- a reward function r that maps state-action pairs to real numbers. $r(x, u)$ is the reward obtained for performing action u in state x. The goal of the control problem is to maximize the total reward obtained over a sequence of actions.

A *strategy* or *policy* is a function $\pi : S \mapsto U$ that maps states to actions. Applying a policy from a starting state x_0 produces a sequence of states $x_0, x_1, x_2, \ldots$ that is called a *trajectory* and is defined by

$$\forall i \in \mathbb{N} \quad x_{i+1} = \nu\big(x_i, \pi(x_i)\big).$$

Cumulative reward obtained over such a trajectory depends only on π and x_0. The function of x_0 that returns this total reward is called the *value function* of π. It is denoted V^π and is defined by

$$V^\pi(x_0) = \sum_{i=0}^{\infty} r\big(x_i, \pi(x_i)\big).$$

A problem with this sum is that it may diverge. $V^\pi(x_0)$ converges only when a limit cycle with zero reward is reached. In order to get rid of these convergence issues, a discounted reward is generally introduced, where each term of the sum is weighted by an exponentially decaying coefficient:

$$V^\pi(x_0) = \sum_{i=0}^{\infty} \gamma^i r\big(x_i, \pi(x_i)\big).$$

γ is a constant ($\gamma \in [0, 1[$) called the *discount factor*. The effect of γ is to introduce a time horizon to the value function: the smaller γ, the more short-sighted V^π.

The goal is to find a policy that maximizes the total amount of reward over time, whatever the starting state x_0. More formally, the optimal control problem consists in finding π^* so that

$$\forall x_0 \in S \quad V^{\pi^*}(x_0) = \max_{\pi : S \mapsto U} V^\pi(x_0).$$

It is easy to prove that such a policy exists. It might not be unique, however, since it is possible that two different policies lead to the same cumulative reward from a given state. V^{π^*} does not depend on π^* and is denoted V^*. It is called the optimal value function.

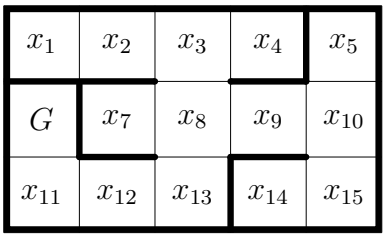


Figure 1.1: Example of a discrete deterministic control problem: from any starting state x, move in the maze to reach the goal state G, without crossing the heavy lines, which represent walls

1.1.2 Example

Figure 1.1 shows a simple discrete deterministic control problem. The goal of this control problem is to move in a maze and reach a goal state G as fast as possible. This can fit into the formalism defined previously:

- S is the set of the 15 squares that make up the maze.
- Possible actions in a specific state are a subset of $\{Up, Down, Left, Right, NoMove\}$. The exact value of $U(x)$ depends on the walls that surround state x. For instance, $U(x_5) = \{Down, NoMove\}$.
- The transition function is defined by the map of the maze. For instance, $\nu(x_8, Down) = x_{13}$, $\nu(x_9, NoMove) = x_9$.

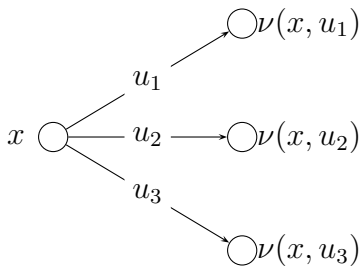
The reward function has to be chosen so that maximizing the reward is equivalent to minimizing the number of steps necessary to reach the goal. A possible choice for r is -1 everywhere, except at the goal:

$$\begin{aligned} \forall x \neq G \quad \forall u \in U(x) \quad & r(x,u) = -1, \\ \forall u \in U(G) \quad & r(G,u) = 0. \end{aligned}$$

This way, the optimal value function is equal to the opposite of the number of steps needed to reach the goal.

1.1.3 Value Iteration

Solving a discrete deterministic decision process is an optimization problem over the set of policies. One big difficulty with this problem is that the number of policies may be huge. For instance, if $|U|$ does not depend on current state, then there are $|U|^{|S|}$ policies. So, exploring this set directly to find the optimal one may be very costly. In order to avoid this difficulty, the



$$V^*(x) = \max_{u \in \{u_1, u_2, u_3\}} \Big(r(x,u) + V^*\big(\nu(x,u)\big) \Big)$$

Figure 1.2: Bellman's equation. Possible actions in state x are u_1, u_2, and u_3. If the optimal value is known for the corresponding successor states, then this formula gives the optimal value of state x.

basic idea of dynamic programming consists in evaluating the optimal value function V^* first. Once V^* has been computed, it is possible to obtain an optimal policy by taking a greedy action with respect to V^*, that is to say

$$\pi^*(x) = \arg \max_{u \in U(x)} \Big(r(x,u) + \gamma V^*\big(\nu(x,u)\big) \Big).$$

So, the problem is reduced to estimating the optimal value function V^*. This can be done thanks to *Bellman's equation* (Figure 1.2), which gives the value of a state x as a function of the values of possible successor states $\nu(x,u)$:

$$V^*(x) = \max_{u \in U(x)} \Big(r(x,u) + V^*\big(\nu(x,u)\big) \Big).$$

When using discounted reward, this equation becomes

$$V^*(x) = \max_{u \in U(x)} \Big(r(x,u) + \gamma V^*\big(\nu(x,u)\big) \Big). \tag{1.1.1}$$

So, using $\vec{V}$ to denote the vector of unknown values:

$$\vec{V} = \big(V(x_1)\ V(x_2)\ \ldots\ V(x_n)\big)^t,$$

then the optimal value function is a solution of an equation of the type

$$\vec{V} = g(\vec{V}).$$

That is to say it is a fixed point of g. A solution of this equation can be obtained by an algorithm that iteratively applies g to an estimation of the value function:

$\vec{V} \leftarrow \vec{0}$
repeat
  $\vec{V} \leftarrow g(\vec{V})$
until convergence.

Algorithm 1.1 explicitly shows the details of this algorithm called *value iteration* for discrete deterministic control problems. Figures 1.3 and 1.4 illustrate its application to the maze problem[1].

Algorithm 1.1 Value Iteration

for all $x \in S$ **do**
  $V_0(x) \leftarrow 0$
end for
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  for all $x \in S$ **do**
    $V_i(x) \leftarrow \max_{u \in U(x)} \Big(r(x,u) + \gamma V_{i-1}\big(\nu(x,u)\big) \Big)$
  end for
until V has converged

When discounted reward is used ($\gamma < 1$), it is rather easy to prove that value iteration always converges. The proof is based on the fact that g is a contraction with a factor equal to γ: for two estimations of the value function, $\vec{V}_1$ and $\vec{V}_2$,

$$\left\| g(\vec{V}_1) - g(\vec{V}_2) \right\|_\infty \leq \gamma \left\| \vec{V}_1 - \vec{V}_2 \right\|_\infty .$$

Convergence of the value-iteration algorithm can be proved easily thanks to this property.

When reward is not discounted, however, convergence is a little more difficult to prove. Value iteration can actually diverge in this case, since the sum of rewards may be infinite. For instance, this could happen in the maze problem if some states were in a "closed room". There would be no path to reach the goal and the value function would diverge to $-\infty$ at these states. Nevertheless, when the optimal value is well-defined, it is possible to prove that value iteration does converge to the optimal value function. Notice that it is important that $\vec{V}$ be initialized to $\vec{0}$ in this case (this was not necessary in the discounted case, since the contraction property ensures convergence from any initial value of $\vec{V}$.)

[1] Other algorithms (such as Dijkstra's) work better than value iteration for this kind of maze problems. Unfortunately, they do not work in the general case, so they will not be explained in details here.

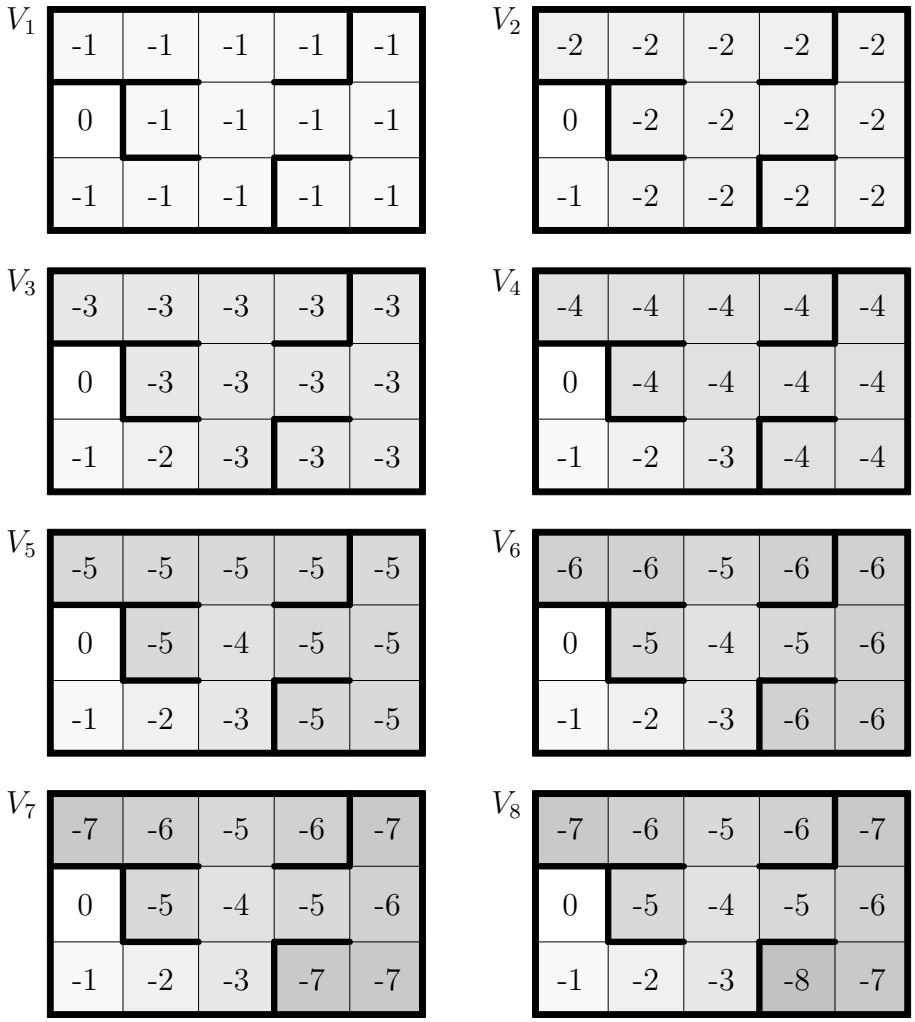


Figure 1.3: Application of value iteration: the value function is initialized with null values (V_0), and Bellman's equation is applied iteratively until convergence (see Algorithm 1.1).

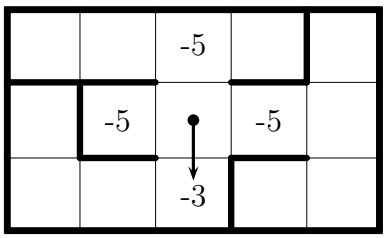


Figure 1.4: Optimal control can be found by a local observation of the value function.

Value iteration can be proved to have a computational cost polynomial in $|U|$ and $|S|$. Although this might still be very costly for huge state or action spaces, value iteration usually takes much less time than exploring the whole set of policies.

1.1.4 Policy Evaluation

Another task of interest in finite deterministic decision processes is the one of evaluating a fixed policy π. It is possible to deal with this problem in a way that is very similar to value iteration, with the only difference that the set of equations to be solved is, for all states x,

$$V^\pi(x) = r\big(x, \pi(x)\big) + \gamma V^\pi\Big(\nu\big(x, \pi(x)\big)\Big).$$

The same kind of fixed point algorithm can be used, which leads to Algorithm 1.2. Convergence of this algorithm can be proved thanks to the contraction property when $\gamma < 1$. It also converges when $\gamma = 1$ and all values are well-defined.

Algorithm 1.2 Policy Evaluation

for all $x \in S$ **do**
  $V_0(x) \leftarrow 0$
end for
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  for all $x \in S$ **do**
    $V_i(x) \leftarrow r\big(x, \pi(x)\big) + \gamma V_{i-1}\Big(\nu\big(x, \pi(x)\big)\Big)$
  end for
until V has converged

1.1.5 Policy Iteration

Policy Iteration is another very important approach to dynamic programming. It is attributed to Howard [31], and consists in using the policy-evaluation algorithm defined previously to obtain successive improved policies. Algorithm 1.3 shows the details of this algorithm. It is rather easy to prove that, for each x, $V_i(x)$ is bounded and monotonic, which proves that this algorithm converges when $\gamma < 1$ or when $\gamma = 1$ and π_0 is a proper strategy (that is to say, a strategy with a well-defined value function).

Algorithm 1.3 Policy Iteration

$\pi_0 \leftarrow$ an arbitrary policy
$i \leftarrow 0$
repeat
  $V_i \leftarrow$ evaluation of policy π_i
  $\pi_{i+1} \leftarrow$ a greedy policy on V_i
  $i \leftarrow i + 1$
until V has converged or π has converged

1.2 Continuous Problems

The formalism defined previously in the discrete case can be extended to continuous problems. This extension is not straightforward because the number of states is infinite (so, the value function can not be stored as a table of numerical values), and time is continuous (so, there is no such thing as a "next state" or a "previous state"). As a consequence, discrete algorithms cannot be applied directly and have to be adapted.

1.2.1 Problem Definition

The first element that has to be adapted to the continuous case is the definition of the problem. A continuous deterministic decision process is defined by:

- A state space $S \subset \mathbb{R}^p$. This means that the state of the system is defined by a vector $\vec{x}$ of p real valued variables. In the case of mechanical systems, these will typically be angles, velocities or positions.

- A control space $U \subset \mathbb{R}^q$. The controller can influence the behavior of the system via a vector $\vec{u}$ of q real valued variables. These will typically be torques, forces or engine throttle. U may depend on the state $\vec{x}$. $\vec{u}$ is also called the action.

- System dynamics $f : S \times U \mapsto \mathbb{R}^p$. This function maps states and actions to derivatives of the state with respect to time. That is to say $\dot{\vec{x}} = f(\vec{x}, \vec{u})$. This is analogous to the $\nu(x, u)$ function, except that a derivative is used in order to deal with time continuity.

- A reward function $r : S \times U \mapsto \mathbb{R}$. The problem consists in maximizing the cumulative reward as detailed below.

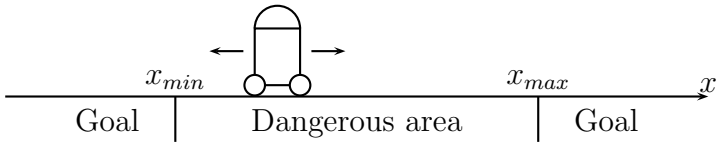


Figure 1.5: A simple optimal control problem in one dimension: get the robot out of the dangerous area as fast as possible

- A shortness factor $s_\gamma \geq 0$. This factor measures the short-sightedness of the optimization. s_γ plays a role that is very similar to the discount factor γ in the discrete case. These values can be related to each other by $\gamma = e^{-s_\gamma \delta t}$, where δt is a time step. If $s_\gamma = 0$, then the problem is said to be *non-discounted*. If $s_\gamma > 0$ the problem is *discounted* and $1/s_\gamma$ is the typical time horizon.

A *strategy* or *policy* is a function $\pi : S \mapsto U$ that maps states to actions. Applying a policy from a starting state $\vec{x}_0$ at time t_0 produces a trajectory $\vec{x}(t)$ defined by the ordinary differential equation

$$\begin{aligned} \forall t \geq t_0 \quad \dot{\vec{x}} &= f\big(\vec{x}, \pi(\vec{x})\big), \\ \vec{x}(t_0) &= \vec{x}_0. \end{aligned}$$

The value function of π is defined by

$$V^\pi(\vec{x}_0) = \int_{t=t_0}^{\infty} e^{-s_\gamma (t-t_0)} r\Big(\vec{x}(t), \pi\big(\vec{x}(t)\big)\Big) dt. \tag{1.2.1}$$

The goal is to find a policy that maximizes the total amount of reward over time, whatever the starting state $\vec{x}_0$. More formally, the optimal control problem consists in finding π^* so that

$$\forall \vec{x}_0 \in S \quad V^{\pi^*}(\vec{x}_0) = \max_{\pi : S \mapsto U} V^\pi(\vec{x}_0).$$

Like in the discrete case, V^{π^*} does not depend on π^*. It is denoted V^* and is called the optimal value function.

1.2.2 Example

Figure 1.5 shows a very simple problem that can fit in this general formalism. It consists in finding a time-optimal control for a robot to move out of a dangerous area. The robot is controlled by a command that sets its velocity.

- The state space is the set of positions the robot can take. It is equal to the $S = [x_{min}; x_{max}]$ segment. The robot may have a position that is outside this interval, but this case is of little interest, as the problem of finding an optimal control only makes sense in the dangerous area. Any control is acceptable outside of it. The dimensionality of the state space is 1 ($p = 1$).

- The control space is the set of possible velocity commands. We will suppose it is the interval $U = [v_{min}; v_{max}]$. This means that the dimensionality of the control space is also 1 ($q = 1$).

- The time derivative of the robot's position is the velocity command. So, the dynamics is defined by $f(x, u) = u$. In order to prevent the robot from getting out of its state space, boundary states are absorbing, that is to say $f(x_{min}, u) = 0$ and $f(x_{max}, u) = 0$.

The previous elements of the optimal control problem have been taken directly from the mechanical specifications of the system to be controlled, but we are left with the choice of the shortness factor and the reward function. There is actually an infinite number of possible values of these parameters that can be used to find the time-optimal path for the robot. It is often important to make the choice that will be easiest to handle by the method used to solve the control problem. Here, they are chosen to be similar to the maze problem described in section 1.1.2:

- The goal of the present optimal control problem is to reach the boundary of the state space as fast as possible. To achieve this, a constant negative reward $r(x, u) = -1$ can be used inside the state space, and a null reward at boundary states ($r(x_{min}, u) = 0$) and $r(x_{max}, u) = 0$). Thus, maximizing the total reward is equivalent to minimizing the time spent in the state space.

- $s_\gamma = 0$. This choice will make calculations easier. Any other value of s_γ would have worked too.

If t_0 is the starting time of a trial, and t_b the time when the robot reaches the boundary, then

$$V^\pi\big(\vec{x}(t_0)\big) = \int_{t=t_0}^{t_b} (-1)dt + \int_{t=t_b}^{\infty} 0dt = t_0 - t_b.$$

This means that the value function is equal to the opposite of the time spent in the dangerous area. Figure 1.6 shows some value functions for three

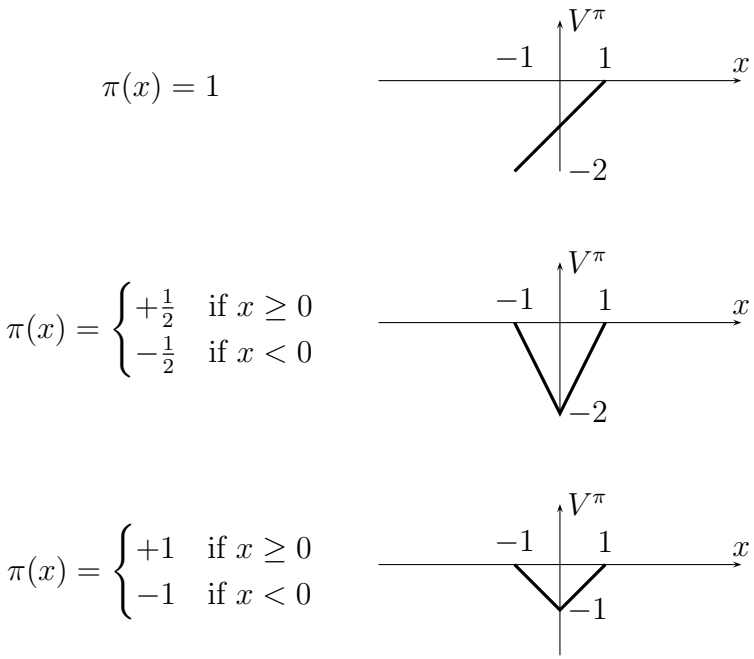


Figure 1.6: Examples of value functions for different policies π

different policies (numerical values are[2] $x_{min} = -1$, $x_{max} = +1$, $v_{min} = -1$, and $v_{max} = +1$.) It is intuitively obvious that the third policy is optimal. It consists in going at maximum speed to the right if the robot is on the right side of the dangerous area and at maximum speed to the left if the robot is on its left side.

1.2.3 Problem Discretization

The optimal policy was very easy to guess for this simple problem, but such an intuitive solution cannot be found in general. In order to find a method that works with all problems, it is possible to apply some form of discretization to the continuous problem so that techniques presented in the first section of this chapter can be applied.

Discretization of the Robot Problem

Let us try to apply this idea to the one-dimensional robot problem. In order to avoid confusion, discrete S, U and r will be denoted S_d, U_d and r_d. It is

[2] All physical quantities are expressed in SI units

possible to define an "equivalent" discrete problem this way:

- $S_d = \{\frac{-9}{8}, \frac{-7}{8}, \frac{-5}{8}, \frac{-3}{8}, \frac{-1}{8}, \frac{+1}{8}, \frac{+3}{8}, \frac{+5}{8}, \frac{+7}{8}, \frac{+9}{8}\}$, as shown on Figure 1.7.
- $U_d = \{-1, 0, +1\}$
- In order to define the ν function, a fixed time step $\delta t = \frac{1}{4}$ can be used. This way, $\nu(\frac{1}{8}, +1) = \frac{3}{8}$. More generally, $\nu(x, u) = x + u\delta t$, except at boundaries ($\nu(\frac{-9}{8}, -1) = \frac{-9}{8}$ and $\nu(\frac{9}{8}, +1) = \frac{9}{8}$).
- $r_d(x, u) = -\delta t$ except at boundaries, where $r_d(x, u) = 0$. This way, the total reward is still equal to the opposite of the total time spent in the dangerous area.
- $\gamma = 1$

Figure 1.8 shows the approximate value function obtained by value iteration for such a discretization. It is very close to the V shape of the optimal value function.

General Case

In the general case, a finite number of sample states and actions have to be chosen to make up the state and action sets: $S_d \subset S$ and $U_d \subset U$. These sample elements should be chosen to be representative of the infinite set they are taken from.

Once this has been done, it is necessary to define state transitions. It was rather easy with the robot problem because it was possible to choose a constant time step so that performing an action during this time step lets the system jump from one discrete state right into another one. Unfortunately, this can not be done in the general case (see Figure 1.9), so it is not always possible to transform a continuous problem to make it fit into the discrete deterministic formalism.

Although there is no hope to define discrete deterministic state transitions for a continuous problem, it is still possible to apply dynamic programming algorithms to a state discretization. The key issue is to find an equivalent to the discrete Bellman equation. So, let us consider a time step of length δt. It is possible to split the sum that defines the value function (1.2.1) into two parts:

$$V^\pi(\vec{x}_0) = \int_{t=t_0}^{t_0+\delta t} e^{-s(t-t_0)} r\Big(\vec{x}(t), \pi\big(\vec{x}(t)\big)\Big) dt + e^{-s\delta t} V^\pi\big(\vec{x}(t_0+\delta t)\big). \quad (1.2.2)$$

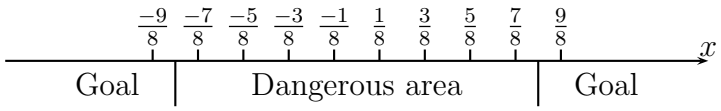


Figure 1.7: Discretization of the robot's state space

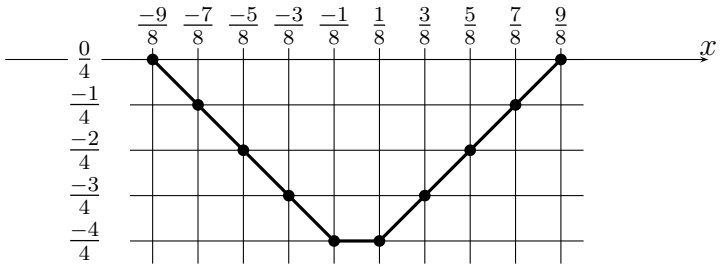


Figure 1.8: Value function obtained by value iteration

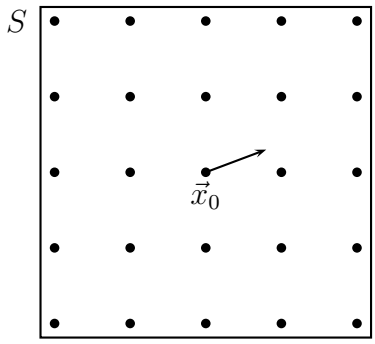


Figure 1.9: Dots represent the set of discrete states (S_d). In general, performing an action in a discrete state $\vec{x}_0$ cannot jump right into another nearby discrete state, whatever the time step.

When δt is small, this can be approximated by

$$V^\pi(\vec{x}_0) \approx r\big(\vec{x}_0, \pi(\vec{x}_0)\big)\delta t + e^{-s\delta t} V^\pi(\vec{x}_0 + \delta\vec{x}), \tag{1.2.3}$$

with

$$\delta\vec{x} = f\big(\vec{x}_0, \pi(\vec{x}_0)\big)\delta t.$$

Thanks to this discretization of time, it is possible to obtain a semi-continuous Bellman equation that is very similar to the discrete one (1.1.1):

$$V^*(\vec{x}) \approx \max_{\vec{u}\in U_d} \big(r(\vec{x},\vec{u})\delta t + e^{-s\delta t} V^*(\vec{x}+\delta\vec{x})\big). \tag{1.2.4}$$

In order to solve equation (1.2.4), one might like to try to replace it by an assignment. This would allow to iteratively update $V(\vec{x})$ for states $\vec{x}$ in S_d, similarly to the discrete value iteration algorithm. One major obstacle to this approach, however, is that $\vec{x}+\delta\vec{x}$ is not likely to be in S_d. In order to overcome this difficulty, it is necessary to use some form of interpolation to estimate $V(\vec{x}+\delta\vec{x})$ from the values of discrete states that are close to $\vec{x}+\delta\vec{x}$. Algorithm 1.4 shows this general algorithm for value iteration.

Algorithm 1.4 Semi-Continuous Value Iteration

for all $\vec{x} \in S_d$ **do**
  $V_0(\vec{x}) \leftarrow 0$
end for
$i \leftarrow 0$
repeat
  $i \leftarrow i+1$
  for all $\vec{x} \in S_d$ **do**
    $V_i(\vec{x}) \leftarrow \max_{\vec{u}\in U_d} \Big(r(\vec{x},\vec{u})\delta t + e^{-s_\gamma \delta t} \underbrace{V_{i-1}\big(\vec{x}+f(\vec{x},\vec{u})\delta t\big)}_{\text{estimated by interpolation}}\Big)$
  end for
until V has converged

Finite Difference Method

Algorithm 1.4 provides a general framework for continuous value iteration, but a lot of its elements are not defined accurately: how to choose S_d, U_d, δt? how to interpolate between sample states? Many methods have been designed to make these choices so that value iteration works efficiently. One of the simplest is the finite difference method.

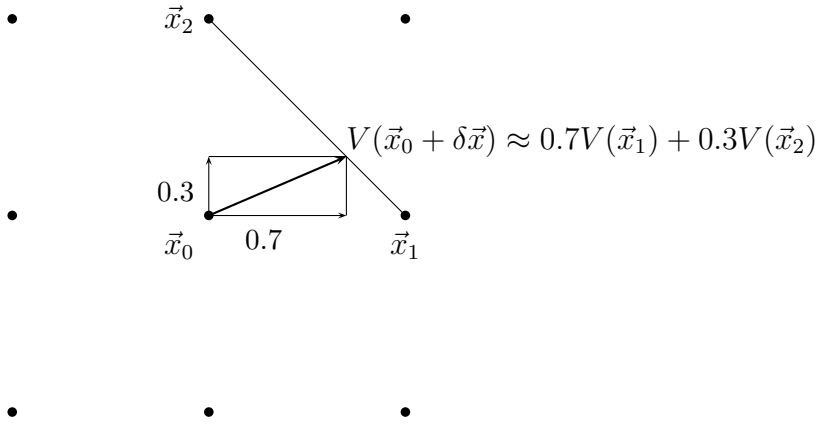


Figure 1.10: Finite Difference Method

This method consists in using a rectangular grid for S_d. The time step δt is chosen so that applying action $\vec{u}$ during a time interval of length δt moves the state to an hyperplane that contains nearby states (Figure 1.10). The value function is estimated at $\vec{x} + \delta\vec{x}$ by linear interpolation between these nearby states.

A problem with this method is that the time it takes to move to nearby states may be very long when $\|f(\vec{x}, \vec{u})\|$ is small. In this case, the finite difference method does not converge to the right value function because the small-time-step approximation (1.2.3) is not valid anymore. Fortunately, when $\delta\vec{x}$ is small and δt is not, it is possible to obtain a more accurate Bellman equation. It simply consists in approximating (1.2.2) by supposing that r is almost constant in the integral sum:

$$V^\pi(\vec{x}) \approx r\big(\vec{x}, \pi(\vec{x})\big) \frac{1 - e^{-s_\gamma \delta t}}{s_\gamma} + e^{-s_\gamma \delta t} V^\pi(\vec{x} + \delta\vec{x}), \tag{1.2.5}$$

which is a good approximation, even if δt is large. This equation can be simplified into

$$V^\pi(\vec{x}) \approx \frac{r\big(\vec{x}, \pi(\vec{x})\big)\delta t + (1 - s_\gamma \delta t/2) V^\pi(\vec{x} + \delta\vec{x})}{1 + s_\gamma \delta t/2}, \tag{1.2.6}$$

which keeps a second order accuracy in δt. Thanks to this equation, finite difference methods converge even for problems that have stationary states. Besides, (1.2.6) is not only more accurate than (1.2.3), but it is also more computationally efficient since there is no costly exponential to evaluate.

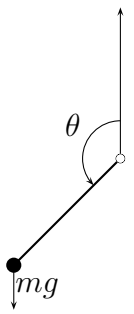


Figure 1.11: The pendulum swing-up problem

Convergence

When an averaging interpolation is used, it is easy to prove that this algorithm converges. Similarly to the discrete case, this result is based on a contraction property, with a factor equal to $e^{-s_\gamma \delta t}$ (or $\left|\frac{1-s_\gamma \delta t/2}{1+s_\gamma \delta t/2}\right|$ when (1.2.6) is used).

This convergence result, however, does not give any indication about how close to the exact value function it converges (that is to say, how close to the value of the continuous problem). In particular, (1.2.3) can give a significantly different result from what (1.2.6) gives. Although this discretization technique has been known since the early days of dynamic programming, it is only very recently that Munos [43][44] proved that finite difference methods converge to the value function of the continuous problem, when the step size of the discretization goes to zero.

1.2.4 Pendulum Swing-Up

The pendulum-swing-up task [4][23] is a simple control problem that can be used to test this general algorithm. The system to be controlled consists of a simple pendulum actuated by a bounded torque (Figure 1.11). The goal is to reach the vertical upright position. Since the torque available is not sufficient to reach the goal position directly, the controller has to swing the pendulum back and forth to accumulate energy. It has then to decelerate it early enough so that it does not fall over. In order to reach this goal, the reward used is $\cos(\theta)$. Detailed specifications of the problem are given in Appendix B.

Figure 1.12 shows an accurate estimation of the value function obtained with a 1600×1600 discretization of the state space. The minimum value is at ($\theta = \pm\pi$, $\dot{\theta} = 0$), that is to say the steady balance position (when the pendulum is down). The maximum is at ($\theta = 0$, $\dot{\theta} = 0$), that is to say the

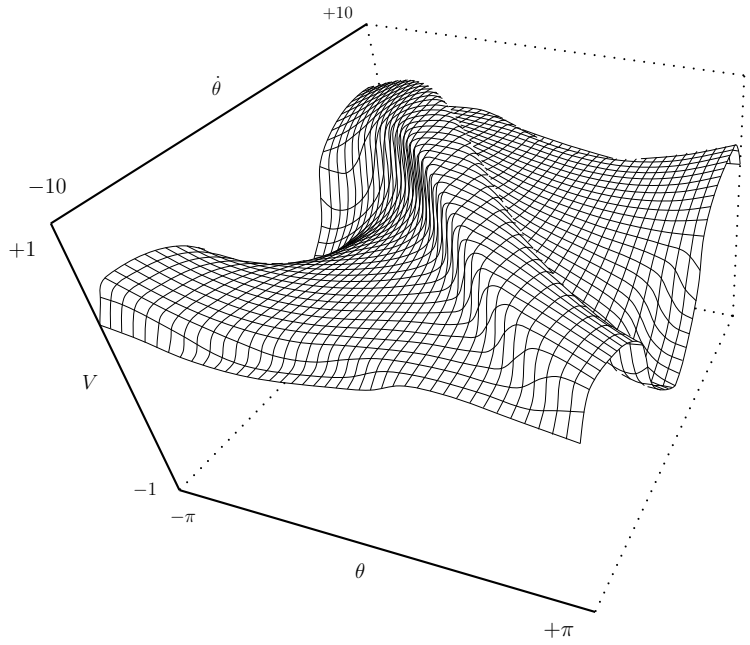


Figure 1.12: Value function obtained by value iteration on a 1600×1600 grid for the pendulum swing-up task.

upright position. A striking feature of this value function is a diagonal ridge from positive θ and negative $\dot{\theta}$ to negative θ and positive $\dot{\theta}$, limited by vertical cliffs. As shown on Figure 1.13, optimal trajectories follow this ridge toward the goal position.

1.2.5 The Curse of Dimensionality

Dynamic programming is very simple and applicable to a wide variety of problems, but it suffers from a major difficulty that Bellman called the *curse of dimensionality*: the cost of discretizing the state space is exponential with the state dimension, which makes value iteration computationally intractable when the dimension is high. For instance, if we suppose that each state variable is discretized with 100 samples, then a one-dimensional problem would have 100 states, which is very easy to handle. A 2-dimensional problem would have 10 000 states, which becomes a little hard to process. A problem with a 4-dimensional state space would have 100 million states, which reaches the limit of what modern microcomputers can handle.

One way to deal with this problem consists in using a discretization that is more clever than a uniform grid [46][51]. By using a coarser discretization in areas where little accuracy is needed (because the value function is almost

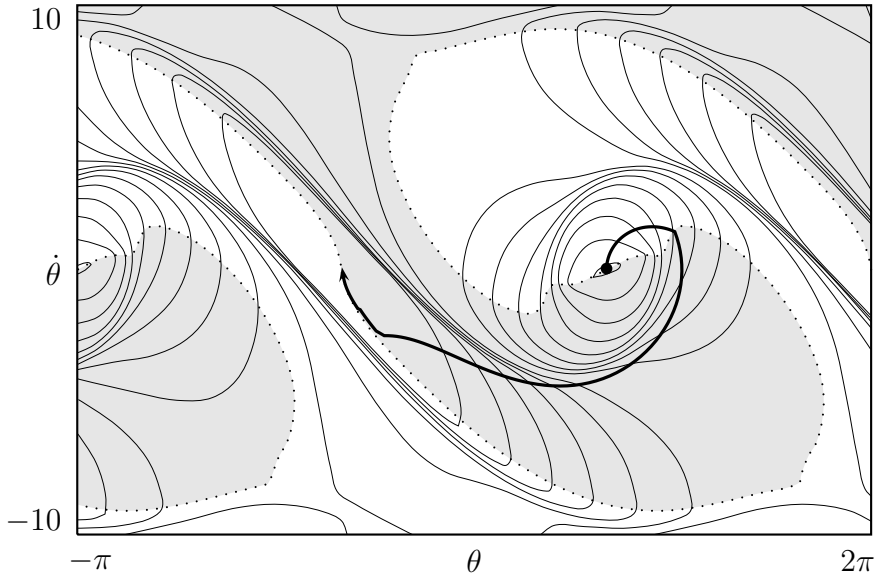


Figure 1.13: Trajectory obtained for the pendulum, starting from the downward position. Lines of constant estimated V^* are plotted every 0.1, from -0.4 to 0.9. Control is $-u_{max}$ in the gray area and $+u_{max}$ in the white area.

linear, for instance), it is possible to solve large problems efficiently. According to Munos, this kind of method can handle problems up to dimension 6.

Although Munos results pushed the complexity limit imposed by the curse of dimensionality, many problems are still out of reach of this kind of methods. For instance, a simple model of the dynamics of a human arm between shoulder and wrist has 7 degrees of freedom. This means that a model of its dynamics would have 14 state variables, which is far beyond what discretization-based methods can handle.

Nevertheless, this does not mean that ideas of dynamic programming cannot be applied to high-dimensional problems. A possible alternative approach consists in using the generalization capabilities of artificial neural networks in order to break the cost of discretization. This method will be presented in the next chapters.

Chapter 2

Artificial Neural Networks

The grid-based approximation of the value function that was used in the previous chapter is only a particular case of a *function approximator*. Grid-based approximation suffers from the curse of dimensionality, which is a major obstacle to its application to difficult motor control tasks. Some other function approximators, however, can help to solve this problem thanks to their ability to *generalize*. This chapter presents artificial neural networks, which are a particular kind of such approximators.

2.1 Function Approximators

2.1.1 Definition

A *parametric function approximator* (or *estimator*) can be formally defined as a set of functions indexed by a vector $\vec{w}$ of scalar values called *weights*. A typical example is the set of linear functions $f_{\vec{w}}$ defined by

$$f_{\vec{w}}(x) = w_1 x + w_0$$

where

$$\vec{w} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \in \mathbb{R}^2$$

is the vector of weights. Polynomials of any degree can be used too. Other architectures will be described in Section 2.3.

As its name says, a function approximator is used to approximate data. One of the main reason to do this is to get some form of *generalization*. A typical case of such a generalization is the use of linear regression to interpolate or extrapolate some experimental data (see Figure 2.1).

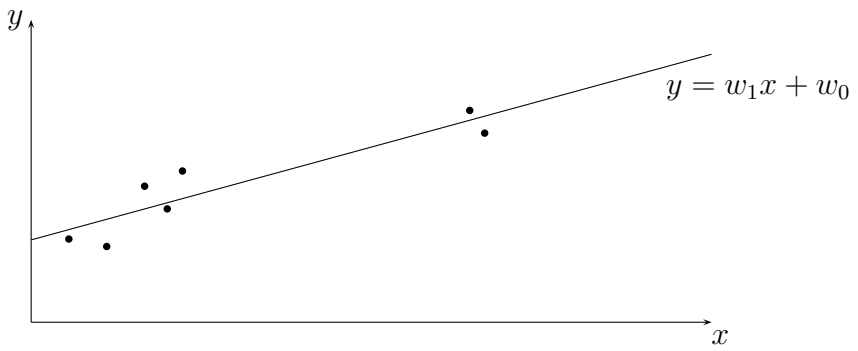


Figure 2.1: Linear regression can interpolate and extrapolate data. That is to say it can *generalize*. Generalization helps function approximators to break the curse of dimensionality.

2.1.2 Generalization

The problem of defining generalization accurately is very subtle. The simple example below can help to explore this question:

1 2 3 4 5 6 ? 8 9 10

This is a short sequence of numbers, one of which has been replaced by a question mark. What is the value of this number? It might as well be 2, 6 or 29. There is no way to know. It is very likely, however, that many people would answer "7" when asked this question. "7" seems to be the most obvious answer to give, if one has to be given.

Why "7"? This could be explained by Occam's razor[1]: "One should not increase, beyond what is necessary, the number of entities required to explain anything." Let us apply this principle to some function approximators:

- table of values: $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, $f(4) = 4$, $f(5) = 5$, $f(6) = 6$, $f(7) = 29$, $f(8) = 8$, $f(9) = 9$, $f(10) = 10$.
- linear regression: $f(i) = i$.

Occam's razor states that $f(i) = i$ should be preferred to the table of values because it is the simplest explanation of visible numbers.

So, finding the best generalization would consist in finding the simplest explanation of visible data. A big problem with this point of view on generalization is that the "simplicity" of a function approximator is not defined accurately. For instance, let us imagine an universe, the laws of which are based on the "1 2 3 4 5 6 29 8 9 10" sequence. An inhabitant of this universe

[1]This principle is attributed to William of Occam (or Okham), a medieval philosopher (1280?–1347?).

might find that 29 is the most natural guess for the missing number! Another (less weird) possibility would be that, independently of this sequence, other numbers had been presented to this person the day before:

1 2 3 4 5 6 29 8 9 10
1 2 3 4 5 6 29 8 9 10
1 2 3 4 5 6 29 8 9 10...

This means that deciding whether a generalization is good depends on *prior knowledge*. This prior knowledge may be any kind of information. It may be other data or simply an intuition about what sort of function approximator would be well suited for this specific problem.

Some theories have been developed to formalize this notion of generalization, and to build efficient algorithms. Their complexity is way beyond the scope of this chapter, but further developments of this discussion can be found in the machine-learning literature. In particular, Vapnik's theory of structural risk minimization is a major result of this field [75]. Many other important ideas, such as Bayesian techniques, are clearly explained in Bishop's book [16].

Without going into these theories, it is possible to estimate the generalization capabilities of a parametric function approximator intuitively: it should be as "simple" as possible, and yet be able to approximate as many "usual" functions as possible.

2.1.3 Learning

In order to approximate a given target function, it is necessary to find a good set of weights. The problem is that changing one weight is likely to alter the output of the function on the whole input space, so it is not as easy as using a grid-based approximation. One possible solution consists in minimizing an *error function* that measures how bad an approximation is.

Usually, there is a finite number of sample input/output pairs and the goal is to find a function that approximates them well. Let us call these samples (x_i, y_i) with $i \in \{1, \ldots, p\}$. In this situation, a quadratic error can be used:

$$E(\vec{w}) = \frac{1}{2} \sum_{i=1}^{p} \left(f_{\vec{w}}(x_i) - y_i \right)^2.$$

The process of finding weights that minimize the error function E is called *training* or *learning* by artificial intelligence researchers. It is also called *curve-fitting* or *regression* in the field of data analysis. In the particular case of linear functions (Figure 2.1), the linear regression method directly provides

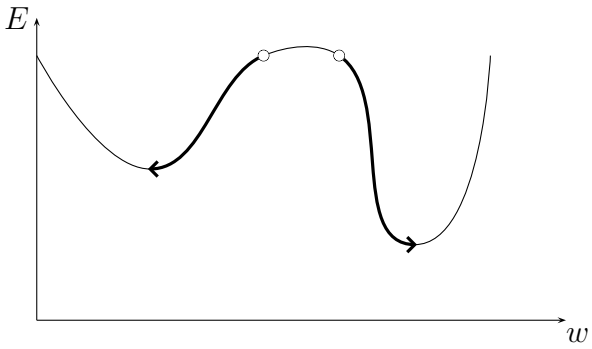


Figure 2.2: Gradient descent. Depending on the starting position, different local optima may be reached.

optimal weights in closed form. In the general case, more complex algorithms have to be used. Their principle will be explained in Section 2.2.

Training a function estimator on a set of (input, output) pairs is called *supervised* learning. This set is called the *training set*. Only this kind of learning will be presented in this chapter. Actually, many of the ideas related to supervised learning can be used by reinforcement learning algorithms. These will be discussed in the next chapters.

2.2 Gradient Descent

As explained in Section 2.1.3, training a function estimator often reduces to finding a value of $\vec{w}$ that minimizes a scalar error function $E(\vec{w})$. This is a classical optimization problem, and many techniques have been developed to solve it. The most common one is *gradient descent*. Gradient descent consists in considering that E is the altitude of a landscape on the weight space: to find a minimum, starting from a random point, walk downward until a minimum is reached (see Figure 2.2).

As this figure shows, gradient descent will not always converge to an absolute minimum of E, but only to a local minimum. In most usual cases, this local minimum is good enough, provided that a reasonable initial value of $\vec{w}$ has been chosen.

2.2.1 Steepest Descent

The most basic algorithm to perform gradient descent consists in setting a step size with a parameter η called the *learning rate*. Weights are iteratively added the value of

$$\delta\vec{w} = -\eta\frac{\partial E}{\partial\vec{w}}.$$

This is repeated until some termination criterion is met. This algorithm is called *steepest descent*[2].

2.2.2 Efficient Algorithms

Choosing the right value for the learning rate η is a difficult problem. If η is too small, then learning will be too slow. If η is too large, then learning may diverge. A good value of η can be found by trial and error, but it is a rather tedious and inefficient method. In order to address this problem, a very large variety of efficient learning techniques has been developed. This section presents the most important theoretical ideas underlying them.

One of the most fundamental ideas to accelerate learning consists in using second order information about the error function. As Figure 2.3 shows, for a quadratic error in one dimension, the best learning rate is the inverse of the second order derivative. This can help to design efficient learning techniques; if it is possible to evaluate this second order derivative, then it is possible to automatically find a good learning rate.

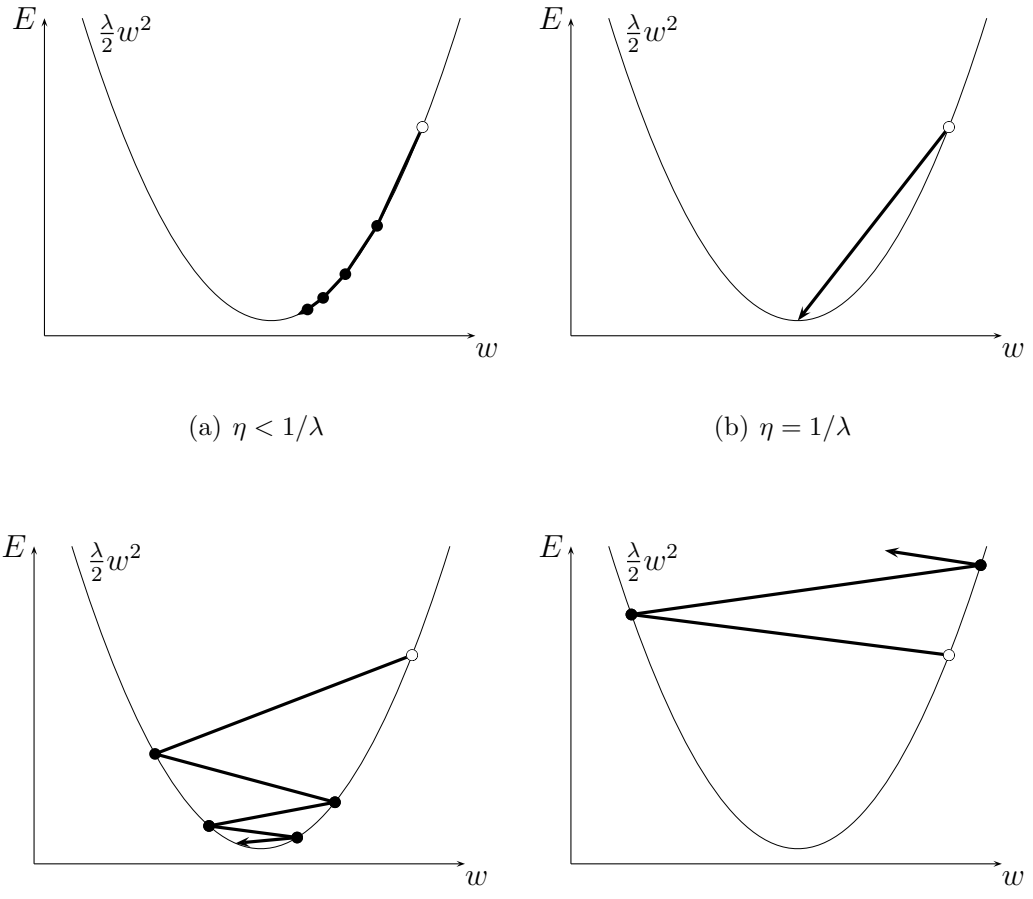
Unfortunately, the error function might not be quadratic at all. So, setting the learning coefficient to the reverse of the second order derivative only works near the optimum, in areas where the quadratic approximation is valid. But when, for instance, the second order derivative is negative, this does not work at all (that is the case at the starting positions in Figure 2.2.) Special care must be taken to handle these situations.

Some other problems arise when the dimension of the weight space is larger than one, which is always the case in practice. The second order derivative is not a single number anymore, but a matrix called the *Hessian* and defined by

$$H = \frac{\partial^2 E}{\partial \vec{w}^2} = \begin{pmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_n \partial w_1} & \frac{\partial^2 E}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_n^2} \end{pmatrix}.$$

As Figure 2.4 shows, it is possible to have different curvatures in different directions. It can create a lot of trouble if there is, say, a second derivative of 100 in one direction, and a second derivative of 1 in another. In this case,

[2]This algorithm is sometimes also called *standard backprop*, which is short for *backpropagation of error*. This vocabulary is very confusing. In this document, the "backpropagation" term will only refer to the algorithm used to compute the gradient of the error function in feedforward neural networks.



(a) $\eta < 1/\lambda$

(b) $\eta = 1/\lambda$

(c) $1/\lambda < \eta < 2/\lambda$

(d) $\eta > 2/\lambda$

Figure 2.3: Effect of the learning rate η on gradient descent. λ is the second derivative of the error function E. At each time step, the weight w is added $\delta w = -\eta \frac{\partial E}{\partial w} = -\eta \lambda w$.

the learning coefficient must be less than 2/100 in order to avoid divergence. This means that convergence will be very slow in the direction where the second derivative is 1. This problem is called *ill-conditioning*.

So, efficient algorithms often try to transform the weight space in order to have uniform curvatures in all direction. This has to be done carefully so that cases where the curvature is negative work as well. Some of the most successful algorithms are conjugate gradient [61], scaled conjugate gradient [39], Levenberg Marquardt, RPROP [55] and QuickProp [26]. A collection of techniques for efficient training of function approximators is available in a book chapter by Le Cun *et al.* [37].

2.2.3 Batch *vs.* Incremental Learning

When doing supervised learning, the error function is very often defined as a sum of error terms over a finite number of training samples that consist of (input, output) pairs, as explained in Section 2.1.3. $(\vec{x}_i, \vec{y}_i)_{1 \leq i \leq p}$ are given and the error function is

$$E = \sum_{i=1}^{p} E_i,$$

with

$$E_i = \frac{1}{2}\left(f_{\vec{w}}(\vec{x}_i) - \vec{y}_i\right)^2.$$

Performing steepest descent on E is called *batch* learning, because the gradient of the error has to be evaluated on the full training set before weights are modified. Another method to modify weights in order to minimize E is *incremental* learning. It consists in performing gradient descent steps on E_i's instead of E (see algorithms 2.1 and 2.2.) Incremental learning is often also called *online* learning or *stochastic* learning. See the *Neural Network FAQ* [58] for a more detailed discussion about these vocabulary issues.

Algorithm 2.1 Batch Learning

$\vec{w} \leftarrow$ some random initial value
repeat
  $\vec{g} \leftarrow \vec{0}$
  for $i = 1$ to p **do**
    $\vec{g} \leftarrow \vec{g} + \partial E_i / \partial \vec{w}$
  end for
  $\vec{w} \leftarrow \vec{w} - \eta \vec{g}$
until termination criterion is met

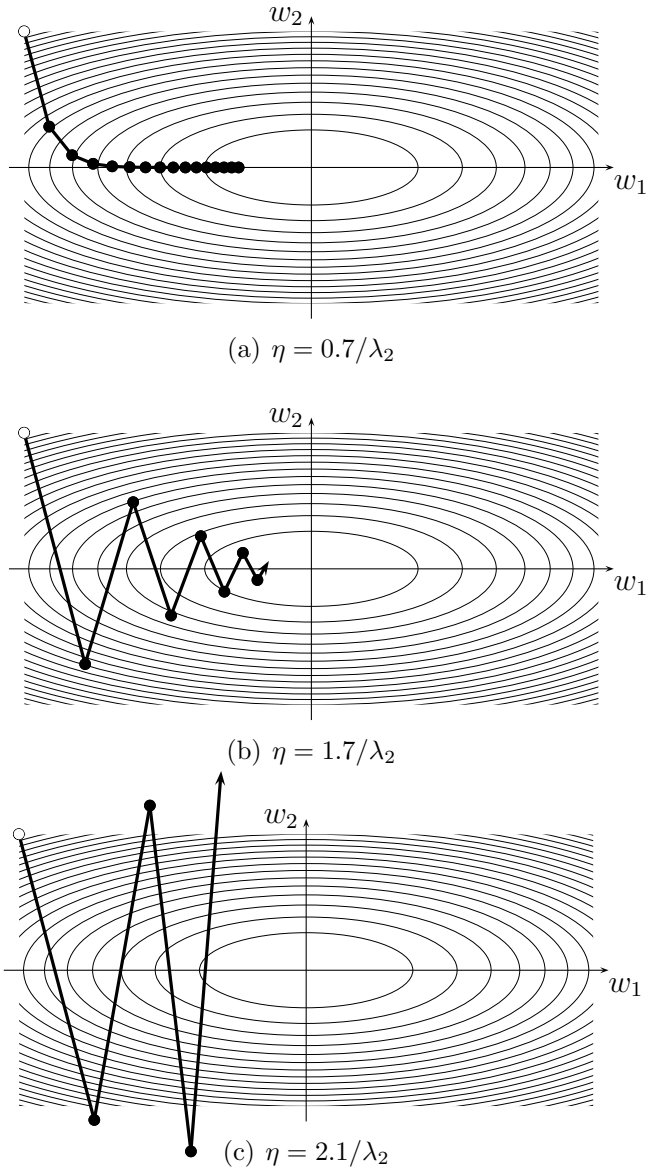


Figure 2.4: Ill-Conditioning. Ellipses are lines of constant error E. The Hessian of E is $H = \left(\begin{smallmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{smallmatrix}\right)$ with $\lambda_1 = 1$, and $\lambda_2 = 8$. The steepest descent algorithm is applied. No learning rate η gives fast convergence.

Algorithm 2.2 Incremental Learning

$\vec{w} \leftarrow$ some random initial value
repeat
  $i \leftarrow$ random value between 1 and p
  $\vec{w} \leftarrow \vec{w} - \eta \partial E_i / \partial \vec{w}$
until termination criterion is met

Which of these techniques is the best? This is a difficult question, the answer of which depends on the specific problem to be solved. Here are some of the points to consider: (This discussion is very deeply inspired by a book chapter by Le Cun *et al.* [37].)

Advantages of Incremental Learning

1. Incremental learning is usually faster than batch learning, especially when the training set is redundant. In the case when the training set has input/output patterns that are similar, batch learning wastes time computing and adding similar gradients before performing one weight update.

2. Incremental learning often results in better solutions. The reasons is that the randomness of incremental learning creates noise in the weight updates. This noise helps weights to jump out of bad local optima [48].

3. Incremental learning can track changes. A typical example is when learning a model of the dynamics of a mechanical system. As this system gets older, its properties might slowly evolve (due to wear of some parts, for instance). Incremental learning can track this kind of drift.

Advantages of Batch Learning

1. Conditions of convergence are well understood. Noise in incremental learning causes weights to constantly fluctuate around a local optimum, and they never converge to a constant stable value. This does not happen in batch learning, which makes it easier to analyze.

2. Many acceleration techniques only operate in batch learning. In particular, algorithms listed in the previous subsection can be applied to batch learning only (Conjugate gradient, RPROP, QuickProp, ...).

3. Theoretical analysis of the weight dynamics and convergence rates are simpler. This is also related to the lack of noise in batch learning.

2.3 Some Approximation Schemes

2.3.1 Linear Function Approximators

The general form of linear function approximators is

$$V_{\vec{w}}(\vec{x}) = \sum_i w_i \phi_i(\vec{x}).$$

This kind of function approximator has many nice characteristics that have made it particularly successful in reinforcement learning.

Unlike some more complex function approximation schemes, like feedforward neural networks, it is rather easy to train a linear function approximator. In particular, there is no poor local optimum and the Hessian of the error function is diagonal (and easy to evaluate).

Another nice quality of such a function approximator is *locality* [77]. By choosing ϕ_i's such that $\phi_i(\vec{x})$ has a non-zero value in a small area of the input space only, it is possible to make sure that a change in w_i will have significant consequences in a small area only. This is often considered a good property for reinforcement learning. The reason is that reinforcement learning is often performed incrementally, and the value-function approximator should not forget what it has learnt in other areas of the state space when trained on a new single input $\vec{x}$.

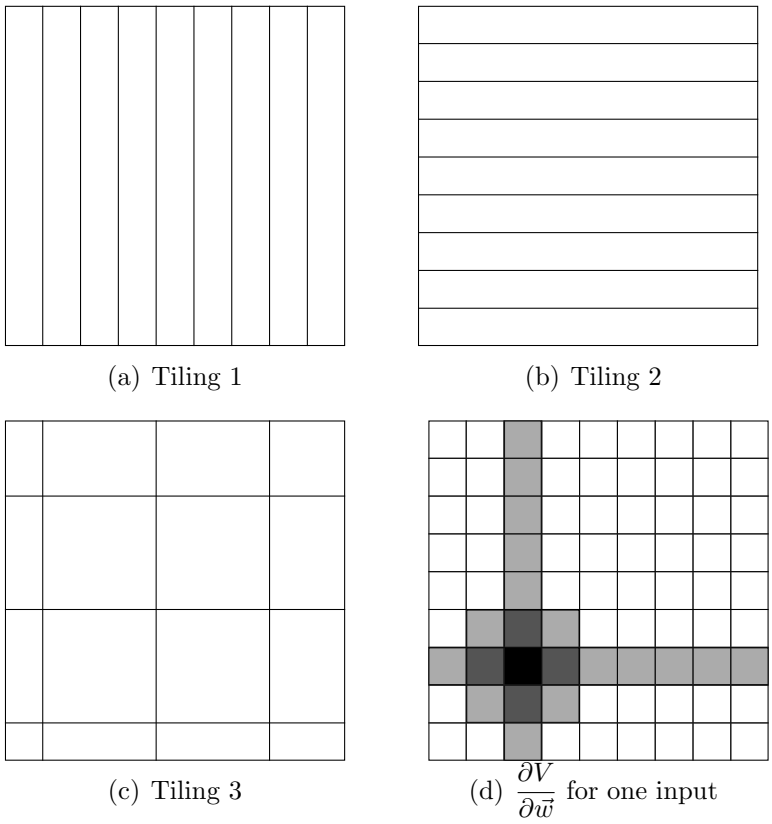
Some of the most usual linear function approximators are described in the sections below. Many variations on these ideas exist.

Grid-Based Approximation

The grid-based approximation of the value function used in the previous chapter is a particular case of a linear function approximator. As explained previously, this kind of function approximation suffers from the curse of dimensionality, because it takes a huge number of weights to sample a high-dimensional state space with enough accuracy (even with clever discretization methods). In terms of neural-network learning, this means that this kind of function approximation scheme has very poor generalization capabilities.

Tile Coding (or CMAC[1])

Instead of using one single grid to approximate the value function, tile coding consists in adding multiple overlapping tilings (*cf.* Figure 2.5). This is a way



(a) Tiling 1 (b) Tiling 2

(c) Tiling 3 (d) $\frac{\partial V}{\partial \vec{w}}$ for one input

Figure 2.5: Tile Coding

to add generalization to tables of values. As the figure shows, changing the value function at one point will alter it at other points of the input space. This can help a little to alleviate the curse of dimensionality, while keeping good locality. Using tile coding for high-dimensional problems is not that easy though; it requires a careful choice of tilings.

Normalized Gaussian Networks

A major problem with the application of tile coding to continuous reinforcement learning is that such a function approximator is not continuous. That is to say a smooth estimate of the value function is needed, but tile coding produces "steps" at the border of tiles. In order to solve this problem, it is necessary to make gradual transitions between tiles. This is what is done by normalized Gaussian networks.

In normalized Gaussian networks [40], the rough rectangular ϕ_i's that

were used in previous approximators are replaced by smooth bumps:

$$\phi_i(\vec{x}) = \frac{G_i(\vec{x})}{\sum_j G_j(\vec{x})},$$
$$G_i(\vec{x}) = e^{-(\vec{x}-\vec{c}_i)^t M_i (\vec{x}-\vec{c}_i)}.$$

$\vec{c}_i$ is the center of the Gaussian number i; M_i defines how much this Gaussian is "spread" in all directions (it is the inverse of the covariance matrix).

It is important to choose the variance of each G_i carefully. If it is too high, then G_i will be very wide, and locality will not be good. If it is too small, then G_i will not be smooth enough (see Figure 2.6).

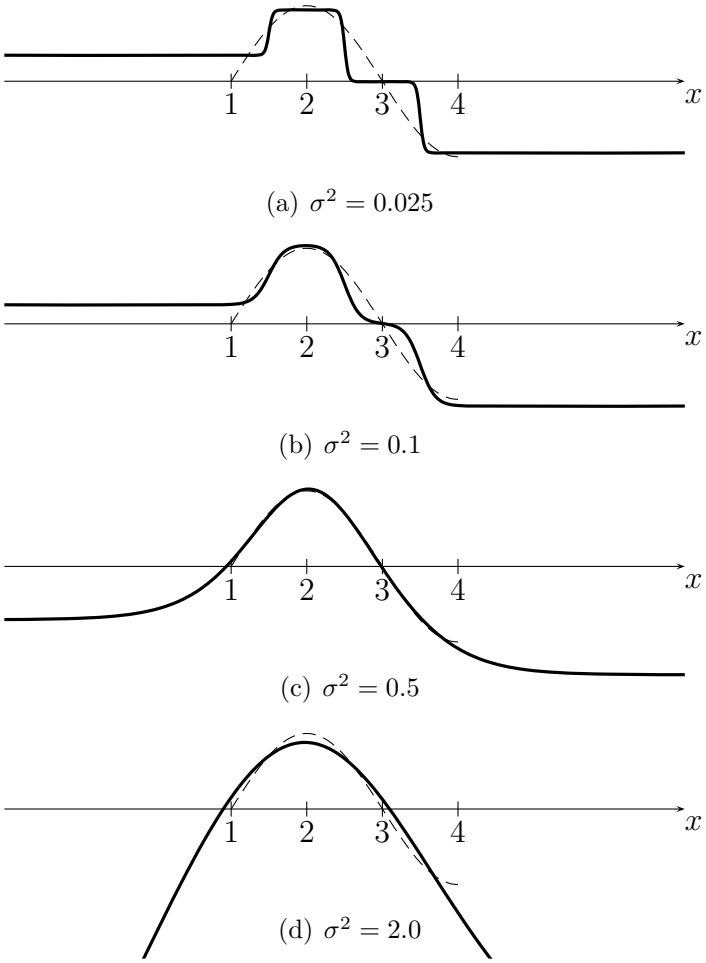
The behavior of such a normalized Gaussian network is actually closer to a grid-based function approximator than to tile coding. In order to avoid the curse of dimensionality, it is still possible to overlap as many sums as needed, with a different distribution of $\vec{c}_i$'s in each sum, similarly to what is shown on Figure 2.5.

Various techniques allow to make an efficient implementation of normalized Gaussian networks. In particular, if the $\vec{c}_i$'s are allocated on a regular grid, and the M_i matrices are diagonal and identical, then the G_i's can be computed efficiently as the outer product of the activation vectors for individual input variables. Another technique that can produce significant speedups consists in considering only some of the closest basis functions in the sum; $\vec{c}_i$'s that are too far away from $\vec{x}$ can be neglected. Normalized Gaussian Networks are still much more costly to use than tile coding, though.

2.3.2 Feedforward Neural Networks

Feedforward neural networks consist of a graph of nodes, called neurons, connected by weighted links. These nodes and links form a directed acyclic graph, hence the name "feedforward". Neurons receive input values and produce output values. The mapping from input to output depends on the link weights (see Figure 2.7), so a feedforward neural network is a parametric function approximator. The gradient of the error with respect to weights can be computed thanks to the backpropagation algorithm. More technical details about this algorithm can be found in Appendix A, along with the formal definition of a neural network.

In reinforcement learning problems, linear function approximators are often preferred to feedforward neural networks. The reason is that feedforward neural networks are much more tricky to handle than linear function approximators: locality is poor, learning can be trapped in a local optimum, and ill-conditioning can hurt a lot. Besides, as we shall see in the next chapter,



(a) $\sigma^2 = 0.025$

(b) $\sigma^2 = 0.1$

(c) $\sigma^2 = 0.5$

(d) $\sigma^2 = 2.0$

Figure 2.6: Effect of σ^2 on a normalized Gaussian network. Gaussians are centered at 1, 2, 3, and 4. Each network is trained to fit the dashed function: $y = \sin\big((x-1)\pi/2\big)$. Vertical scale (arbitrary units) represents the value function.

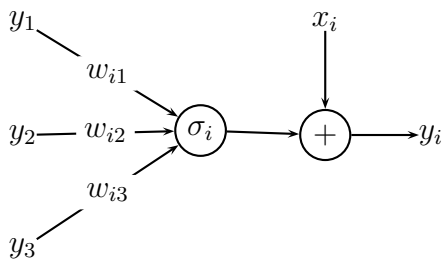


Figure 2.7: Neuron in a feedforward network: $y_i = x_i + \sigma_i\left(\sum_{j<i} w_{ij} y_j\right)$

reinforcement learning with non-linear function approximators has very few guarantees of convergence.

It is worth noting, however, that Tesauro used a feedforward neural network with a lot of success in his backgammon player [70]. This backgammon player is often regarded as one of the most remarkable achievements of reinforcement learning research. A major strength of feedforward neural networks is their ability to handle high-dimensional input. Barron [9] proved that they do not suffer from the curse of dimensionality, whereas linear function approximators do.

So, although their complexity can make them difficult to harness, feedforward neural networks can be very efficient. They have excellent generalization capabilities, which can help to solve difficult reinforcement-learning tasks, especially when the dimension of the state space is high.

Chapter 3

Continuous Neuro-Dynamic Programming

The previous two chapters introduced dynamic programming and neural networks. They provided some intuition about the reasons why generalization capabilities of neural networks could help to break the curse of dimensionality that is hanging over discretization-based approaches to dynamic programming. This chapter describes the theory of algorithms that combine these two techniques, which Bertsekas and Tsitsiklis called *neuro-dynamic programming* [15]. Emphasis is placed on their application to continuous problems.

3.1 Value Iteration

Value iteration algorithms presented in Chapter 1 consist in solving a fixed point equation of the kind $\vec{V} = g(\vec{V})$ by iteratively applying g to an initial value. When a function approximator $V_{\vec{w}}$ is used instead of a table of values, these algorithms cannot be applied because $\vec{V}$ is estimated indirectly as a function of parameters $\vec{w}$. The key issue is that the assignment $\vec{V}_{i+1} \leftarrow g(\vec{V}_i)$ has to be replaced by some form of weight update.

3.1.1 Value-Gradient Algorithms

The most straightforward method to perform this weight update consists in replacing an assignment such as

$$V(x) \leftarrow y$$

by one step of gradient descent on the error

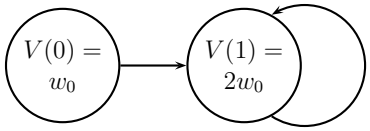
$$E(\vec{w}) = \frac{1}{2}\big(V_{\vec{w}}(x) - y\big)^2.$$

This leads to a weight update

$$\vec{w} \leftarrow \vec{w} + \eta\big(y - V_{\vec{w}}(x)\big)\frac{\partial V_{\vec{w}}(x)}{\partial \vec{w}},$$

where η is the learning rate.

Although this might look like a natural way to generalize value iteration to function approximators, this method does not work well. Baird showed that this approach to value iteration may diverge for a very small decision process [7]. Tsitsiklis and Van Roy gave another very simple two-state counterexample [73]:



In this process, each state has only one possible action and all transitions give a reward of zero. Thus, when a discount factor $\gamma < 1$ is used, the value function is equal to zero everywhere. A linear function approximator is used to approximate this value function. The value-gradient approach to value iteration leads to a weight update of

$$w_0 \leftarrow w_0 + \eta\left(\big(\gamma V(1) - V(0)\big)\frac{\partial V(0)}{\partial w_0} + \big(\gamma V(1) - V(1)\big)\frac{\partial V(1)}{\partial w_0}\right),$$

that is to say

$$w_0 \leftarrow w_0 + \eta\big((2\gamma - 1)w_0 + (2\gamma - 2)2w_0\big),$$

which can be simplified into

$$w_0 \leftarrow w_0\big(1 + \eta(6\gamma - 5)\big).$$

So, when $\gamma > \frac{5}{6}$, this algorithm diverges.

A reason why this method does not work is that the contraction property that proved convergence of lookup-table approaches can not be established anymore. Gordon [29] studied characteristics that a function approximator should have so that the process of updating weights still is a contraction. Unfortunately, the conditions imposed to keep this property are extremely restrictive, and dismiss most of the function approximators with good generalization abilities, such as tile coding or feedforward neural networks.

3.1.2 Residual-Gradient Algorithms

Divergence of value-gradient algorithms is related to *interference*: when weights are updated to change the value of one state, it is very likely that the value of other states will change as well. As a consequence, the target value y in the $V(x) \leftarrow y$ assignment might change as $\vec{w}$ is modified, which may actually let y move away from $V(x)$ and cause divergence of the algorithm.

In order to solve this problem, Baird proposed *residual-gradient algorithms* [7]. They consist in using an error function that takes into account the dependency of y on $\vec{w}$:

$$E(\vec{w}) = \frac{1}{2}\big(\vec{V}_{\vec{w}} - g(\vec{V}_{\vec{w}})\big)^2$$

Since E is a function of $\vec{w}$ that does not change as learning progresses, there is no moving-target problem anymore, which lets the gradient-descent algorithm converge[1].

One major limit to this approach, however, is that there is no guarantee that the estimated value function obtained is close to the solution of the dynamic programming problem. In fact, it is possible to get a value of $\vec{w}$ so that $\vec{V}_{\vec{w}}$ is extremely close to $g(\vec{V}_{\vec{w}})$, but very far from the single solution of the $\vec{V} = g(\vec{V})$ equation. This phenomenon is particularly striking in the continuous case, which is presented in the next section.

3.1.3 Continuous Residual-Gradient Algorithms

The semi-continuous Bellman equation derived in Chapter 1 is based on the approximation

$$V^\pi(\vec{x}) \approx r\big(\vec{x}, \pi(\vec{x})\big)\delta t + e^{-s_\gamma \delta t} V^\pi(\vec{x} + \delta\vec{x}).$$

When using a discretization of the state space, a value of δt is used so that it lets the system move from one discrete state to nearby discrete states. In the general case of function approximation, however, there is no such thing as "nearby states". δt could be chosen to be any arbitrarily small time interval. This leads to completely continuous formulations of dynamic programming algorithms.

[1] Note that, in general, E is not differentiable because of the max operator in g. Since E is continuous and differentiable almost everywhere, gradient descent should work, anyway.

The Hamilton-Jacobi-Bellman Equation

Let us suppose that δt is replaced by an infinitely small step dt. The policy-evaluation equation becomes

$$\begin{aligned} V^\pi(\vec{x}) &\approx r\big(\vec{x}, \pi(\vec{x})\big)dt + e^{-s_\gamma dt} V^\pi(\vec{x} + d\vec{x}) \\ &\approx r\big(\vec{x}, \pi(\vec{x})\big)dt + (1 - s_\gamma dt) V^\pi(\vec{x} + d\vec{x}). \end{aligned}$$

By subtracting $V^\pi(\vec{x})$ to each term and dividing by dt we get:

$$0 = r\big(\vec{x}, \pi(\vec{x})\big) - s_\gamma V^\pi(\vec{x}) + \dot{V}^\pi(\vec{x}). \tag{3.1.1}$$

If we note that

$$\dot{V}^\pi(\vec{x}) = \frac{\partial V^\pi}{\partial \vec{x}} \cdot \frac{d\vec{x}}{dt} = \frac{\partial V^\pi}{\partial \vec{x}} \cdot f(\vec{x}, \vec{u}),$$

then (3.1.1) becomes

$$0 = r\big(\vec{x}, \pi(\vec{x})\big) - s_\gamma V^\pi(\vec{x}) + \frac{\partial V^\pi}{\partial \vec{x}} \cdot f\big(\vec{x}, \pi(\vec{x})\big). \tag{3.1.2}$$

A similar equation can be obtained for the optimal value function:

$$0 = \max_{\vec{u} \in U} \left(r(\vec{x}, \vec{u}) - s_\gamma V^*(\vec{x}) + \frac{\partial V^*}{\partial \vec{x}} \cdot f(\vec{x}, \vec{u}) \right). \tag{3.1.3}$$

(3.1.3) is the *Hamilton-Jacobi-Bellman* equation. It is the continuous equivalent to the discrete Bellman equation. Besides, for any value function V, the *Hamiltonian* $\mathcal{H}$ is defined as

$$\mathcal{H} = \max_{\vec{u} \in U} \left(r(\vec{x}, \vec{u}) - s_\gamma V(\vec{x}) + \frac{\partial V}{\partial \vec{x}} \cdot f(\vec{x}, \vec{u}) \right),$$

which is analogous to the discrete Bellman residual.

Continuous Value Iteration

So, continuous residual-gradient algorithms would consist in performing gradient decent[2] on $E = \frac{1}{2} \int_{x \in S} \mathcal{H}^2 dx$. Munos, Baird and Moore [45] studied this algorithm and showed that, although it does converge, it does not converge to the right value function. The one-dimensional robot problem presented

[2]Performing gradient descent on this kind of error function is a little bit more complex than usual supervised learning, because the error depends on the gradient of the value function with respect to its input. This problem can be solved by the differential backpropagation algorithm presented in Appendix A.

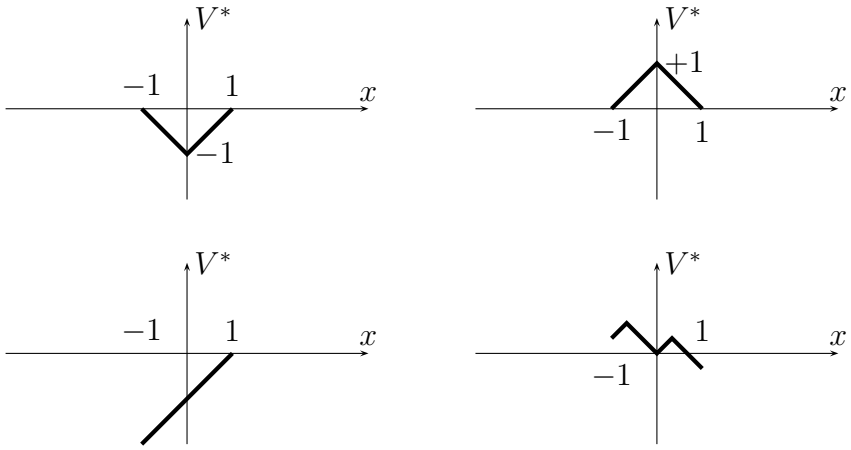


Figure 3.1: Some of the many solutions to the Hamilton-Jacobi-Bellman equation for the robot-control problem $\left|\frac{\partial V^*}{\partial x}\right| = 1$. The right solution is at the top left.

in Chapter 1 can illustrate this phenomenon. The Hamilton-Jacobi-Bellman equation (3.1.3) for this problem (with $s_\gamma = 0$ and $r(\vec{x}, \vec{u}) = -1$) is

$$0 = \max_{u \in [-1,1]} \left(-1 + \frac{\partial V^*}{\partial x} u \right).$$

Finding the value of u that maximizes this is easy. It simply consists in taking $u = +1$ when $\partial V^*/\partial x$ is positive and $u = -1$ otherwise. Thus, we get

$$\left| \frac{\partial V^*}{\partial x} \right| = 1.$$

The value function may be discontinuous or not differentiable. If we consider functions that are differentiable almost everywhere, then this differential equation clearly has an infinite number of solutions (see Figure 3.1).

Munos *et al* [45] used the theory of viscosity solutions to explain this: out of the infinity of solutions to the Hamilton-Jacobi-Bellman equation, the viscosity solution is the only value function that solves the optimal control problem. Gradient descent with a function approximator does not guarantee convergence to this solution, so the result of this algorithm may be completely wrong.

Doya [24] gives another interpretation in terms of symmetry in time. A key idea of discrete value iteration is that the value of the current state is updated by trusting values of future states. By taking the time step δt to

zero, this asymmetry in time disappears and the learning algorithm may converge to a wrong value function.

So, the conclusion of this whole section is that value iteration with function approximators usually does not work. In order to find an algorithm able to estimate the right value function, it is necessary to enforce some form of asymmetry in time and get away from the self-reference problems of fixed point equations. Temporal difference methods, which were developed in the theory of reinforcement learning, can help to overcome these difficulties.

3.2 Temporal Difference Methods

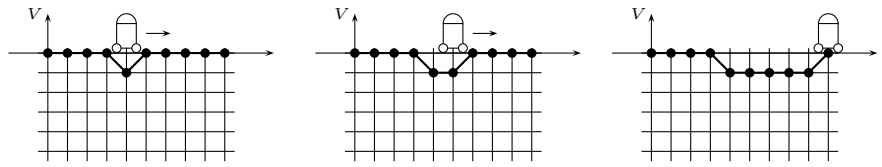
The easiest way to get rid of self-reference consists in calculating the value of one state from the outcome of a full trajectory, instead of relying on estimates of nearby states. This idea guided Boyan and Moore [19] to design the grow-support algorithm. This method uses complete "rollouts" to update the value function, which provides robust and stable convergence. Sutton [67] followed up Boyan and Moore's results with experiments showing that the more general TD(λ) algorithm [66] also produces good convergence and is faster than Boyan and Moore's method. TD stands for "temporal difference".

3.2.1 Discrete TD(λ)

A key idea of this form of learning algorithm is *online* training. Value iteration consists in updating the value function by full sweeps on the whole state space. An online algorithm, on the other hand, proceeds by following actual successive trajectories in the state space. These trajectories are called *trials* or *episodes*.

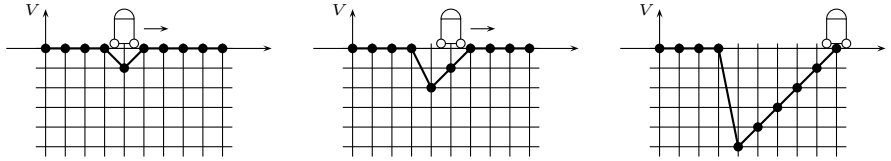
Figure 3.2 illustrates what happens when Bellman's equation is applied along a trajectory. The value function is initialized to be equal to zero everywhere. A random starting position is chosen for the first trial, and one iteration of dynamic programming is applied. Then, an optimal action is chosen. In this particular situation, there are two optimal actions: $+1$ and -1. One of them ($+1$) is chosen arbitrarily. Then, the robot moves according to this action and the same process is repeated until it reaches the boundary (Figure 3.2(c)).

By applying this algorithm, trajectory information was not used at all. It is possible to take advantage of it with the following idea: when a change is made at one state of the trajectory, apply this change to the previous states as well. This is justified by the fact that the evaluation of the previous state was based on the evaluation of the current state. If the latter changes, then



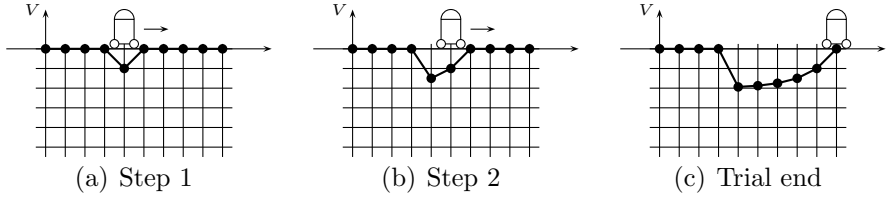
(a) Step 1 (b) Step 2 (c) Trial end

Figure 3.2: online application of value iteration (TD(0))



(a) Step 1 (b) Step 2 (c) Trial end

Figure 3.3: Monte-Carlo method (TD(1))



(a) Step 1 (b) Step 2 (c) Trial end

Figure 3.4: TD(λ) algorithm with $\lambda = \frac{1}{2}$

the previous evaluation should change accordingly. This method is called the Monte-Carlo algorithm (Figure 3.3).

A problem with the Monte-Carlo algorithm is that there is a probability that a change in the current value function does not affect the previous value as much. This is handled by the TD(λ) algorithm. Let us suppose that the correction to the current value is δV. TD(λ) consists in supposing that the expected change for the previous state is equal to $\lambda\delta V$ ($\lambda \in [0, 1]$). This change is backed up iteratively to all previous states. So the state two time steps before gets $\lambda^2\delta V$, the one before gets $\lambda^3\delta V$, *etc.* The coefficients 1, λ, λ^2, ... are the *eligibility traces*. TD(λ) is a generalization of online value iteration (TD(0)) and Monte-Carlo algorithm (TD(1)). TD(λ) has been reported by Sutton and others to perform significantly better than TD(0) or TD(1) if the value of λ is well chosen.

Algorithm 3.1 TD(λ)

$\vec{V} \leftarrow$ an arbitrary initial value
$\vec{e} \leftarrow \vec{0}$
for each episode **do**
  $x \leftarrow$ random initial state
  while not end of episode **do**
    $x' \leftarrow \nu\big(x, \pi(x)\big)$
    $\delta \leftarrow r\big(x, \pi(x)\big) + \gamma V(x') - V(x)$
    $e(x) \leftarrow e(x) + 1$
    $\vec{V} \leftarrow \vec{V} + \delta\vec{e}$
    $\vec{e} \leftarrow \lambda\gamma\vec{e}$
    $x \leftarrow x'$
  end while
end for

Algorithm 3.1 shows the details of TD(λ). π may either be a constant policy, in which case the algorithm evaluates its value function V^π, or a greedy policy with respect to the current value, in which case the algorithm estimates the optimal value function V^*. In the latter case, the algorithm is called generalized policy iteration (by Sutton and Barto [68]) or optimistic policy iteration (by Bertsekas and Tsitsiklis [15]). TD(λ) policy evaluation has been proved to converge [21, 22, 32]. Optimistic policy iteration has been proved to converge, but only with a special variation of TD(λ) that is different from Algorithm 3.1 [72].

3.2.2 TD(λ) with Function Approximators

In order to use function approximators, value updates can be replaced by weight updates in the direction of the value gradient. This is similar to what has been presented in Section 3.1.1. Algorithm 3.2 shows the details of this algorithm. Notice that, if a table-lookup approximation is used with $\eta = 1$, it is identical to Algorithm 3.1

Convergence properties of this algorithm are not very well known. The strongest theoretical result, obtained by Tsitsiklis and Van Roy [74], proves that discrete policy evaluation with linear function approximators converges when learning is performed along trajectories with TD(λ). They also proved that the error on the value function is bounded by

$$E \leq \frac{1 - \lambda\gamma}{1 - \gamma} E^*,$$

where E^* is the optimal quadratic error that could be obtained with the same function approximator. This indicates that, the more λ is close to 1, the more accurate the approximation.

Convergence of algorithms that compute the optimal value function has not been established. Tsitsiklis and Van Roy also gave an example where policy evaluation with a non-linear function approximator diverges. Nevertheless, although it has little theoretical guarantees, this technique often works well in practice.

Algorithm 3.2 Discrete-time TD(λ) with function approximation

$\vec{w} \leftarrow$ an arbitrary initial value
$\vec{e} \leftarrow \vec{0}$ {dimension of $\vec{e}$ = dimension of $\vec{w}$}
for each episode **do**
  $x \leftarrow$ random initial state
  while not end of episode **do**
    $x' \leftarrow \nu\big(x, \pi(x)\big)$
    $\delta \leftarrow r\big(x, \pi(x)\big) + \gamma V_{\vec{w}}(x') - V_{\vec{w}}(x)$
    $\vec{e} \leftarrow \lambda\gamma\vec{e} + \partial V_{\vec{w}}(x)/\partial\vec{w}$
    $\vec{w} \leftarrow \vec{w} + \eta\delta\vec{e}$
    $x \leftarrow x'$
  end while
end for

3.2.3 Continuous TD(λ)

Although the traditional theoretical framework for reinforcement learning is discrete [68], the special characteristics of problems with continuous state and action spaces have been studied in a number of research works [6, 8, 28, 57]. Doya [23, 24] first published a completely continuous formulation of TD(λ). Similarly to the continuous residual-gradient method that was presented previously, it uses the Hamilton-Jacobi-Bellman equation to get rid of the discretization of time and space.

Let us suppose that at time t_0 we get a Hamiltonian $\mathcal{H}(t_0)$. The principle of the TD(λ) algorithm consists in backing-up the measured $\mathcal{H}(t_0)$ error on past estimates of the value function on the current trajectory. Instead of the discrete exponential decay 1, $\lambda\gamma$, $(\lambda\gamma)^2$, ..., the correction is weighted by a smooth exponential. More precisely, the correction corresponds to a peak of reward $\mathcal{H}(t_0)$ during an infinitely short period of time dt_0, with a shortness $s_\gamma + s_\lambda$. This way, it is possible to keep the asymmetry in time although the time step is infinitely small. Learning is performed by moving the value function toward $\hat{V}$, defined by

$$\forall t < t_0 \quad \hat{V}(t) = V_{\vec{w}(t_0)} + \mathcal{H}(t_0) dt_0 e^{-(s_\gamma + s_\lambda)(t_0 - t)}.$$

A quadratic error can be defined as

$$dE = \frac{1}{2} \int_{-\infty}^{t_0} \left(V_{\vec{w}(t_0)}(t) - \hat{V}(t) \right)^2 dt,$$

the gradient of which is equal to

$$\begin{aligned} \frac{\partial dE}{\partial \vec{w}} &= - \int_{-\infty}^{t_0} \mathcal{H}(t_0) dt_0 e^{-(s_\gamma + s_\lambda)(t_0 - t)} \frac{\partial V_{\vec{w}(t_0)}\left(\vec{x}(t)\right)}{\partial \vec{w}} dt \\ &= -\mathcal{H}(t_0) dt_0 e^{-(s_\gamma + s_\lambda) t_0} \int_{-\infty}^{t_0} e^{(s_\gamma + s_\lambda) t} \frac{\partial V_{\vec{w}(t_0)}\left(\vec{x}(t)\right)}{\partial \vec{w}} dt \\ &= -\mathcal{H}(t_0) dt_0 \vec{e}(t_0), \end{aligned}$$

with

$$\vec{e}(t_0) = e^{-(s_\gamma + s_\lambda) t_0} \int_{-\infty}^{t_0} e^{(s_\gamma + s_\lambda) t} \frac{\partial V_{\vec{w}(t_0)}\left(\vec{x}(t)\right)}{\partial \vec{w}} dt.$$

$\vec{e}(t_0)$ is the eligibility trace for weights. A good numerical approximation can be computed efficiently if we assume that

$$\frac{\partial V_{\vec{w}(t_0)}\left(\vec{x}(t)\right)}{\partial \vec{w}} \approx \frac{\partial V_{\vec{w}(t)}\left(\vec{x}(t)\right)}{\partial \vec{w}}.$$

If $V_{\vec{w}}$ is linear with respect to $\vec{w}$, then this approximation is an equality. If it is non-linear, then it can be justified by the fact that weights usually do not change much during a single trial. Under this assumption, $\vec{e}$ is the solution of the ordinary differential equation

$$\dot{\vec{e}} = -(s_\gamma + s_\lambda)\vec{e} + \frac{\partial V_{\vec{w}}}{\partial \vec{w}}.$$

Using this result about the gradient of error, a gradient descent algorithm can be applied using a change in weights equal to

$$d\vec{w} = -\eta \frac{\partial dE}{\partial \vec{w}} = \eta \mathcal{H} \vec{e} dt.$$

Dividing by dt gives

$$\dot{\vec{w}} = \eta \mathcal{H} \vec{e}.$$

To summarize, the continuous TD(λ) algorithm consists in integrating the following ordinary differential equation:

$$\begin{cases} \dot{\vec{w}} = \eta \mathcal{H} \vec{e} \\ \dot{\vec{e}} = -(s_\gamma + s_\lambda)\vec{e} + \dfrac{\partial V_{\vec{w}}(\vec{x})}{\partial \vec{w}} \\ \dot{\vec{x}} = f\big(\vec{x}, \pi(\vec{x})\big) \end{cases} \tag{3.2.1}$$

with

$$\mathcal{H} = r\big(\vec{x}, \pi(\vec{x})\big) - s_\gamma V_{\vec{w}}(\vec{x}) + \frac{\partial V_{\vec{w}}}{\partial \vec{x}} \cdot f\big(\vec{x}, \pi(\vec{x})\big).$$

Initial conditions are

$$\begin{cases} \vec{w}(0) & \text{is a chosen at random,} \\ \vec{e}(0) = \vec{0}, & \\ \vec{x}(0) & \text{is the starting state of the system.} \end{cases}$$

Learning parameters are

$$\begin{cases} \eta & \text{the learning rate,} \\ s_\lambda & \text{the learning shortness factor } (\lambda = e^{-s_\lambda \delta t}). \end{cases}$$

Tsitsiklis and Van Roy's bound becomes

$$E \leq \Big(1 + \frac{s_\lambda}{s_\gamma}\Big) E^*.$$

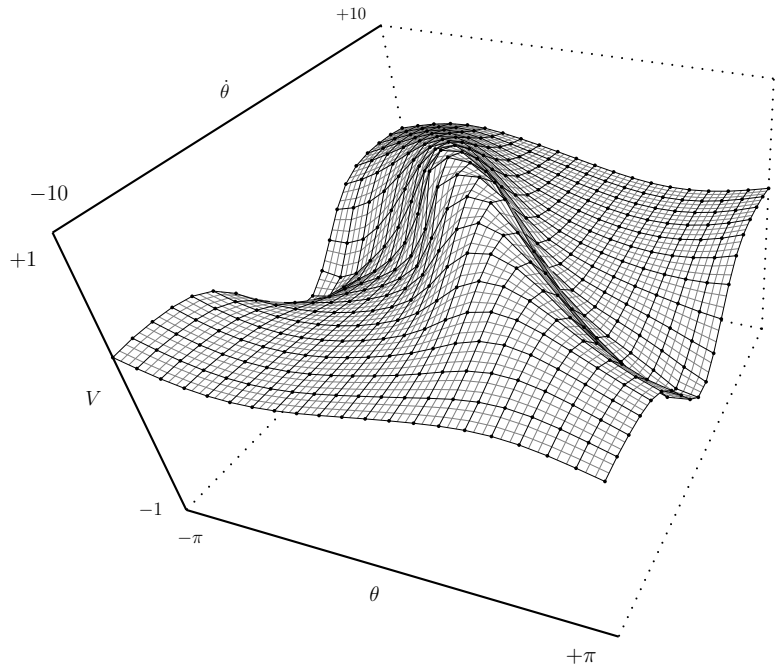


Figure 3.5: Value function obtained by value iteration with a 20×20 discretization of the pendulum swing-up task. Nodes are at the intersection of thick black lines. Thin gray lines show multilinear interpolations between nodes.

3.2.4 Back to Grid-Based Estimators

Results in this chapter show that value iteration cannot be applied as-is to continuous problems with continuous function approximators. In order to obtain convergence to the right value function, it is necessary to use methods based on the analysis of complete trajectories. Algorithms presented in Chapter 1 did not require this, but one might naturally think that they would also benefit from online training.

In order to test this, let us compare value iteration applied to the pendulum swing-up task (Figure 3.5) to policy iteration with Monte-Carlo trials (Figures 3.6–3.9). Both algorithms converge, but to significantly different value functions. By comparing to the 1600 × 1600 discretization shown on Figure 1.12, it is obvious that Monte-Carlo policy iteration gives a much more accurate result than value iteration.

The reason why value iteration is inaccurate is not only the poor resolution of state-space discretization, but also the accumulation of approximation errors that propagate from discrete state to discrete state. When the esti-

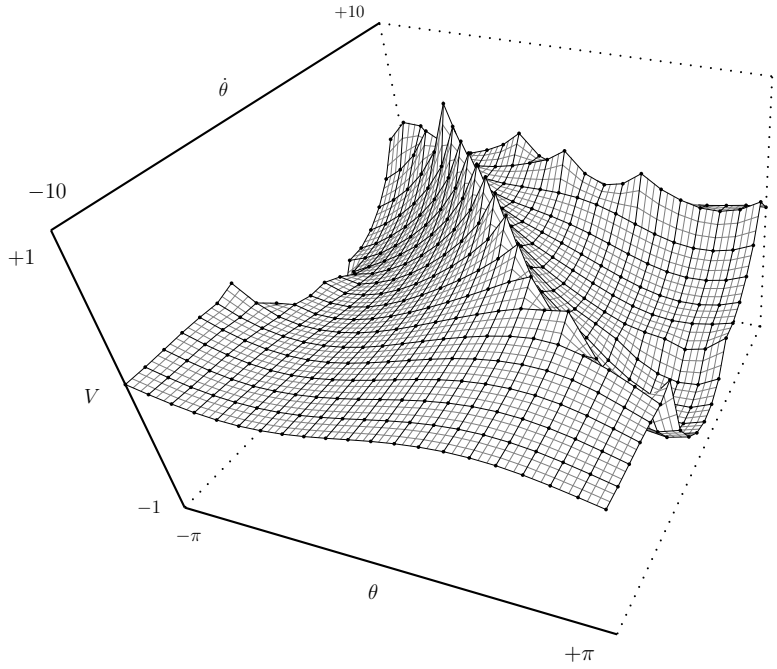


Figure 3.6: Value function obtained after the first step of policy iteration. The initial policy was $\pi_0(\vec{x}) = -u_{max}$.

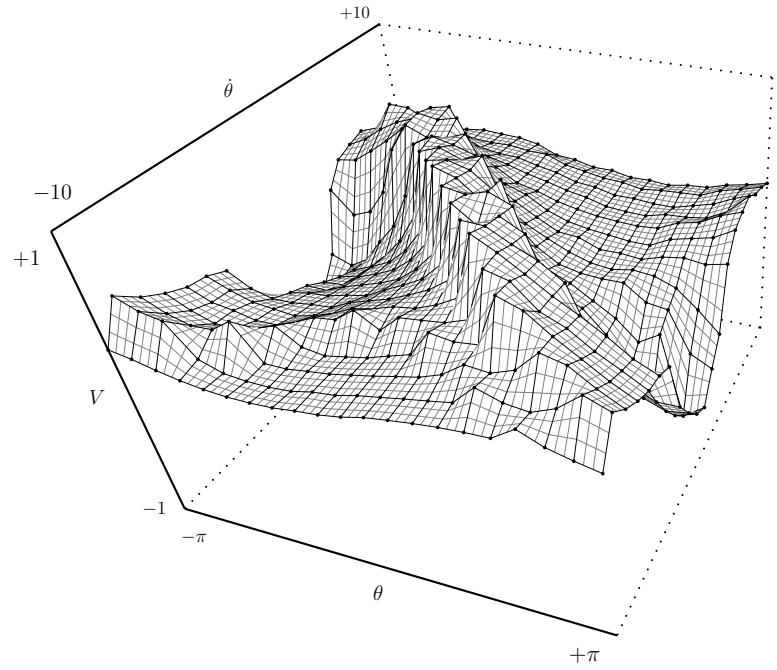


Figure 3.7: Second step of policy iteration

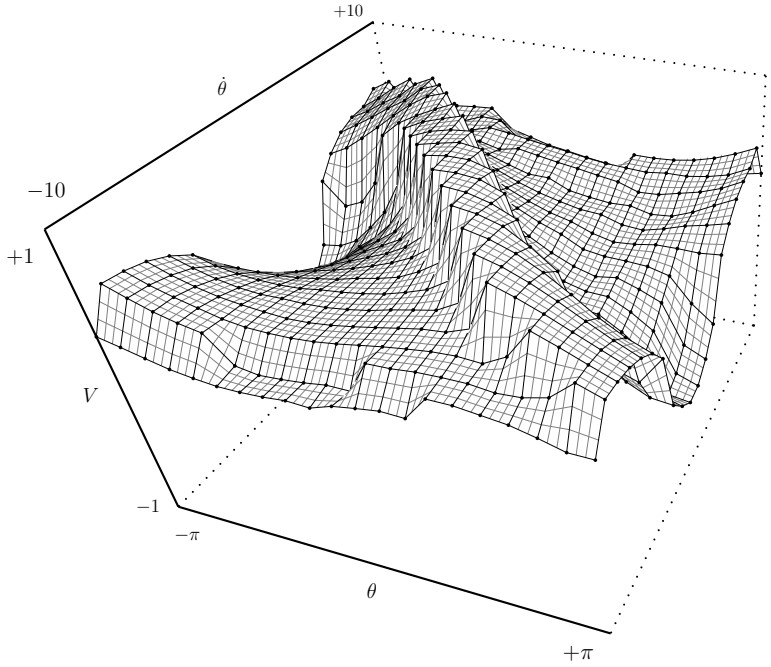


Figure 3.8: Third step of policy iteration

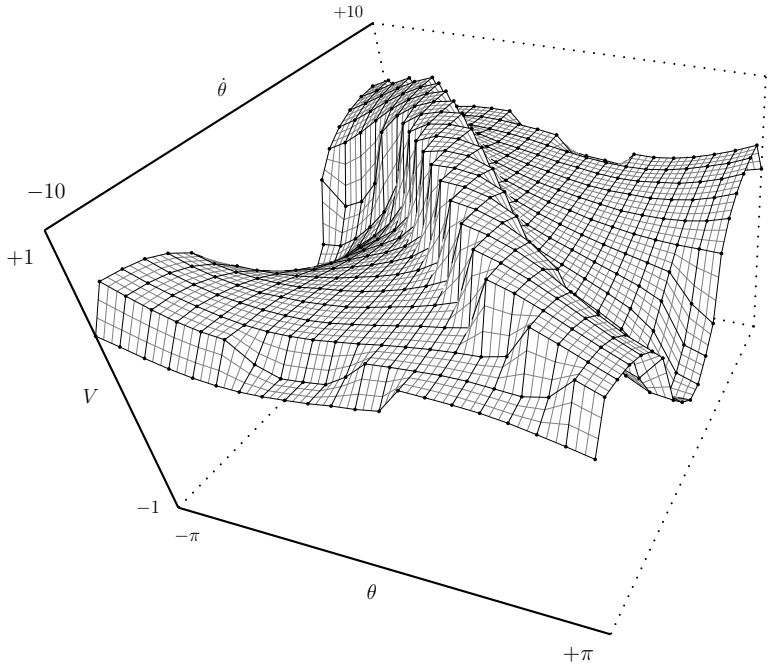


Figure 3.9: Fourth step of policy iteration

mation of the value function is based on the outcome of long trajectories, a much more accurate result is obtained.

3.3 Summary

This chapter shows that combining function approximators with dynamic programming produces a lot of complexity, notably in the analysis of convergence properties of learning algorithms. Among all the possible learning methods, those that have the best stability and accuracy are algorithms that learn over trajectories. Training along trajectories is necessary to keep asymmetry in time and get a more accurate estimation of the value function.

Another important result presented in this chapter is Doya's continuous TD(λ). Thanks to a completely continuous formulation of this reinforcement learning algorithm, it is possible to get rid of time and space discretizations, and of inaccuracies related to this approximation. The next chapter presents some optimizations and refinements of the basic algorithm offered by this late discretization.

Chapter 4

Continuous TD(λ) in Practice

The previous chapter presented the basic theoretical principles of the continuous TD(λ) algorithm. In order to build a working implementation, some more technical issues have to be dealt with. This chapter shows how to find the greedy control, how to use a good numerical integration algorithm, and how to perform gradient descent efficiently with feedforward neural networks.

4.1 Finding the Greedy Control

When integrating the TD(λ) ordinary differential equation, it is necessary to find the greedy control with respect to a value function, that is to say

$$\pi(\vec{x}) = \arg\max_{\vec{u}\in U} \left(r(\vec{x},\vec{u}) - s_\gamma V_{\vec{w}}(\vec{x}) + \frac{\partial V_{\vec{w}}}{\partial \vec{x}} \cdot f(\vec{x},\vec{u}) \right).$$

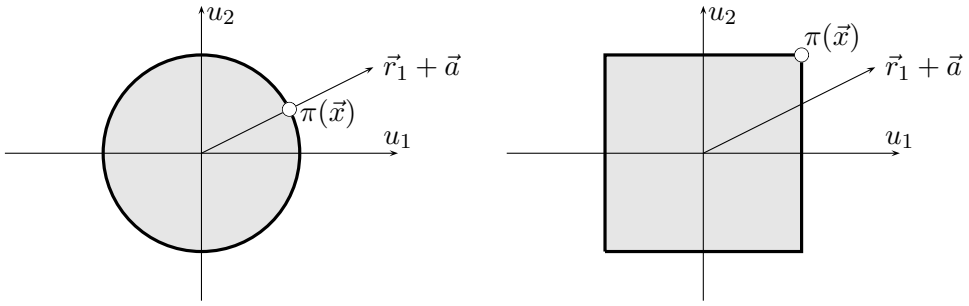
As Doya pointed out [24], a major strength of continuous model-based TD(λ) is that, in most usual cases, the greedy control can be obtained in closed form as a function of the value gradient. It is not necessary to perform a costly search to find it, or to use complex maximization techniques such as those proposed by Baird [8].

In order to obtain this expression of the greedy control, some assumptions have to be made about state dynamics and the reward function. In the case of most motor control problems where a mechanical system is actuated by forces and torques, state dynamics are linear with respect to control, that is to say

$$f(\vec{x},\vec{u}) = A(\vec{x})\vec{u} + \vec{b}(\vec{x}).$$

Thanks to this property, the greedy control equation becomes

$$\pi(\vec{x}) = \arg\max_{\vec{u}\in U} \left(r(\vec{x},\vec{u}) - s_\gamma V_{\vec{w}}(\vec{x}) + \frac{\partial V_{\vec{w}}}{\partial \vec{x}} \cdot \left(A(\vec{x})\vec{u} + \vec{b}(\vec{x})\right) \right).$$



(a) U bounded by a sphere (b) U bounded by a cube

Figure 4.1: Linear programming to find the time-optimal control

By removing terms that do not depend on $\vec{u}$ we get:

$$\begin{aligned}\pi(\vec{x}) &= \arg\max_{\vec{u}\in U}\left(r(\vec{x},\vec{u}) + \frac{\partial V_{\vec{w}}}{\partial \vec{x}}\cdot\big(A(\vec{x})\vec{u}\big)\right)\\ &= \arg\max_{\vec{u}\in U}\left(r(\vec{x},\vec{u}) + \left(A^t(\vec{x})\frac{\partial V_{\vec{w}}}{\partial \vec{x}}\right)\cdot\vec{u}\right)\\ &= \arg\max_{\vec{u}\in U}\big(r(\vec{x},\vec{u}) + \vec{a}\cdot\vec{u}\big)\end{aligned}$$

with

$$\vec{a} = A^t(\vec{x})\frac{\partial V_{\vec{w}}}{\partial \vec{x}}$$

For many usual reward functions, this maximization can be performed easily. The simplest situation is when the reward is linear with respect to control:

$$r(\vec{x},\vec{u}) = r_0(\vec{x}) + \vec{r}_1(x)\cdot\vec{u}.$$

This is typically the case of time-optimal problems. The greedy action becomes

$$\pi(\vec{x}) = \arg\max_{\vec{u}\in U}\big((\vec{r}_1 + \vec{a})\cdot\vec{u}\big).$$

Thus, finding $\pi(\vec{x})$ is a linear programming problem with convex constraints, which is usually easy to solve. As illustrated on Figure 4.1, it is straightforward when U is an hypersphere or an hypercube. In the more general case, $\pi(\vec{x})$ is the farthest point in the direction of $\vec{r}_1+\vec{a}$.

Another usual situation is energy-optimal control. In this case, the reward has an additional term of $\big|\vec{r}_1(\vec{x})\cdot\vec{u}\big|$, which means that the reward is linear by part. So, the greedy action can be obtained with some form of linear programming by part.

Quadratic penalties are also very common. For instance, if the reward is

$$r(\vec{x},\vec{u}) = r_0(\vec{x}) - \vec{u}^t S_2 \vec{u},$$

then the optimal control can be obtained by quadratic programming. Figure 4.2 summarizes the three kinds of control obtained for these three kinds of reward function when a one-dimensional control is used.

4.2 Numerical Integration Method

The most basic numerical integration algorithm for ordinary differential equations is the Euler method. If the equation to be solved is

$$\begin{aligned}\vec{x}(0) &= \vec{x}_0\\ \dot{\vec{x}} &= f\big(\vec{x},\pi(\vec{x})\big),\end{aligned}$$

then, the Euler method consists in choosing a time step δt, and calculating the sequence of vectors defined by

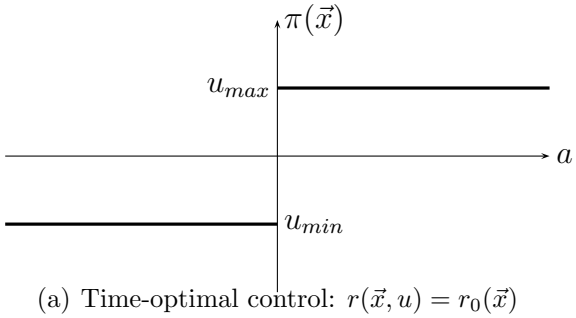
$$\vec{x}_{n+1} = \vec{x}_n + \delta t f\big(\vec{x}_n,\pi(\vec{x}_n)\big).$$

The theory of numerical algorithms provides a wide range of other methods that are usually much more efficient [53]. Their efficiency is based on the assumption that f is smooth. Unfortunately, the equation of TD(λ) rarely meets these smoothness requirements, because the "max" and "arg max" operators in its right-hand side may create discontinuities. In this section, some ideas are presented that can help to handle this difficulty.

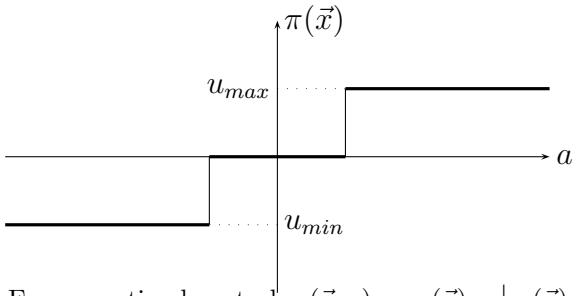
4.2.1 Dealing with Discontinuous Control

Figure 4.3 shows what happens when applying the Euler method to integrate an ordinary differential equation with a discontinuous right hand: $\vec{x}$ might be attracted to a discontinuity frontier and "slide" along this frontier. The time step of the Euler method causes the approximated trajectory to "chatter" around this line. High-order and adaptive time-step methods are totally inefficient in this situation. Besides, if such a high-frequency switching control is applied to a real mechanism, it may trigger unmodelled system resonances and cause physical damage.

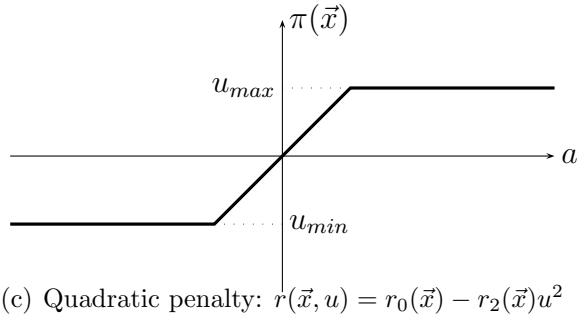
The theory of this kind of differential equation was introduced by Filippov [27]. At the limit of "infinite-frequency switching", the solution of the



(a) Time-optimal control: $r(\vec{x}, u) = r_0(\vec{x})$



(b) Energy-optimal control: $r(\vec{x}, u) = r_0(\vec{x}) - \left|r_1(\vec{x})u\right|$



(c) Quadratic penalty: $r(\vec{x}, u) = r_0(\vec{x}) - r_2(\vec{x})u^2$

Figure 4.2: Some usual cases of greedy 1D control

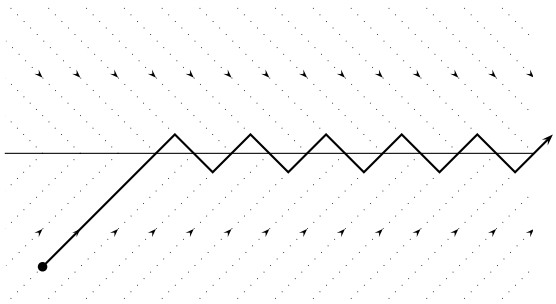


Figure 4.3: Chattering. Dotted lines represent the direction of $\dot{\vec{x}}$.

differential equation is defined as smoothly following the frontier, at a velocity that is a weighted average of velocities on both sides. This velocity is called the Filippov velocity.

A usual method to integrate such a differential equation consists in replacing the discontinuous step in the control by a stiff (but continuous) approximation [76]. Doya used this method in his continuous TD(λ) experiments [24], replacing steps by sigmoids, thus allowing more advanced numerical integration methods to be applied. More precisely, he replaced the optimal "bang-bang" control law

$$u = u_{max} \operatorname{sign}\left(\frac{\partial f}{\partial u} \cdot \frac{\partial V}{\partial \vec{x}}\right)$$

by

$$u = u_{max} \tanh\left(10 \times \frac{\partial f}{\partial u} \cdot \frac{\partial V}{\partial \vec{x}}\right)$$

in the pendulum swing-up task. The smooth control obtained is plotted on Figure 4.5(b).

The control law obtained by this smoothing clearly removes all chattering, but it is also less efficient than the bang-bang control. In particular, it is much slower to get out of the bottom position. This is because this position is a minimum of the value function, so it is close to a switching boundary of the discontinuous optimal control and the sigmoid gives a low value to the control, whereas it should be constant and maximal. Smoothing the control works very well near the upright balance position, because the system is really in sliding mode there, but it works very poorly in the bottom position, because it is a discontinuity that does not attract the system.

Instead of performing such a spatial low-pass filter on the control law, it would make more sense to apply a *temporal* low-pass filter. Actually, the Filippov velocity can be obtained by using a temporal low-pass filter on the velocity. So, since the system dynamics are linear with respect to the control, the Filippov velocity corresponds to applying an equivalent *Filippov control* $\vec{u}_F$, obtained by low-pass filtering the bang-bang control in time. For instance, this could be done by integrating

$$\dot{\vec{u}}_F = \frac{1}{\tau}(\vec{u}^* - \vec{u}_F).$$

Unfortunately, this kind of filtering technique does not help the numerical integration process at all, since it does not eliminate discontinuous variables. Applying a temporal low-pass filter on the bang-bang control only helps when a real device has to be controlled (this is the principle of Emelyanov *et al.*'s [25] "higher-order sliding modes").

Temporal low-pass filtering does not help to perform the numerical integration, but there is yet another possible approach. It consists in finding the control that is optimal for the whole interval of length δt. Instead of

$$\vec{u}^* = \arg\max_{\vec{u}\in U} \left(r(\vec{x},\vec{u}) - s_\gamma V_{\vec{w}}(\vec{x}) + \frac{\partial V_{\vec{w}}}{\partial \vec{x}} \cdot f(\vec{x},\vec{u}) \right),$$

which is optimal when infinitely small time steps are taken, the Filippov control can be estimated by

$$\vec{u}_F = \arg\max_{\vec{u}\in U} V\big(\vec{x}(t+\delta t)\big).$$

$\vec{u}_F$ cannot be obtained in closed form as a function of the value gradient, so it is usually more difficult to evaluate than $\vec{u}^*$. Nevertheless, when the dimension of the control space is 1, it can be estimated at virtually no cost by combining the value-gradient information obtained at $\vec{x}_n$ with the value obtained at $\vec{x}_{n+1}$ to build a second-order approximation of the value function. Figure 4.4 and Algorithm 4.1 illustrate this technique. Figure 4.5(c) shows the result obtained on the pendulum task. It still has a few discontinuities at switching points, but not in sliding mode.

Unfortunately, it is hard to generalize this approach to control spaces that have more than one dimension. It is all the more unfortunate as the system is much more likely to be in sliding mode when the dimension of the control is high (experiments with swimmers in Chapter 7 show that they are in sliding mode almost all the time.) It might be worth trying to estimate $\vec{u}_F$, anyway, but we did not explore this idea further. The key problem is that building and using a second order approximation of the value function is much more difficult when the dimension is higher. So we are still stuck with the Euler method in this case.

4.2.2 Integrating Variables Separately

The TD(λ) differential equation defines the variations of different kinds of variables: the state $\vec{x}$, eligibility traces $\vec{e}$ and weights $\vec{w}$. All these variables have different kinds of dynamics, and different kinds of accuracy requirements, so it might be a good idea to use different kinds of integration algorithms for each of them.

A first possibility is to split the state into position $\vec{p}$ and velocity $\vec{v}$. Mechanical systems are usually under a "cascade" form because they are actually second order differential equations:

$$\begin{cases} \dot{\vec{p}} = \vec{v} \\ \dot{\vec{v}} = f_v(\vec{x},\vec{v},\vec{u}). \end{cases}$$

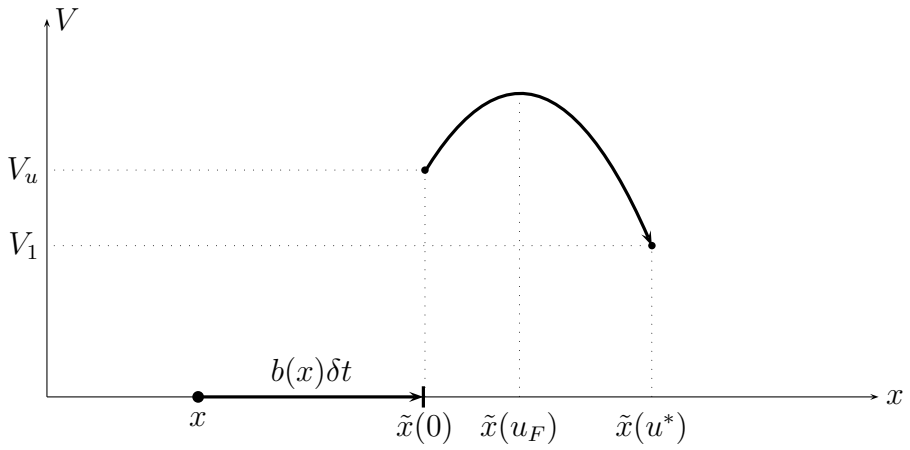
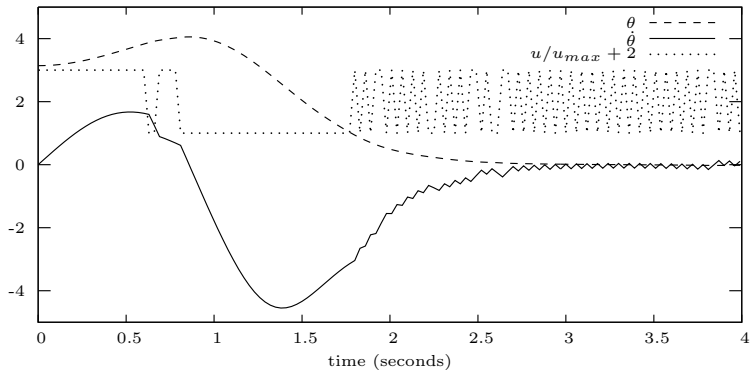


Figure 4.4: $\tilde{x}(u) = x + b(x)\delta t + A(x)u\delta t$. The Filippov control u_F can be evaluated with a second order estimation of the value function.

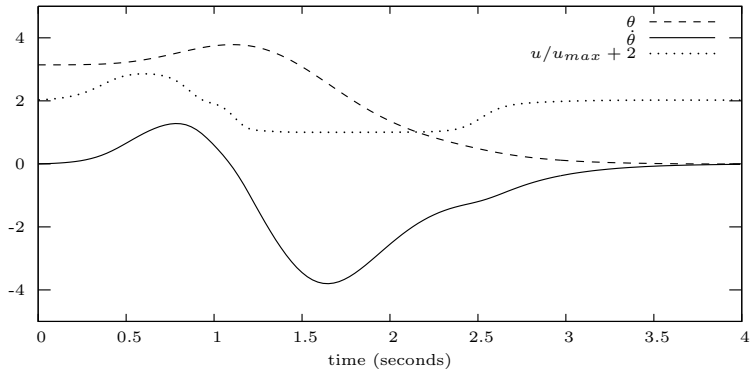
Algorithm 4.1 One-dimensional Filippov control

$\vec{x}_u \leftarrow \vec{x} + \vec{b}(\vec{x})\delta t$
$V_u \leftarrow V_{\vec{w}}(\vec{x}_u)$
$u^* \leftarrow \arg\max_u \left(\partial V_{\vec{w}}(\vec{x}_u)/\partial\vec{x} \cdot A(\vec{x})u\right)$
$\dot{V}_u \leftarrow \partial V_{\vec{w}}(\vec{x}_u)/\partial\vec{x} \cdot A(\vec{x})u^*$ {Theoretical $\dot{V}$ in an infinitely small time step}

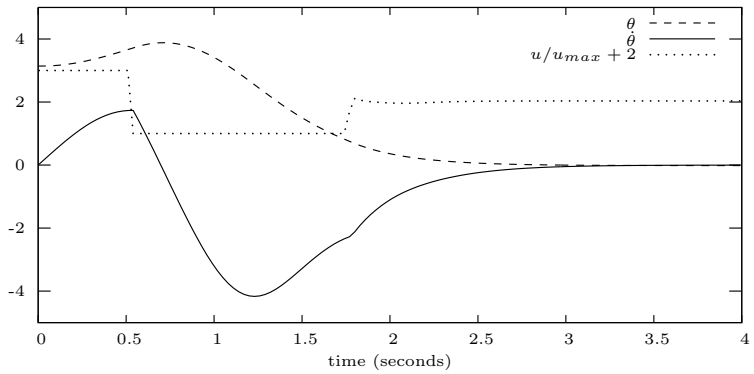
$V_1 \leftarrow V_{\vec{w}}\big(\vec{x}_u + A(\vec{x})u^*\delta t\big)$
$\dot{V}_e \leftarrow (V_1 - V_u)/\delta t$ {Effective $\dot{V}$ on a step of length δt}
if $2\dot{V}_e < \dot{V}_u$ **then**
  $u^* \leftarrow u^* \times \dot{V}_u/\big(2(\dot{V}_u - \dot{V}_e)\big)$ {Filippov control}
end if



(a) Bang-bang control



(b) Smooth control



(c) Filippov control

Figure 4.5: Some control methods for time-optimal pendulum swing-up. The value function is approximated with a 15×15 normalized Gaussian network.

It is possible to take advantage of this in the numerical integration algorithm, by cascading it too. For instance, this can be done by using a first order integration for $\vec{v}$ and a second order integration for $\vec{p}$ (algorithm 4.2).

Algorithm 4.2 Integration with Split Position and Velocity

while End of trial not reached **do**
  $\vec{v}_{i+1} \leftarrow \vec{v}_i + \dot{\vec{v}}_i \delta t$
  $\vec{p}_{i+1} \leftarrow \vec{p}_i + \frac{1}{2}(\vec{v}_i + \vec{v}_{i+1})\delta t$ {Second order integration}
  $\vec{e}_{i+1} \leftarrow \vec{e}_i + \dot{\vec{e}}_i \delta t$
  $\vec{w}_{i+1} \leftarrow \vec{w}_i + \eta \mathcal{H}_i \vec{e}_i \delta t$
  $i \leftarrow i + 1$
end while

When system dynamics are stiff, that is to say there can be very short and very high accelerations (this is typically the case of swimmers of Chapter 7), the Euler method can be unstable and requires short time steps. In this case, it might be a good idea to separate the integration of eligibility traces and weights from the integration of state dynamics, using a shorter time step for the latter.

4.2.3 State Discontinuities

We have considered continuous and differentiable state dynamics so far. Many interesting real problems, however, have discrete deterministic discontinuities. A typical example is the case of a mechanical shock that causes a discontinuity in the velocity. Another typical case will be met in Section 6.3: the distance to the obstacle ahead for a robot in a complex environment may have discontinuities when the heading direction varies continuously.

In these discontinuous cases, the main problem is that the Hamiltonian cannot be computed with the gradient of the value function. It can be evaluated using an approximation over a complete interval:

$$\begin{aligned}\mathcal{H} &= r - s_\gamma V + \dot{V} \\ &\approx r - s_\gamma \frac{V(t+\delta t) + V(t)}{2} + \frac{V(t+\delta t) - V(t)}{\delta t}.\end{aligned}$$

This incurs some cost, mainly because the value function needs to be computed at each value of $\vec{x}$ for two values of $\vec{w}$. But this cost is usually more than compensated by the fact that it might not be necessary to compute the

full gradient $\partial V/\partial \vec{x}$ in order to get the optimal control[1]. Changes to the numerical method are given in algorithm 4.3.

Algorithm 4.3 Integration with accurate Hamiltonian

while End of trial not reached **do**
$\vec{v}_{i+1} \leftarrow \vec{v}_i + \dot{\vec{v}}_i \delta t$ {or a discontinuous jump}
$\vec{p}_{i+1} \leftarrow \vec{p}_i + \frac{1}{2}(\vec{v}_i + \vec{v}_{i+1})\delta t$ {or a discontinuous jump}
$\vec{e}_{i+1} \leftarrow \vec{e}_i + \dot{\vec{e}}_i \delta t$
$V_1 \leftarrow V_{\vec{w}_i}(\vec{x}_{i+1})$
$V_0 \leftarrow V_{\vec{w}_i}(\vec{x}_i)$
$\mathcal{H} \leftarrow r(\vec{x}_i) - s_\gamma(V_1 + V_0)/2 + (V_1 - V_0)/\delta t$
$\vec{w}_{i+1} \leftarrow \vec{w}_i + \eta \mathcal{H} \vec{e}_i \delta t$
$i \leftarrow i + 1$
end while

Practice demonstrated that algorithm 4.3 is better than algorithm 4.2 not only because it runs trials in less CPU time and can handle state discontinuities, but also because it is much more stable and accurate. Experiments showed that with a feedforward neural network as function approximator, algorithm 4.2 completely blows up on the cart-pole task (*cf.* Appendix B) as soon as it starts to be able to maintain the pole in an upright position, whereas algorithm 4.3 works nicely. Figure 4.6 shows what happens at stationary points and why the discrete estimation of the Hamiltonian is better.

4.2.4 Summary

Integrating the continuous TD(λ) algorithm efficiently is very tricky, in particular because the right-hand side of the equation is discontinuous, and the system is often in sliding mode. A few ideas to deal with this have been presented in this section, but we are still not very far from the lowly Euler method. Using second order information about the value function works very well with a one-dimensional control, and generalizing this idea to higher dimensions seems a promising research direction.

[1] In fact, it is possible to completely remove this cost by not updating weights during an episode. Weight changes can be accumulated in a separate variable, and incorporated at the end of the episode.

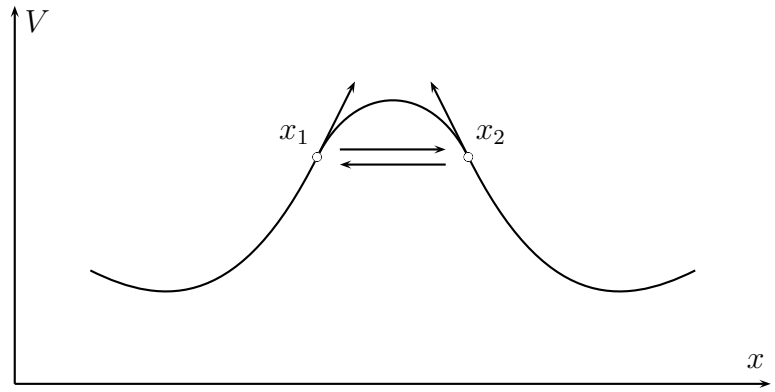


Figure 4.6: Example of a stationary point. The state alternates between x_1 and x_2. The discrete estimation of $\dot{V}$ has an average of zero, whereas its computation with the gradient of the value function has a big positive value.

4.3 Efficient Gradient Descent

As explained in Chapter 2, the steepest descent algorithm used in Equation 3.2.1 is known to perform poorly for ill-conditioned problems. Most of the classical advanced methods that are able to deal with this difficulty are batch algorithms (scaled conjugate gradient [39], Levenberg Marquardt, RPROP [55], QuickProp [26], ...). TD(λ) is incremental by nature since it handles a continuous infinite flow of learning data, so these batch algorithms are not well adapted.

It is possible, however, to use second order ideas in on-line algorithms [49, 60]. Le Cun *et al.* [37] recommend a "stochastic diagonal Levenberg Marquardt" method for supervised classification tasks that have a large and redundant training set. TD(λ) is not very far from this situation, but using this method is not easy because of the special nature of the gradient descent used in the TD(λ) algorithm. Evaluating the diagonal terms of the Hessian matrix would mean derivating with respect to each weight the total error gradient over one trial, which is equal to

$$\int_{t_0}^{t_f} -\mathcal{H}(t)\vec{e}(t)dt.$$

This is not impossible, but still a bit complicated. An additional vector of eligibility traces for second-order information would be required, which would make the implementation more complex.

Another method, the Vario-η algorithm [47], can be used instead. It is well adapted for the continuous TD(λ) algorithm, and it is both virtually costless in terms of CPU time and extremely easy to implement.

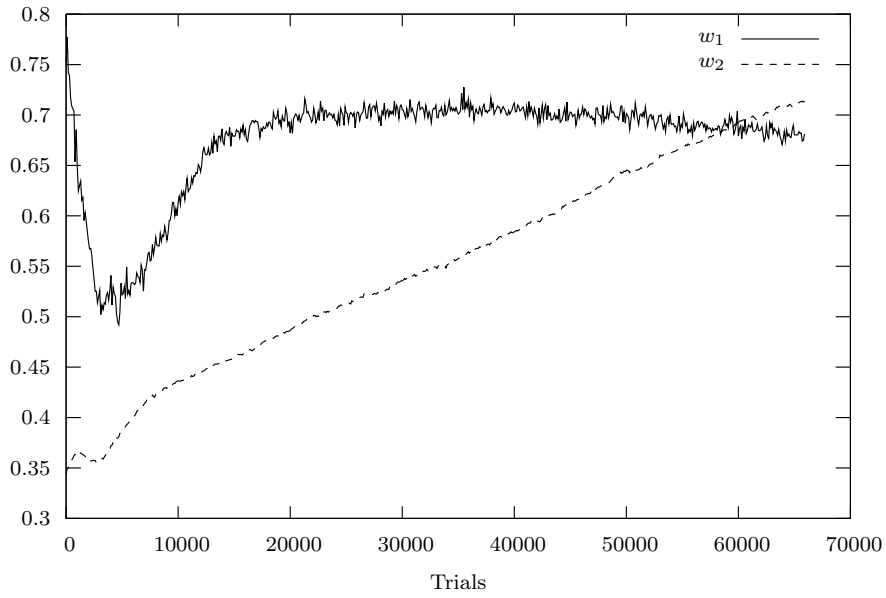


Figure 4.7: Variations of two weights when applying the basic TD(λ) algorithm to the pendulum swing-up task. A value is plotted every 100 trials. The function approximator used is a 66-weight feedforward neural network.

4.3.1 Principle

Figure 4.7 shows the typical variations of two weights during learning. In this figure, the basic algorithm (derived from Equation 3.2.1) was applied to a simple control problem and no special care was taken to deal with ill-conditioning. Obviously, the error function was much more sensitive to w_1 than to w_2. w_2 varied very slowly, whereas w_1 converged rapidly. This phenomenon is typical of ill-conditioning (see Figure 2.4).

Another effect of these different sensitivities is that w_1 looks much more noisy than w_2. The key idea of Vario-η consists in measuring this noise to estimate the sensitivity of the error with respect to each weight, and scale individual learning rates appropriately. That is to say, instead of measuring ill-conditioning of the Hessian matrix, which is the traditional approach of efficient gradient-descent algorithms, ill-conditioning is measured on the covariance matrix.

4.3.2 Algorithm

In theory, it would be possible to obtain a perfect conditioning by performing a principal component analysis with the covariance matrix. This approach

is not practical because of its computational cost, so a simple analysis of the diagonal is performed:

$$v_i(k+1) = (1-\beta)v_i(k) + \beta\left(\frac{w_i(k+1) - w_i(k)}{\eta_i(k)}\right)^2,$$
$$\eta_i(k) = \frac{\eta}{\sqrt{v_i(k)} + \varepsilon}.$$

k is the trial number. $v_i(0)$ is a large enough value. β is the variance decay coefficient. A typical choice is $1/100$. ε is a small constant to prevent division by zero. $\eta_i(k)$ is the learning rate for weight w_i. This formula assumes that the standard deviation of the gradient is large in comparison to its mean, which was shown to be true empirically in experiments.

4.3.3 Results

Experiments were run with fully-connected feedforward networks, with a linear output unit, and sigmoid internal units. Observations during reinforcement learning indicated that the variances of weights on connections to the linear output unit were usually n times larger than those on internal connections, n being the total number of neurons. The variances of internal connections are all of the same order of magnitude. This means that good conditioning can be simply obtained by scaling the learning rate of the output unit by $1/\sqrt{n}$. This allows to use a global learning rate that is $\sqrt{n}$ times larger and provides a speed-up of about $\sqrt{n}$. The biggest networks used in experiments had 60 neurons, so this is a very significant acceleration.

4.3.4 Comparison with Second-Order Methods

An advantage of this method is that it does not rely on the assumption that the error surface can be approximated by a positive quadratic form. In particular, dealing with negative curvatures is a problem for many second-order methods. There is no such problem when measuring the variance.

It is worth noting, however, that the Gauss-Newton approximation of the Hessian suggested by Le Cun *et al.* is always positive. Besides, this approximation is formally very close to the variance of the gradient (I thank Yann Le Cun for pointing this out to me): the Gauss-Newton approximation of the second order derivative of the error with respect to one weight is

$$\frac{\partial^2 E}{\partial w_{ij}^2} \approx \frac{\partial^2 E}{\partial a_i^2} y_j^2$$

(with notations of Appendix A). This is very close to

$$\left(\frac{\partial E}{\partial w_{ij}}\right)^2 = \left(\frac{\partial E}{\partial a_i}\right)^2 y_j^2.$$

A major difference between the two approaches is that there is a risk that the variance goes to zero as weights approach their optimal value, whereas the estimate of the second order derivative of the error would stay positive. That is to say, the learning rate increases as the gradient of the error decreases, which may be a cause of instability, especially if the error becomes zero. This was not a problem at all in the reinforcement learning experiments that were run, because the variance actually *increased* during learning, and never became close to zero.

4.3.5 Summary

The main advantage of using gradient variance is that, in the case of reinforcement learning problems, it is simpler to implement than an estimation of second order information and still provides very significant speedups. In order to find out whether it is more efficient, it would be necessary to run more experimental comparisons.

Part II

Experiments

Chapter 5

Classical Problems

This chapter gathers results obtained on some classical problems that are often studied in the reinforcement learning literature. These are the simple pendulum swing-up problem (which was presented in previous chapters), the cart-pole swing-up task and the Acrobot. The state spaces of these problem have a low dimension (less than 4), and the control spaces are in one dimension. Thanks to these small dimensionalities, linear function approximators (tile-coding, grid-based discretizations and normalized Gaussian networks) can be applied successfully. In this chapter, controllers obtained with these linear methods are compared to results obtained with feedforward neural networks.

Common Parameters for all Experiments

All feedforward neural networks used in experiments had fully-connected cascade architectures (see Appendix A). Hidden neurons were sigmoids and the output was linear. Weights were carefully initialized, as advised by Le Cun *et al.* [37], with a standard deviation equal to $1/\sqrt{m}$, where m is the number of connections feeding into the node. Inputs were normalized and centered. For each angle θ, $\cos\theta$ and $\sin\theta$ were given as input to the network, to deal correctly with circular continuity. Learning rates were adapted as explained in Chapter 4, by simply dividing the learning rate of the output layer by the square root of the number of neurons. Detailed specification of each problem can be found in Appendix B.

5.1 Pendulum Swing-up

In order to compare feedforward neural networks with linear function approximators, the continuous TD(λ) algorithm was run on the simple pendulum

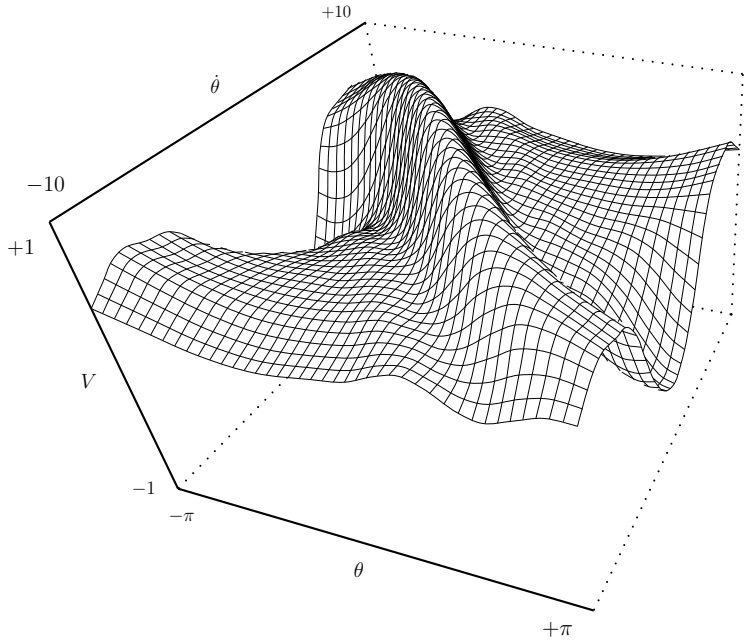


Figure 5.1: Value function obtained with a 15×15 normalized Gaussian network (similar to those used in Doya experiments [24])

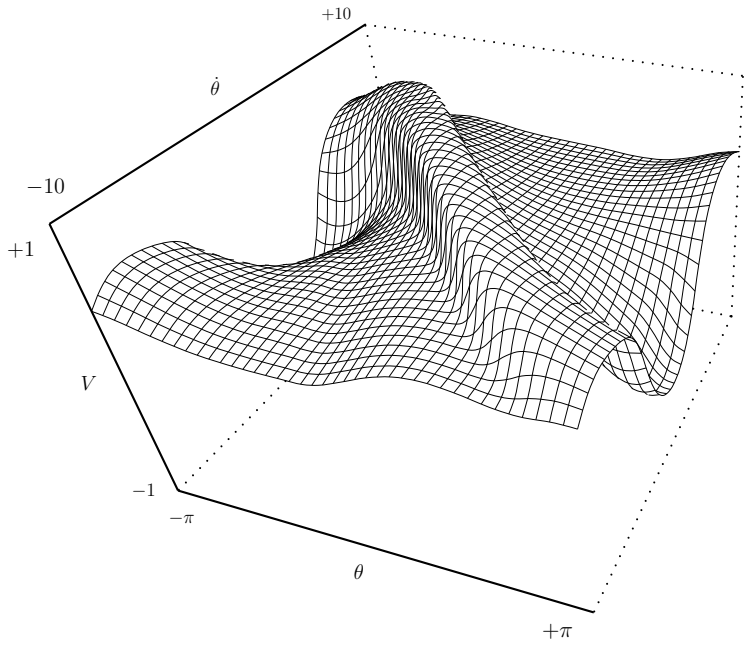


Figure 5.2: Value function learnt by a 12-neuron ($(12 \times 11) \div 2 = 66$-weight) feedforward neural network

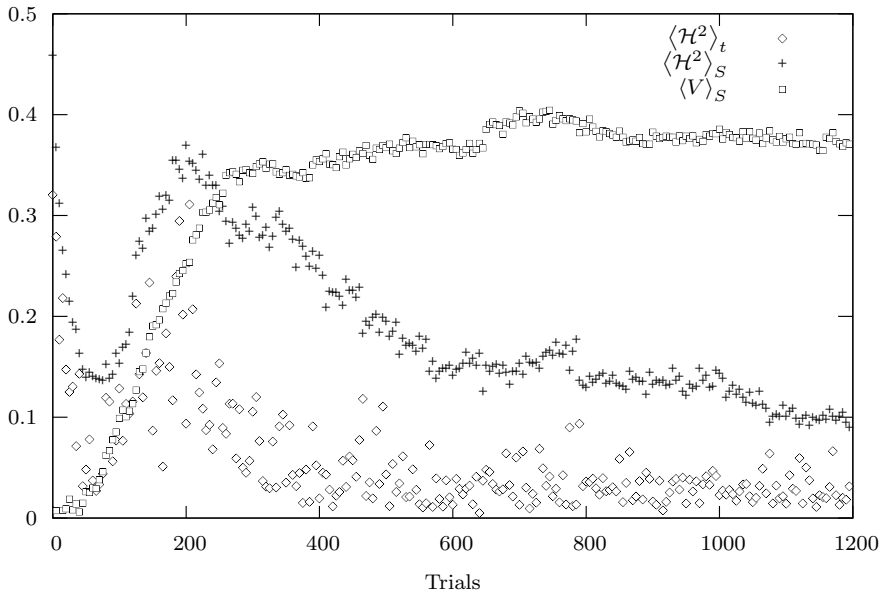


Figure 5.3: Progress of learning for the 15×15 normalized Gaussian network on the pendulum swing-up task. $\langle V \rangle$ and $\langle \mathcal{H} \rangle$ were estimated by averaging 5000 random samples.

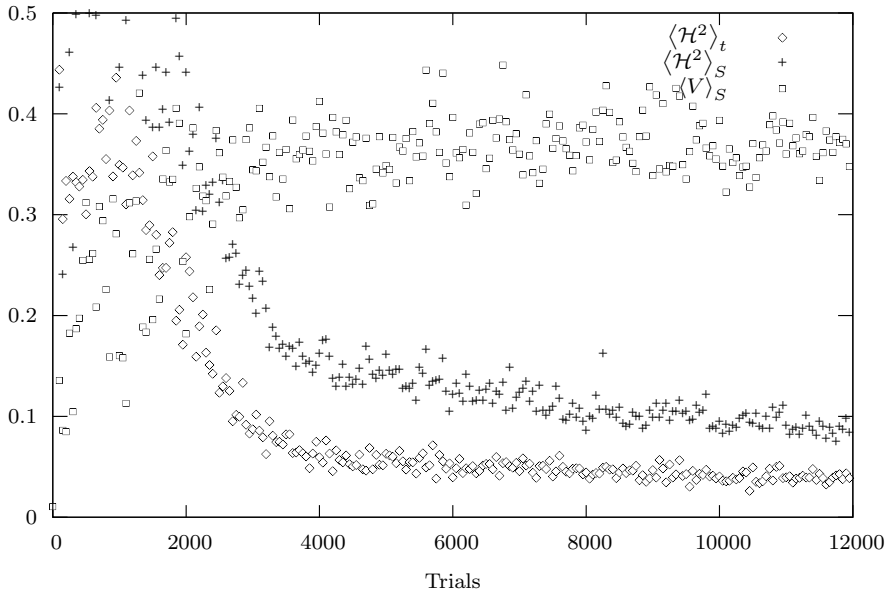


Figure 5.4: Progress of learning for the 12-neuron feedforward network on the pendulum swing-up task

swing-up task with a 12-neuron feedforward neural network and a 15×15 normalized Gaussian network. Parameters for the feedforward network were $\eta = 1.0$, trial length $= 1$ s, $\delta t = 0.03$ s, $s_\lambda = 5$ s^{-1}. Parameters for the normalized Gaussian network were the same, except $\eta = 30$. Results are presented in figures 5.1, 5.2, 5.3 and 5.4.

By comparing figures 5.1 and 5.2 to Figure 1.12, it is clear that the feedforward neural network managed to obtain a much more accurate estimation of the value function. In particular, the stiff sides of the central ridge are much stiffer than with the normalized Gaussian network. Being able to estimate this kind of discontinuity accurately is very important, because value functions are often discontinuous. Sigmoidal units allow to obtain good approximations of such steps.

Another significant difference between these two function approximators is the magnitude of the learning rate and the time it takes to learn. The good locality of the normalized Gaussian network allowed to use a large learning coefficient, whereas the feedforward neural network is much more "sensitive" and a lower learning coefficient had to be used. The consequence is that the feeforward network took many more trials to obtain similar performance (Figure 5.3 and Figure 5.4).

Notice that the plot for the feedforward network was interrupted at 12,000 trials so curves can be easily compared with the normalized Gaussian network, but learning continued to make progress long afterwards, reaching a mean squared Hamiltonian of about 0.047 after 100,000 trials, which is three times less than what the normalized Gaussian network got. Figure 5.2 was plotted after 100,000 trials.

5.2 Cart-Pole Swing-up

The cart-pole problem is a classical motor control task that has been studied very often in control and reinforcement-learning theory. In its traditional form, it consists in trying to balance a vertical pole attached to a moving cart by applying an horizontal force to the cart. Barto first studied this problem in the framework of reinforcement learning [11]. Anderson first used a feedforward neural-network to estimate its value function [2]. Doya studied an extension of the classical balancing problem to a swing-up problem, were the pole has to be swung up from any arbitrary starting position. He managed to solve this problem by using the continuous TD(λ) algorithm with a $7 \times 7 \times 15 \times 15$ normalized Gaussian network on the $(x,\dot{x},\theta,\dot{\theta})$ space [24].

Figures 5.5, 5.6 and 5.7 show the learning progress and typical trajectories obtained by applying the TD(λ) algorithm with a 19-neuron fully-connected

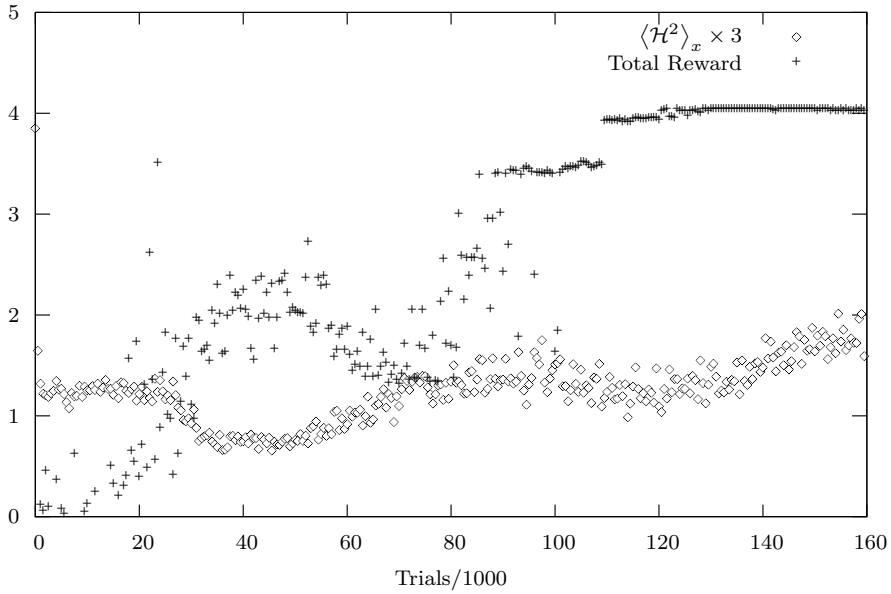


Figure 5.5: Progress of learning for a 19-neuron (171-weight) feedforward neural network on the cart-pole swing-up task

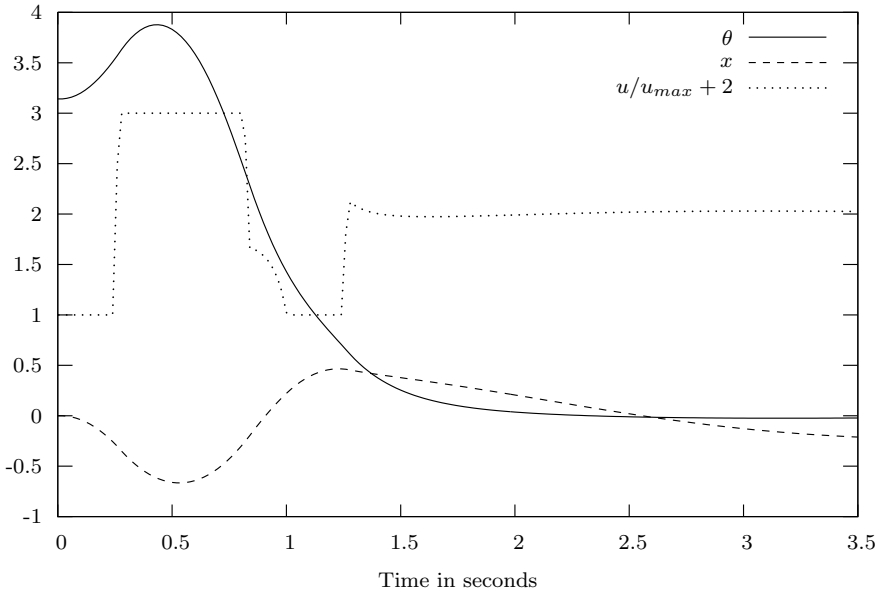
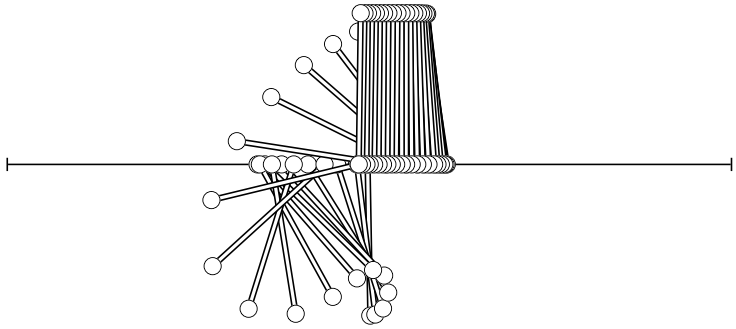
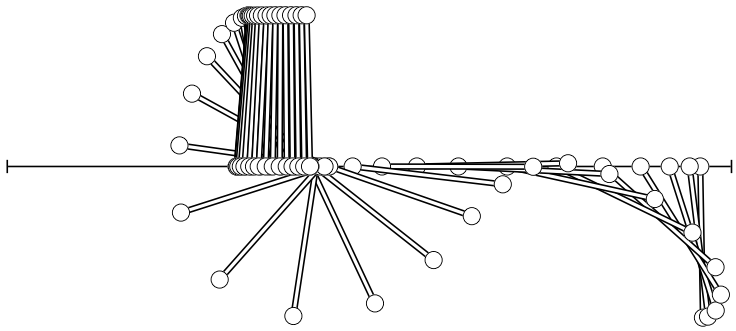


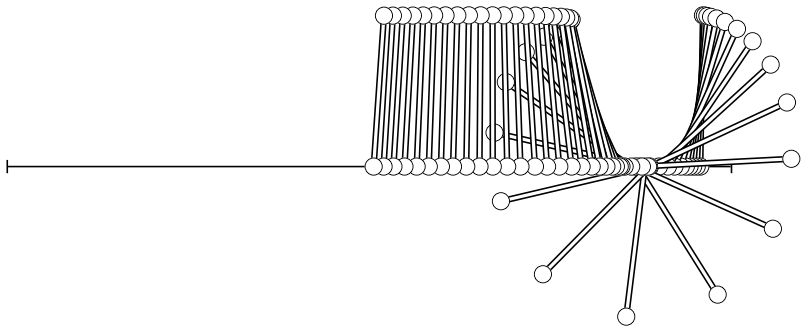
Figure 5.6: Path obtained with a 19-neuron feedforward neural network on the cart-pole swing-up task



(a) Swing-up from the bottom position



(b) Swing-up, starting close to the right border



(c) Starting from the upright position, close to the right border

Figure 5.7: Cart-Pole trajectories. The time step of figures is 0.08 seconds.

feedforward neural network. The learning coefficient was $\eta = 0.03$, episodes were 5-second long, $s_\lambda = 2\text{s}^{-1}$ and the learning algorithm was integrated with an Euler method using a time step of 0.02 seconds. The reward is the height of the pendulum. When the cart bumps into the end of the track, it is punished by jumping into a stationary state with a constant negative reward (full details can be found in Appendix B.)

Results of simulation show that the 19-neuron feedforward network learnt to swing the pole correctly. In comparison with Doya's results, learning took significantly more simulated time than with a normalized Gaussian network (about 500,000 seconds instead of about 20,000). This is similar to what had been observed with the simple pendulum. Trajectories obtained look at least as good as Doya's. In particular, the feedforward neural-network managed to balance the pole in one less swing, when starting from the bottom position in the middle of the track (Figure 5.7). This might be the consequence of a more accurate estimation of the value function obtained thanks to the feedforward neural network, or an effect of the better efficiency of Filippov control (Doya used smoothed control).

A significant difference between the feedforward neural network used here and Doya's normalized Gaussian network is the number of weights: 171 instead of 11,025. This is a strong indication that feedforward neural networks are likely to scale better to problems in large state spaces.

5.3 Acrobot

The acrobot is another very well-known optimal control problem [65] (see Appendix B.2). Here are some results of reinforcement learning applied to this problem:

- Sutton [67] managed to build an acrobot controller with the Q-learning algorithm using a tile-coding approximation of the value function. He used $u_{max} = 1$ Nm, and managed to learn to swing the endpoint of the acrobot above the bar by an amount equal to one of the links, which is somewhat easier than reaching the vertical position.

- Munos [46] used an adaptive grid-based approximation trained by value iteration, and managed to teach the acrobot to reach the vertical position. He used $u_{max} = 2$ Nm, which makes the problem easier, and reached the vertical position with a non-zero velocity, so the acrobot could not keep its balance.

- Yoshimoto *et al.* [80] managed to balance the acrobot with reinforcement learning. They used $u_{max} = 30$ Nm, which makes the problem

much easier than with 1 Nm.

Boone [18, 17] probably obtained the best controllers, but the techniques he used are not really reinforcement learning (he did not build a value function) and are very specific to this kind of problem.

In our experiments, physical parameters were the same as those used by Munos ($u_{max} = 2$ Nm). The value function was estimated with a 30-neuron (378-weight) feed-forward network. $\eta = 0.01$, trial length = 5s, $\delta t = 0.02$s, $s_\lambda = 2$. Figure 5.8 shows the trajectory of the acrobot after training. It managed to reach the vertical position at a very low velocity, but it could not keep its balance.

Figure 5.9 shows a slice of the value function obtained. It is not likely that a linear function approximator of any reasonable size would be able to approximate such a narrow ridge, unless some extremely clever choices of basis functions were made.

5.4 Summary

These experiments show that a feedforward neural network can approximate a value function with more accuracy than a normalized Gaussian network, and with many less weights. This is all the more true as the dimension of the state space is high, which gives and indication that feedforward neural networks might be able to deal with some significantly more complex problems. This superior accuracy is obtained at the expense of requiring many more trials, though.

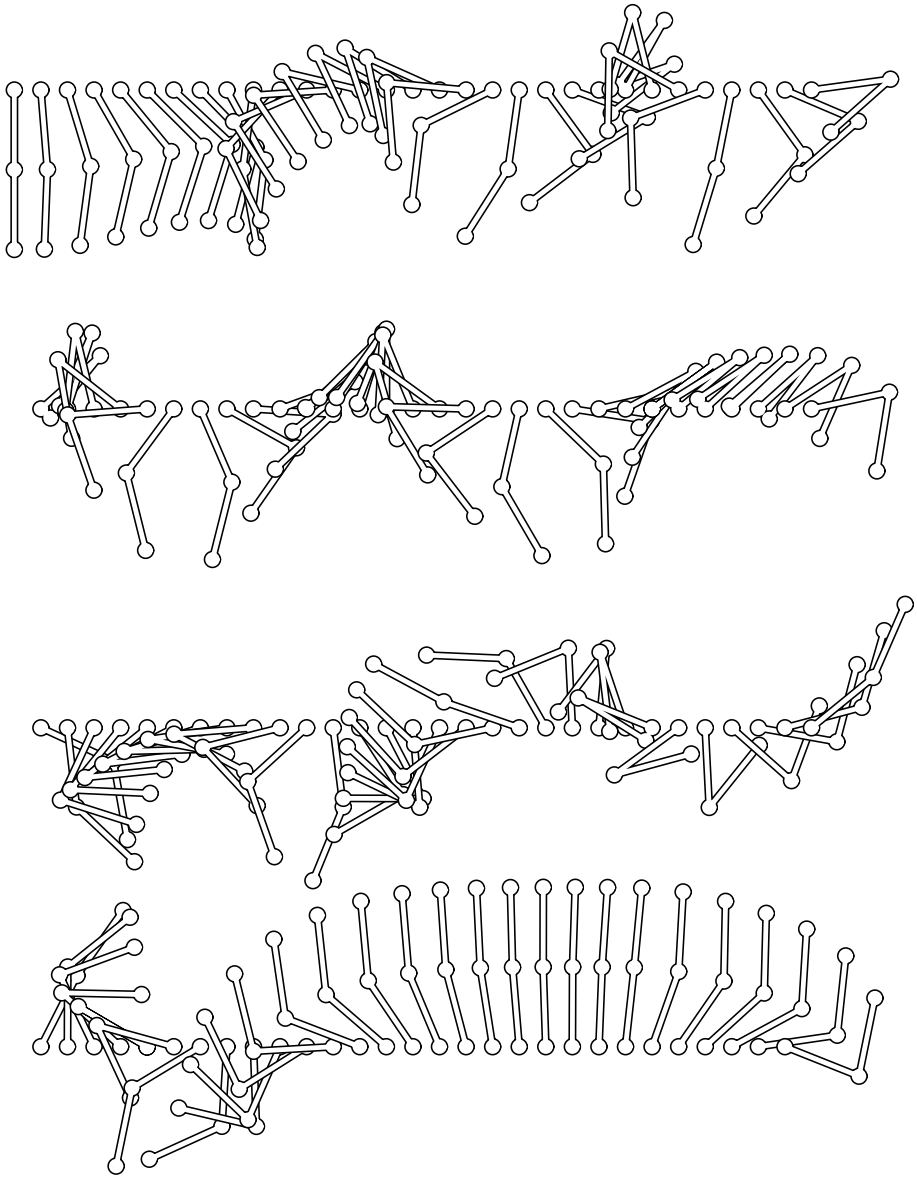


Figure 5.8: Acrobot trajectory obtained with a 30-neuron (378-weight) feed-forward neural network. The time step of this animation is 0.1 seconds. The whole sequence is 12-second long.

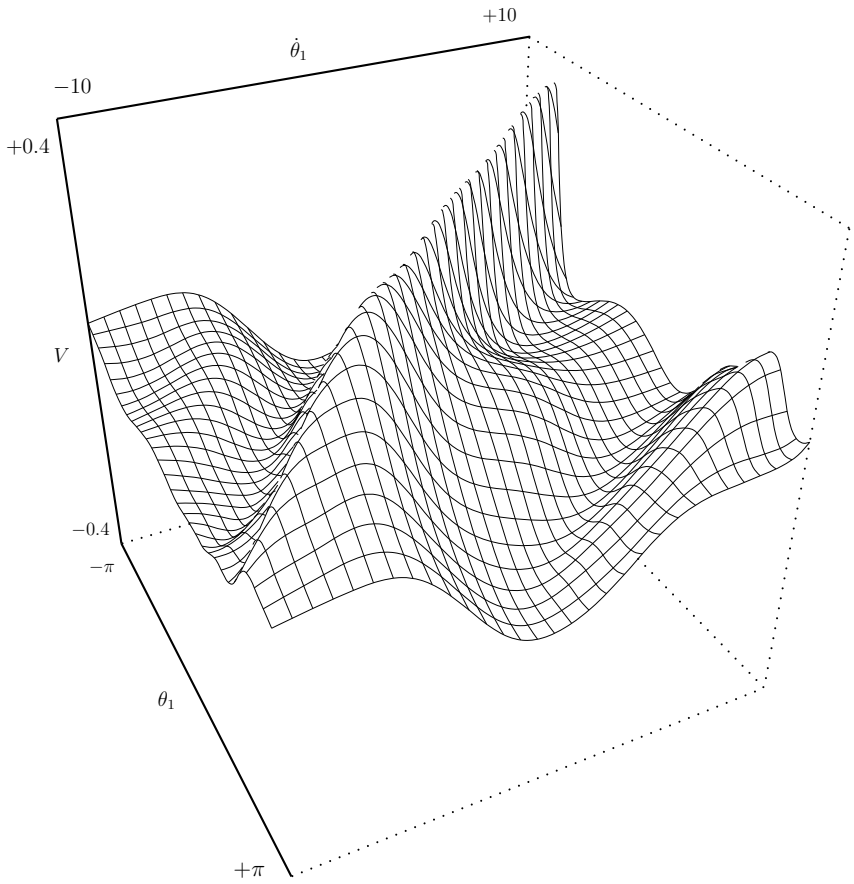


Figure 5.9: A slice of the Acrobot value function ($\theta_2 = 0$, $\dot{\theta}_2$=0), estimated with a 30-neuron (378-weight) feed-forward neural network.

Chapter 6

Robot Auto Racing Simulator

This chapter contains a description of attempts at applying reinforcement learning to a car driver in the Robot Auto Racing Simulator. This is a problem with 4 state variables and 2 control variables that requires a lot of accuracy.

6.1 Problem Description

The Robot Auto Racing Simulator was originally designed and written by Mitchell E. Timin in 1995 [71]. This is the description he gave in his original announcement:

> The Robot Auto Racing Simulation (RARS) is a simulation of auto racing in which the cars are driven by robots. Its purpose is two-fold: to serve as a vehicle for Artificial Intelligence development and as a recreational competition among software authors. The host software, including source, is available at no charge.

The simulator has undergone continuous development since then and is still actively used in a yearly official Formula One season.

6.1.1 Model

The Robot Auto Racing Simulator uses a very simple two-dimensional model of car dynamics. Let $\vec{p}$ and $\vec{v}$ be the two-dimensional vector indicating the position and velocity of the car. Let $\vec{x} = (\vec{p}, \vec{v})^t$ be the state variable of the system. Let $\vec{u}$ be the command. A simplified model of the simulation is

described by the differential equations:

$$\begin{cases} \dot{\vec{p}} = \vec{v} \\ \dot{\vec{v}} = \vec{u} - k\|\vec{v}\|\vec{v} \end{cases} \tag{6.1.1}$$

The command $\vec{u}$ is restricted by the following constraints (Figure 6.1):

$$\begin{cases} \|\vec{u}\| \leq a_t \\ \vec{u} \cdot \vec{v} \leq \dfrac{P}{m} \end{cases} \tag{6.1.2}$$

k, a_t, P and m are numerical constants that define some mechanical characteristics of the car:

- k = air-friction coefficient (aerodynamics of the car)
- P = maximum engine power
- a_t = maximum acceleration (tires)
- m = mass of the car

Numerical values used in official races are $k = 2.843 \times 10^{-4}$ kg $\cdot$ m^{-1}, $a_t = 10.30$ m $\cdot$ s^{-2} and $P/m = 123.9$ m^2 $\cdot$ s^{-3}.

In fact, the physical model is a little more complex and takes into consideration a friction model of tires on the track that makes the friction coefficient depend on slip speed. The mass of the car also varies depending on the quantity of fuel, and damage due to collisions can alter the car dynamics. The simplification proposed above does not significantly change the problem and will make further calculations much easier.

6.1.2 Techniques Used by Existing Drivers

RARS has a long history of racing, and dozens of drivers have been programmed. They can be roughly split into two categories:

- Cars that compute an optimal path first. This method allows to use very costly optimization algorithms. For instance, Jussi Pajala optimized trajectories with A*, Doug Eleveld used a genetic algorithm in DougE1 (and won the 1999 F1 season), and K1999 used gradient descent (and won the 2000 and 2001 F1 seasons)(see Appendix C). A servo-control is used to follow the target trajectory. Drivers based on this kind of methods are usually rather poor at passing (K1999 always stick to its pre-computed trajectory), but often achieve excellent lap time on empty tracks.

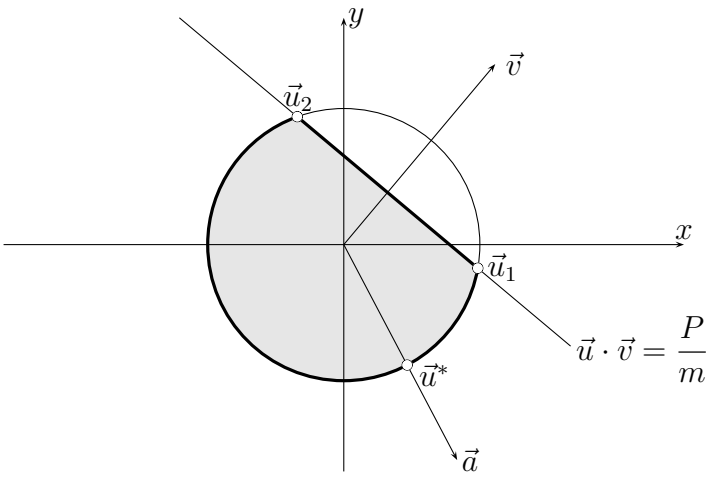


Figure 6.1: The domain U of $\vec{u}$

- Cars that do not compute an optimal path first. They can generate control variables by simply observing their current state, without referring to a fixed trajectory. Felix, Doug Eleveld's second car, uses clever heuristics and obtains very good performance, close to those of DougE1. This car particularly shines in heavy traffic conditions, where it is necessary to drive far away from the optimal path. Another good passer is Tim Foden's Sparky. These car are usually slower than those based on an optimal path when the track is empty.

Reinforcement learning also has been applied to similar tasks. Barto *et al* [10] used a discrete "race track" problem to study real-time dynamic programming. Koike and Doya [34] also ran experiments with a simple dynamic model of a car, but their goal was not racing.

Experiments reported in this chapter were motivated by the project to build a controller using reinforcement learning, that would have had both good trajectories, and good passing abilities. This ambition was probably a bit too big, since our driver failed to obtain either. Still, experiments run on empty tracks were instructive. They are presented in the next sections.

6.2 Direct Application of TD(λ)

In the first experiments, TD(λ) was directly applied to the driving problem, using slightly different (more relevant) state variables to help the neural network to learn more easily. These variables describe the position and velocity of the car relatively to the curve of the track (see Figure 6.2). The track is defined by its curvature as a function of the curvilinear distance to the start

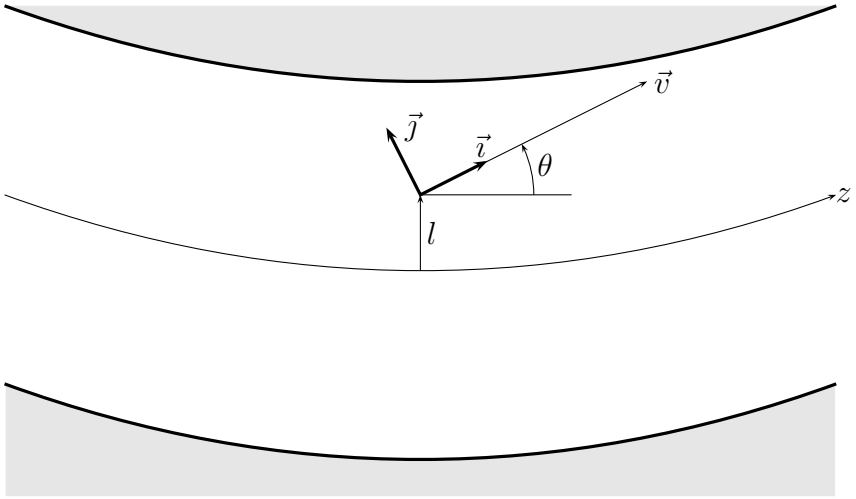


Figure 6.2: The new coordinate system

line $c(z)$. The half-width of the track is $L(z)$. The position and velocity of the car are described by the following variables:

- z is the curvilinear abscissa from the start line along the central lane of the track.
- l is the distance to the center of the track. The car is on the left side of the track when $l > 0$ and on the right side when $l < 0$. It is out of the track when $|l| > L(z)$.
- v is the car velocity ($v = \|\vec{v}\|$)
- θ is the angle of the car velocity relatively to the direction of the track. The car moves toward the left side of the track when $\theta > 0$ and toward the right side when $\theta < 0$.

Besides, the control $\vec{u}$ is decomposed into two components:

$$\vec{u} = u_t\vec{\imath} + u_n\vec{\jmath}$$

where $\vec{\imath}$ is a unit vector pointing in the direction of $\vec{v}$ and $\vec{\jmath}$ is a unit vector pointing to the left of $\vec{\imath}$.

(6.1.1) becomes:

$$\begin{cases} \dot{z} = \dfrac{v\cos\theta}{1 - c(z)l} \\ \dot{l} = v\sin\theta \\ \dot{v} = u_t - kv^2 \\ \dot{\theta} = \dfrac{u_n}{v} - c(z)\dot{z} \end{cases}$$

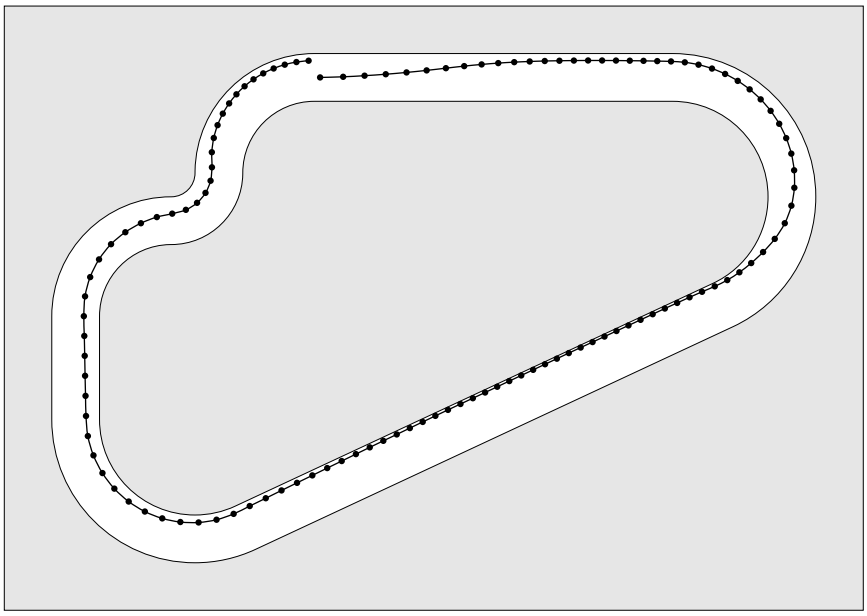


Figure 6.3: Path obtained with a 30-neuron feedforward network. A dot is plotted every 0.5 second.

(6.1.2) becomes:

$$\begin{cases} {u_t}^2 + {u_n}^2 \leq {a_t}^2 \\ u_t v \leq \dfrac{P}{m} \end{cases}$$

The two conditions for this model to be valid are:

$$v > 0, \tag{6.2.1}$$

$$c(z)l \neq 1. \tag{6.2.2}$$

Figure 6.3 shows a simulated path obtained with a 30-neuron feedforward neural network. Learning parameters were

- Trial length: 5.0 s
- Simulation time step: 0.02 s
- $s_\gamma = 1/26$ s^{-1}
- $s_\lambda = 1$ s^{-1}

The reward was the velocity. The controller did not much more than prevent the car from crashing.

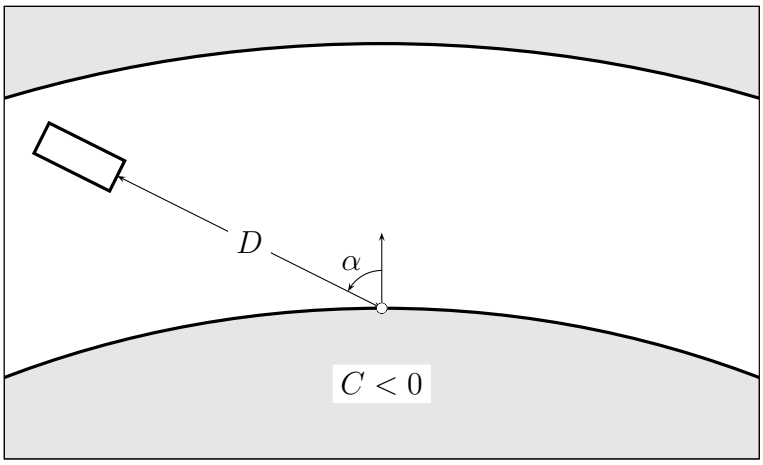


Figure 6.4: D, α and C are some relevant features of the current state

6.3 Using Features to Improve Learning

Features are a key idea in neuro-dynamic programming [15]. It consists in adding some redundant relevant inputs to the function approximator in order to make its learning easier.

The results obtained by the straight implementation of the continuous TD(λ) algorithm were rather poor. The reason is that it was not given data that is easy to handle. For instance, a human driver would have no difficulty to see that a straight fits in the final S curves of `clkwis.trk`. He would use this information to drive fast and straight across them. Learning to do this with the neural network used previously is hard since this straight line has a very complex shape in the (z, l, v, θ) space.

So, some more useful data was added as input to the neural network to make this learning easier. An obvious relevant information for any system that tries to avoid obstacles is the distance D to the wall ahead. The angle α of the velocity with respect to this wall is interesting too. The curvature C of this wall was used as well (see Figure 6.4).

It was necessary to compute the derivatives of these variables in order to be able to compute the optimal control:

$$\begin{cases} \dot{D} = -v + D\dot{\theta}\tan\alpha \\ \dot{\alpha} = \dot{\theta}\left(1 - \dfrac{CD}{\cos\alpha}\right) \\ \dot{C} = 0. \end{cases}$$

It is very important to note that these variables are discontinuous. Even

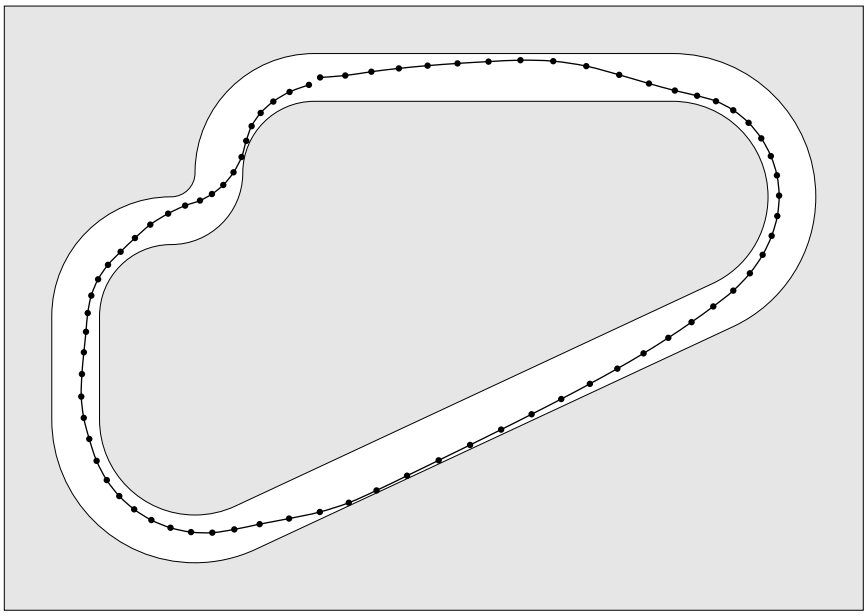


Figure 6.5: Path improved thanks to features (38 seconds instead of 46 seconds)

when they vary continuously, their variations may be very stiff when $\cos(\alpha)$ is close to zero. The consequence of this discontinuity is that the Hamiltonian must be approximated as explained in Section 4.2.3. The algorithm does not converge otherwise. As expected, results obtained were much better than without features (Figure 6.5). There is still a long way to go to K1999's performance, though (K1999 completes one lap in about 30 seconds).

6.4 Conclusion

These experiments show that reinforcement learning applied to the Robot Auto Racing Simulator works, but drivers obtained cannot compete with the best cars. It might have been possible to improve them by adding more relevant features (those used by Felix, for instance), but it is not likely that it would have been enough to close the gap with the best drivers. Using the relative position of the car with respect to a pre-computed path (as features) might be a more interesting alternative, though. Anyway, the problem of creating a good passer that could beat K1999 is still open and very challenging.

Chapter 7

Swimmers

Control problems that have been tested in previous chapters have a rather small dimensionality (≤ 4). Experiment results with these problems tend to indicate that feedforward neural networks could deal accurately with systems with many more independent state variables. In this chapter, this intuition is confirmed by successfully applying feedforward neural networks trained with continuous TD(λ) to complex swimmers. The most complex of them has 12 state variables and 4 control variables, which is significantly beyond the complexity of some of the most difficult dynamical motor control problems that have been solved by reinforcement learning, such as Randløv and Alstrøm's bicycle [54] or Morimoto and Doya's robot [42] (both have 6 state variables).

7.1 Problem Description

The swimmers to be controlled are made of three or more segments (thus providing a large scale of problems, from the simplest to any arbitrarily complex swimmer), connected by joints, and moving in a two-dimensional pool. Control variables are torques applied at these joints. The goal is to swim as fast as possible to the right, by using the friction of water.

The state of a n-segment swimmer is defined by the n angles of its segments, the two Euclidian coordinates G_x and G_y of its center of mass, and the $n+2$ derivatives of these variables. That makes a total of $2n+4$ state variables. The control variables are the $n-1$ torques applied at segment joints. The complete model of this system can be found in Appendix B.

In fact, since the goal is to swim in a given direction ($r(\vec{x},\vec{u}) = \dot{G}_x$), the value function and the optimal control do not depend on G_x and G_y. This means that actually, there are only $2n+2$ significant state variables.

7.2 Experiment Results

Figures 7.1, 7.2, 7.3, 7.4 show swimming styles obtained with a variety of swimmers and function approximators. Since the state space is isotropic (all directions are identical), it is possible to tell a swimmer to swim in any direction by rotating its state variables. On these figures, swimmers were told to perform a U-turn after a few seconds.

Unlike the simpler problems that were presented in the previous chapters, many unstabilities were observed during learning. Most of them are not well understood yet, and would require more investigations.

Figure 7.5 shows instabilities recorded when using a 30-neuron neural network to train a 3-segment swimmer. The reason for these instabilities are not clear, but the shape of the performance curve is strikingly similar to results obtained by Anderson [3] on a simple Q-learning experiment with feedforward neural networks. In the case of Anderson's experiments, instabilities were related to an insufficient number of neurons. In order to test if more neurons would help, the learning experiment was run again with 60 neurons instead of 30. Results are plotted on Figure 7.6. Training was much more stable. After these 2,500,000 trials, performance kept a high level during more than 10,000,000 trials, after which learning was stopped. This does not prove that there would not have been instabilities, but, at least, it is much better than with 30 neurons.

Figure 7.7 shows the learning of a 5-segment swimmer with 60 neurons. Learning made steady progress during the first 2,500,000 trials, but then, performance suddenly collapsed. Unlike the 3-segment instabilities of Figure 7.5, the 5-segment swimmer did not manage to recover from this crisis, and its neural network blew up completely. What happened exactly is still a mystery.

7.3 Summary

These experiment results clearly demonstrate the superior generalization capabilities of feedforward neural networks. In fact, it is extremely unlikely that such a high-dimensional problem could be solved with a linear function approximator. Besides, some instabilities were observed. They might be caused by the insufficient size of the networks used, but this hypothesis would require more investigation. These swimmers clearly show the strength of the dynamic programming approach: with very little computing power, the controller can handle instantaneously any unexpected direction change.

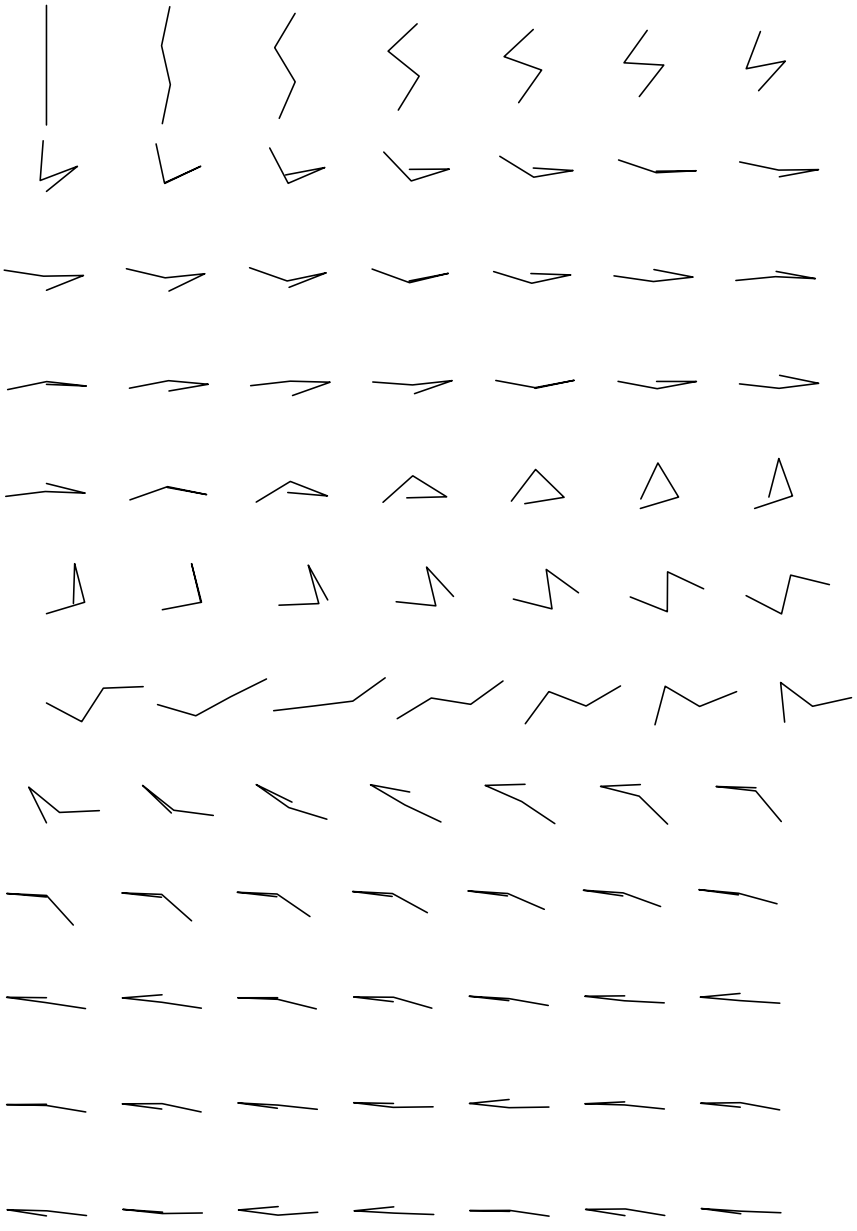


Figure 7.1: A 3-segment swimmer trained with a 30-neuron network. In the first 4 lines of this animation, the target direction is to the right. In the last 8, it is reversed to the left. Swimmers are plotted every 0.1 seconds.

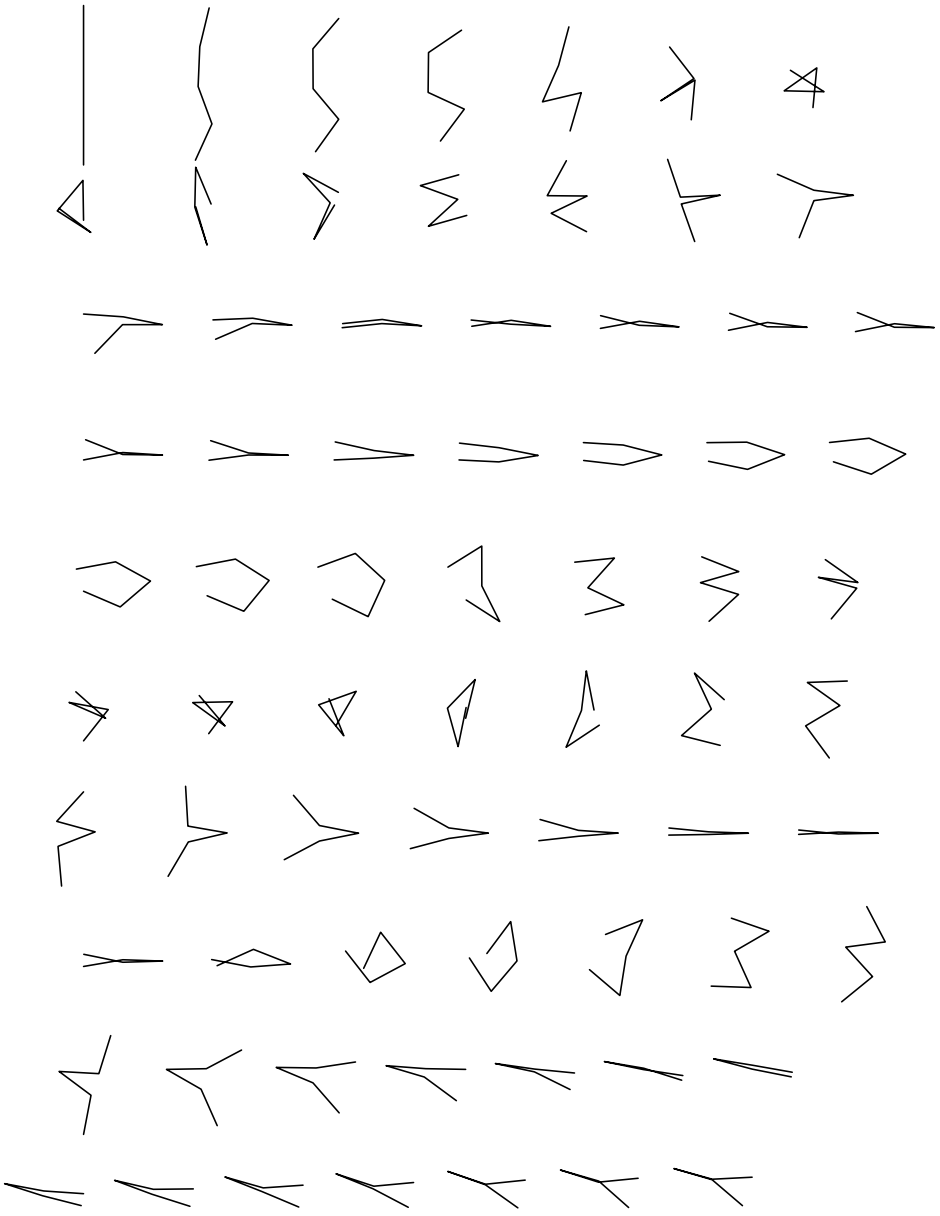


Figure 7.2: A 4-segment swimmer trained with a 30-neuron network. In the first 7 lines of this animation, the target direction is to the right. In the last 3, it is reversed to the left. Swimmers are plotted every 0.2 seconds.

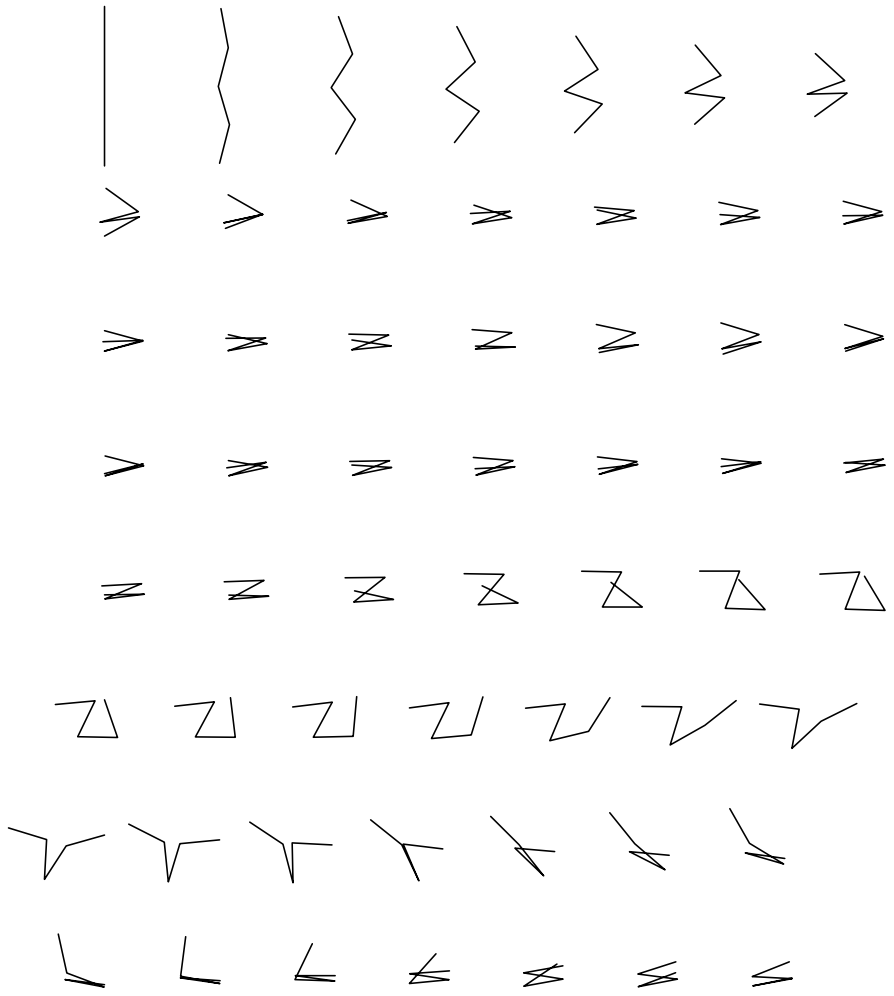


Figure 7.3: A 4-segment swimmer trained with a 60-neuron network. In the first 4 lines of this animation, the target direction is to the right. In the last 4, it is reversed to the left. Swimmers are plotted every 0.1 seconds.

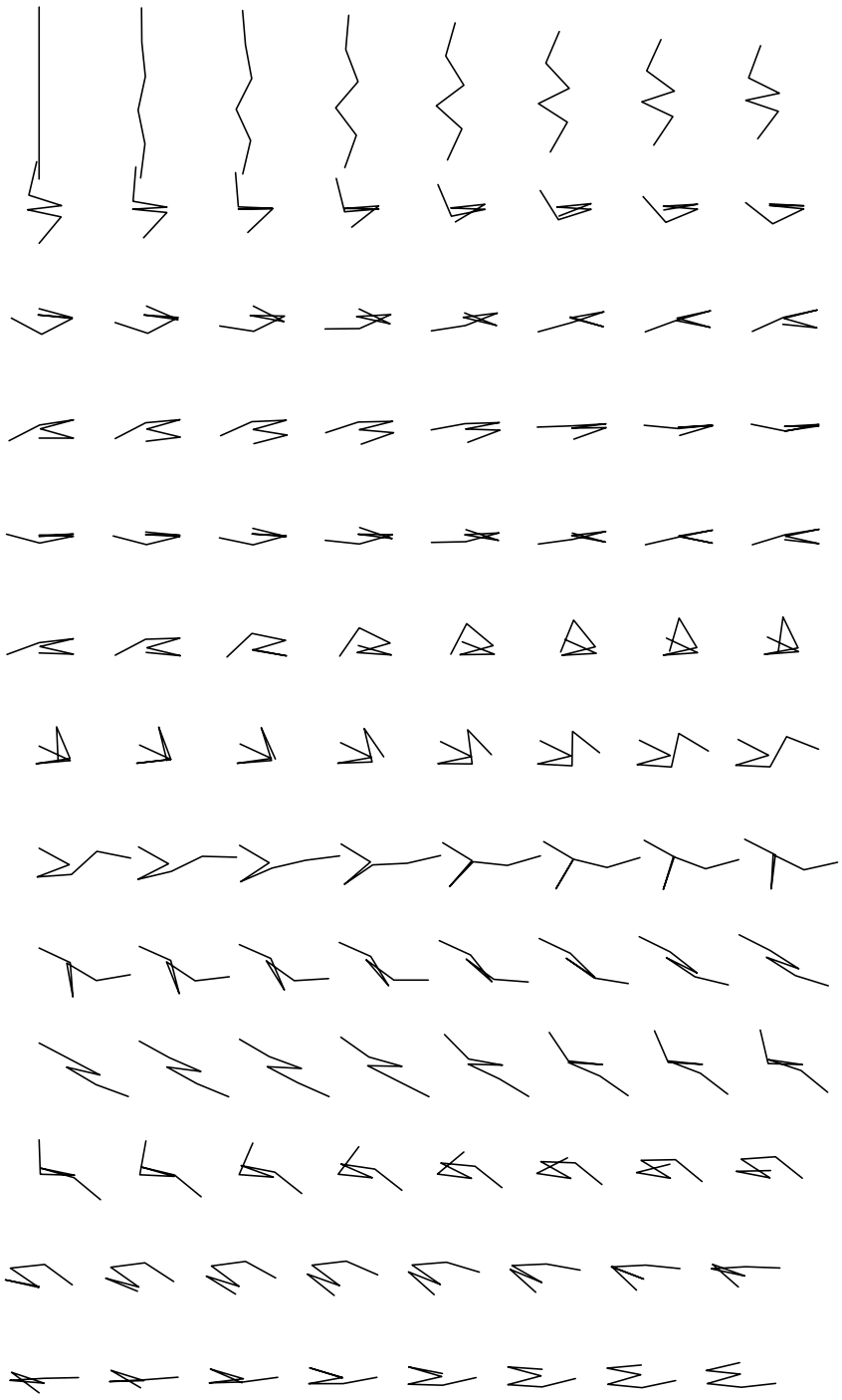


Figure 7.4: A 5-segment swimmer trained with a 60-neuron network. In the first 5 lines of this animation, the target direction is to the right. In the last 8, it is reversed to the left. Swimmers are plotted every 0.1 seconds.

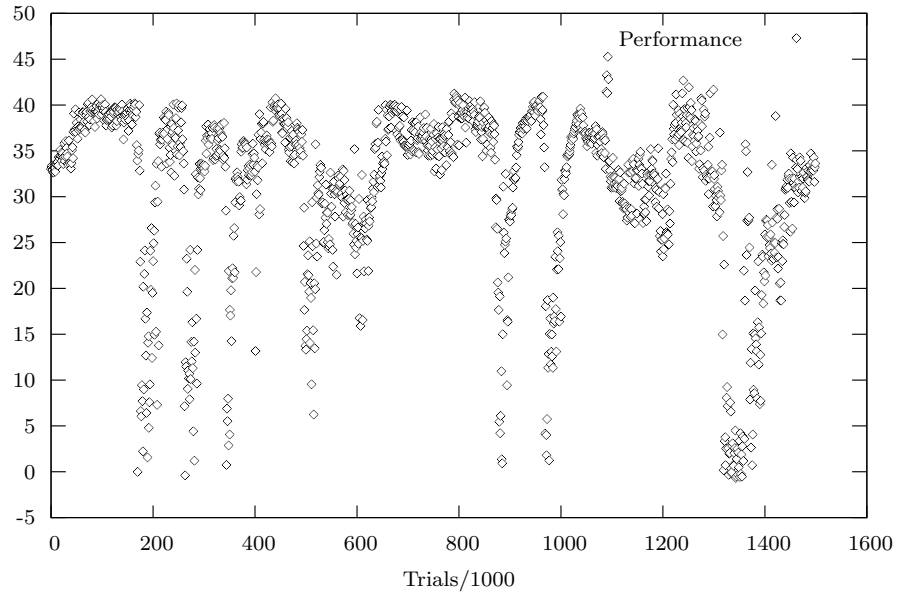


Figure 7.5: Instabilities during learning of a 3-segment swimmer with a 30-neuron feedforward neural network. Performance is measured as the distance swum in 50 seconds from a fixed starting position.

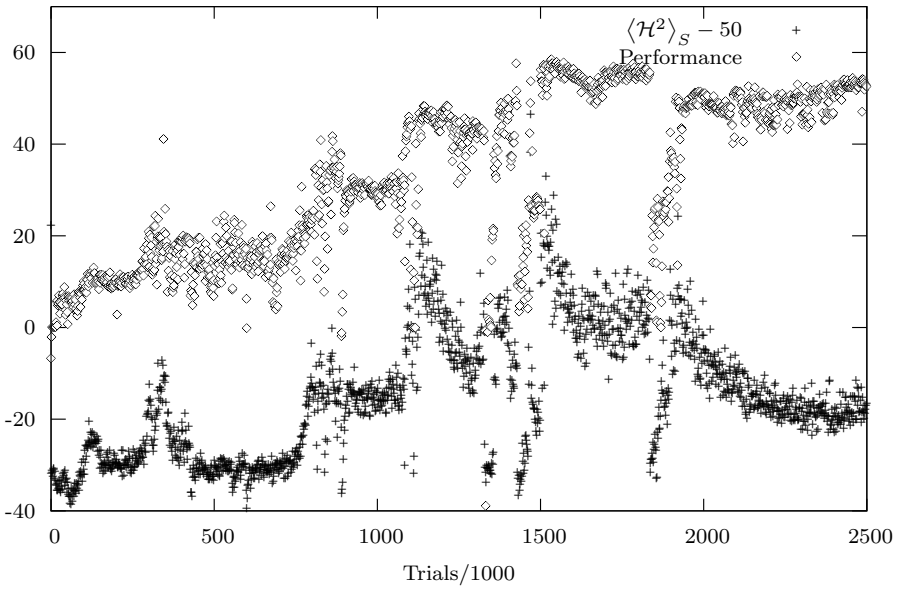


Figure 7.6: Learning of a 3-segment swimmer with a 60-neuron feedforward neural network.

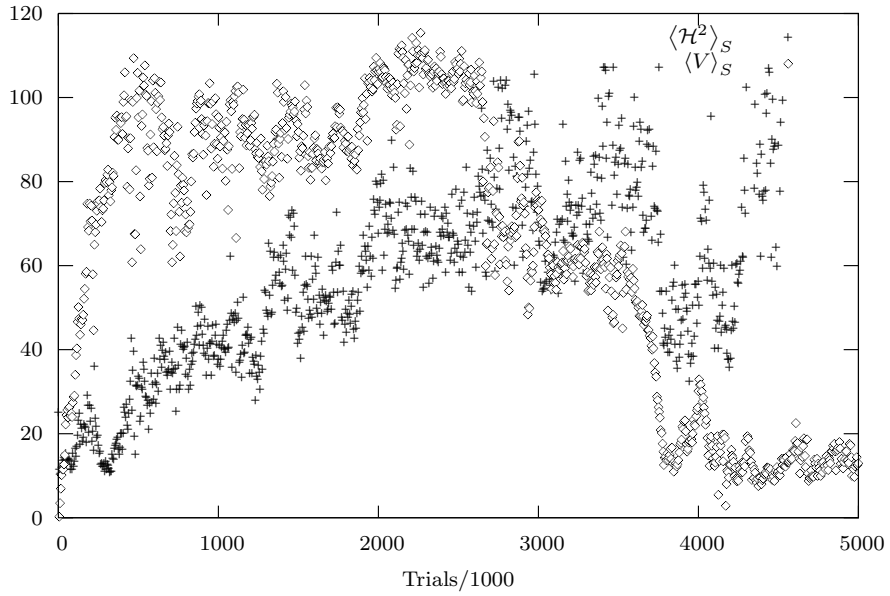


Figure 7.7: Learning of a 5-segment swimmer with a 60-neuron feedforward neural network.

Conclusion

Conclusion

In this thesis, we have presented a study of reinforcement learning using neural networks. Classical techniques of dynamic programming, neural networks and continuous neuro-dynamic programming have been reviewed, and some refinements of these methods have been proposed. Finally, these algorithms have been applied successfully to difficult motor-control problems.

Many original results have been presented in this dissertation: the ∂^* notation and the differential back-propagation algorithm (Appendix A), the K1999 path-optimization algorithm (Appendix C), and the second order equation (1.2.6) for finite difference methods. Besides these, the main original contributions of this work are numerical integration methods to deal with state and action discontinuities in TD(λ), and many original experimental results on a large variety of motor tasks. The most significant of these contributions is probably the success of the swimmer experiments (Chapter 7), which demonstrates that the combination of continuous model-based reinforcement learning with feedforward neural networks can handle motor-control problems that are much more complex than those usually solved with similar methods.

Each of these contributions also opens questions and directions for future research:

- The numerical integration method could certainly be improved significantly. In particular, the idea of using second order information about the value function to estimate the Filippov control might be extended to control spaces with more than one dimension.

- The vario-η algorithm should be compared empirically to other classical second-order methods.

- Many new interesting experiments could be run with swimmers. In particular, the reasons for the observed instabilities should be investigated. Methods based on the actor-critic approach seem to have better convergence properties [69], and should be tried on this problem. Also, bigger swimmers could learn to swim. Beyond swimmers, the method

used might as well be able to build controllers for much more complex problems.

Besides these direct extensions, another very important question to explore is the possibility to apply feedforward neural networks outside the restricted framework of simulated model-based motor-control. In particular, experiments indicate that feedforward neural networks require many more learning episodes than linear function approximators. This requirement might be a major obstacle in situations where learning data is costly to obtain, which is the case when it is collected in real time (like in robotics experiments), or when selecting actions entails a lot of computation (like in computer chess [12]). It was not the case in swimmer experiments, or with Tesauro's backgammon player, because it was possible to produce, at no cost, as much training data as needed.

The key problem here is locality. Very often, linear function approximators are preferred, because their good locality allows them to perform incremental learning efficiently, whereas feedforward networks tend to unlearn past experience as new training data is processed. The performance of the swimmers obtained in this thesis, however, clearly indicates that feedforward networks can solve problems that are orders of magnitude more complex than what linear function approximators can handle. So, it would be natural to try to combine the qualities of these two approximation schemes.

Creating a function approximator that would have both the locality of linear function approximators, and the generalization capabilities of feedforward neural networks seems very difficult. Weaver *et al.* [78] proposed a special learning algorithm to prevent unlearning. Its efficiency on high-dimensional reinforcement learning tasks remains to be demonstrated, but it might be an interesting research direction.

Another possibility to make a better use of scarce learning data would consist in complementing the reinforcement learning algorithm by some form of long term memory that stores it. After some time, the reinforcement learning algorithm could recall this stored data to check that it has not been "unlearnt" by the feedforward neural network. A major difficulty of this approach is that it would require some kind of off-policy TD(λ) learning, because the learning algorithm would observe trajectories that were generated with a different value function.

Appendices

Appendix A

Backpropagation

This appendix gathers results related to backpropagation.

A.1 Notations

A.1.1 Feedforward Neural Networks

Let us consider a n-neuron feedforward neural network defined by its connection matrix $W = (w_{ij})$ and its output functions σ_i. We will suppose neurons are ordered so that W is strictly triangular, that is $\forall j \geq i \; w_{ij} = 0$, and that output functions are differentiable twice. Let $\vec{x}$, $\vec{a}$ and $\vec{y}$ be respectively the input, the activation and the output vectors, each of them having n components. The dynamics of the neural network is described by:

$$\vec{a} = W\vec{y} \tag{A.1.1}$$

$$y_i = x_i + \sigma_i(a_i). \tag{A.1.2}$$

Since W is triangular, the output of the network can be simply obtained by computing a_1, y_1, a_2, y_2, $\ldots a_n$, y_n in sequence.

Having n inputs and n outputs for a n-neuron neural network might look a little strange. Usually, only a few of the first neurons are said to be "input" neurons, a few of the last neurons are "output" neurons, and the others are "hidden" neurons. Here, we will consider that all neurons can play all these roles at the same time. This formalism has the advantage of being both more simple and more general than the traditional input-hidden-output layered architecture[1].

[1] This kind of formalism is attributed to Fernando Pineda by Pearlmutter [52].

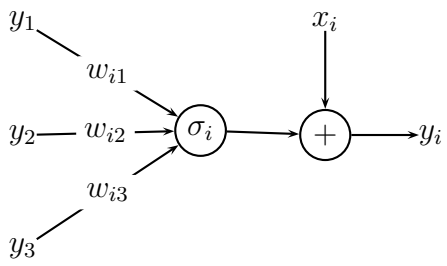


Figure A.1: Neuron in a feedforward network: $y_i = x_i + \sigma_i\left(\sum_{j<i} w_{ij}y_j\right)$

A.1.2 The ∂^* Notation

A special notation will be used to distinguish between two forms of partial derivative. For any variable v, the usual notation $\partial/\partial v$ means a derivative with respect to v, with all other variables being constant. On the other hand, $\partial/\partial^* v$ means a partial derivative with respect to v with all the activations and output of the neural network varying according to (A.1.1) and (A.1.2). v can be an input, an output, an activation or a weight of the network. For instance, for an error function

$$E(\vec{y}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - z_i)^2$$

the usual partial derivative is

$$\frac{\partial E}{\partial y_i} = y_i - z_i,$$

whereas the special partial derivative noted $\partial E/\partial^* y_i$ is also the partial derivative of E with respect to y_i but with all $(y_j)_{j>i}$ varying according to (A.1.1) and (A.1.2).

Why introduce a new notation? In fact, it would be possible to do without it. The problem is that doing rigorous derivations with the usual partial derivative notation would require to define a large number of complicated functions such as $E_{y_i}(y_1, \ldots, y_i)$, and write $\partial E_{y_i}/\partial y_i$ instead of $\partial E/\partial^* y_i$, which would be very tedious. Many of the well-known publications about backpropagation algorithms either have to introduce heavy notations like this, or use the usual partial derivative everywhere, which may be ambiguous. Smith [64] uses d instead of ∂^*, which may be confusing since the derivative is actually a partial derivative. The ∂^* notation allows to make calculations that are both simple and rigorous.

A.2 Computing $\partial E/\partial^* \vec{w}$

This section describes the backpropagation algorithm [36] [56] that computes, for a given input $\vec{x}$, the gradient with respect to weights of an error function $E(\vec{y})$.

$$\frac{\partial E}{\partial^* w_{ij}} = \frac{\partial E}{\partial^* a_i} \frac{\partial a_i}{\partial w_{ij}}. \tag{A.2.1}$$

Let us define

$$\delta_i \hat{=} \frac{\partial E}{\partial^* a_i},$$

and

$$\alpha_i \hat{=} \frac{\partial E}{\partial^* y_i}.$$

(A.2.1) becomes

$$\frac{\partial E}{\partial^* w_{ij}} = \delta_i y_j. \tag{A.2.2}$$

δ_i can be evaluated with

$$\delta_i = \frac{\partial E}{\partial^* y_i} \frac{\partial y_i}{\partial a_i} = \alpha_i \sigma_i'(a_i),$$

and

$$\begin{aligned} \alpha_i = \frac{\partial E}{\partial^* y_i} &= \frac{\partial E}{\partial y_i} + \sum_{j=i+1}^{n} \frac{\partial E}{\partial^* a_j} \frac{\partial a_j}{\partial y_i} \\ &= \frac{\partial E}{\partial y_i} + \sum_{j=i+1}^{n} w_{ji} \delta_j. \end{aligned}$$

These equations allow to compute α_n, δ_n, α_{n-1}, δ_{n-1}, ..., α_1, δ_1 in sequence. Then, equation (A.2.2) gives the gradient of the error with respect to weights.

A.3 Computing $\partial \vec{y}/\partial^* \vec{x}$

By derivating (A.1.2) and using (A.1.1) we get:

$$\frac{\partial y_i}{\partial^* x_k} = \frac{\partial x_i}{\partial x_k} + \sigma_i'(a_i) \sum_{j=k}^{i-1} w_{ij} \frac{\partial y_j}{\partial^* x_k}.$$

This is formally equivalent to the output of a "derivative network" that has the same connection weights W, the input of which is $\partial \vec{x}/\partial x_k$ and the output

functions of which are multiplications by $\sigma_i'(a_i)$. Let us call its activation vector $\vec{a}_k'$. (A.1.1) and (A.1.2) for this network are

$$\vec{a}_k' = W\frac{\partial \vec{y}}{\partial^* x_k} \tag{A.3.1}$$

$$\frac{\partial y_i}{\partial^* x_k} = \frac{\partial x_i}{\partial x_k} + \sigma_i'(a_i)a_{ki}'. \tag{A.3.2}$$

Strangely, this algorithm is sometimes called "computing the Jacobian of the output by backpropagation". In fact, propagation is forward, not backward.

A.4 Differential Backpropagation

This section describes an algorithm to compute, for a feedforward neural-network, the gradient with respect to weights of an error function that depends not only on the output of the network, like in the classical backpropagation, but also on its derivative with respect to inputs. This algorithm can be used to solve partial differential equations with a neural network. Some similar algorithms have been published [45], but, as far as I know, this general formulation for feedforward neural networks is an original result.

Let us suppose that for a given input $\vec{x}$ we want the network to minimize an error function E that depends on the output of the network and its derivatives with respect to inputs:

$$E\left(\vec{y}, \frac{\partial \vec{y}}{\partial^* \vec{x}}\right).$$

Let us calculate its gradient with respect to weights:

$$\frac{\partial E}{\partial^* w_{ij}} = \frac{\partial E}{\partial^* a_i}\frac{\partial a_i}{\partial w_{ij}} + \sum_{k=1}^{n} \frac{\partial E}{\partial^*\left(\frac{\partial a_i}{\partial^* x_k}\right)}\frac{\partial\left(\frac{\partial a_i}{\partial^* x_k}\right)}{\partial w_{ij}}. \tag{A.4.1}$$

If we define

$$\delta_i \mathrel{\hat{=}} \frac{\partial E}{\partial^* a_i}$$

and

$$\delta_{ki}' \mathrel{\hat{=}} \frac{\partial E}{\partial^*\left(\frac{\partial a_i}{\partial^* x_k}\right)},$$

then (A.4.1) becomes

$$\frac{\partial E}{\partial^* w_{ij}} = \delta_i y_j + \sum_{k=1}^{n} \delta_{ki}' \frac{\partial y_j}{\partial^* x_k}. \tag{A.4.2}$$

Let us now find recurrence relations to compute δ_i and δ'_{ki} by backpropagation:

$$\delta_i = \frac{\partial E}{\partial^* y_i}\frac{\partial y_i}{\partial a_i} + \sum_{k=1}^{n} \frac{\partial E}{\partial^* \left(\frac{\partial y_i}{\partial^* x_k}\right)} \frac{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)}{\partial a_i}. \tag{A.4.3}$$

Decomposing the first part of the first term gives

$$\frac{\partial E}{\partial^* y_i} = \frac{\partial E}{\partial y_i} + \sum_{j=i+1}^{n} \frac{\partial E}{\partial^* a_j}\frac{\partial a_j}{\partial y_j} \tag{A.4.4}$$

$$= \frac{\partial E}{\partial y_i} + \sum_{j=i+1}^{n} w_{ji}\delta_j. \tag{A.4.5}$$

Decomposing the second term (let us call it α'_{ki}):

$$\alpha'_{ki} = \frac{\partial E}{\partial^* \left(\frac{\partial y_i}{\partial^* x_k}\right)} = \frac{\partial E}{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)} + \sum_{j=i+1}^{n} \frac{\partial E}{\partial^* \left(\frac{\partial a_j}{\partial^* x_k}\right)} \frac{\partial \left(\frac{\partial a_j}{\partial^* x_k}\right)}{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)} \tag{A.4.6}$$

$$= \frac{\partial E}{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)} + \sum_{j=i+1}^{n} w_{ji}\delta'_{kj}. \tag{A.4.7}$$

By substituting (A.4.5) and (A.4.7) into (A.4.3) we get

$$\delta_i = \sigma'_i(a_i)\left(\frac{\partial E}{\partial y_i} + \sum_{j=i+1}^{n} w_{ji}\delta_j\right) + \sigma''_i(a_i)\sum_{k=1}^{n} a'_{ki}\alpha'_{ki}. \tag{A.4.8}$$

The recurrence relation for δ'_{ki} can be obtained by applying a similar chain rule:

$$\delta'_{ki} = \frac{\partial E}{\partial^* \left(\frac{\partial y_i}{\partial^* x_k}\right)} \frac{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)}{\partial \left(\frac{\partial a_i}{\partial^* x_k}\right)} \tag{A.4.9}$$

$$= \sigma'_i(a_i)\left(\frac{\partial E}{\partial \left(\frac{\partial y_i}{\partial^* x_k}\right)} + \sum_{j=i+1}^{n} w_{ji}\delta'_{kj}\right) \tag{A.4.10}$$

$$= \sigma'_i(a_i)\alpha'_{ki}. \tag{A.4.11}$$

To summarize, backpropagation to evaluate the gradient of an error function that depends on the derivative of outputs with respect to inputs can be performed in 5 steps:

1. Compute the output of the neural network using usual forward propagation (equations (A.1.1) and (A.1.2)).

2. Compute the output and activation of the derivative neural networks by forward propagation (equations (A.3.1) and (A.3.2)).

3. Compute the δ'_{ki}'s and α'_{ki}'s using (A.4.11) and (A.4.7), which is usual backpropagation on the derivative networks.

4. Compute the δ_i's using (A.4.8) which is a special form of backpropagation on the main network.

5. Get the gradient of the error with (A.4.2)

Appendix B

Optimal-Control Problems

This appendix contains technical details about problems mentioned in the text of this document. All numerical values are given in SI units.

B.1 Pendulum

B.1.1 Variables and Parameters

State variables:

- θ, angular position ($\theta \in [-\pi, \pi]$, circular)
- $\dot{\theta}$, angular velocity ($|\dot{\theta}| < \dot{\theta}_{max}$, bounded)

Control variables:

- u, torque applied ($|u| < u_{max}$)

System parameters:

- g, gravity acceleration

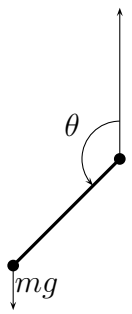


Figure B.1: Pendulum

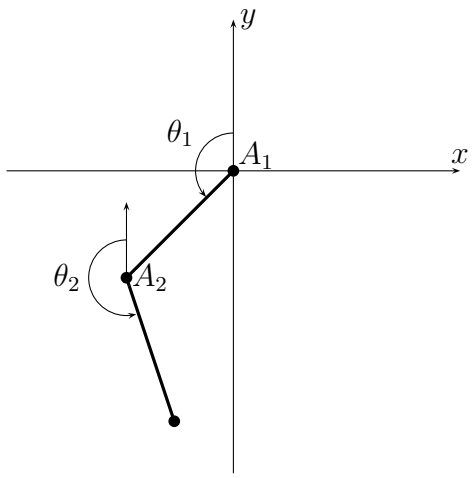


Figure B.2: Acrobot

- m, mass of the pendulum
- l, length of the pendulum
- μ, coefficient of friction

B.1.2 System Dynamics

$$\ddot{\theta} = \frac{1}{ml^2}(-\mu\dot{\theta} + mgl\sin\theta + u)$$

B.1.3 Reward

$r(\vec{x}, \vec{u}) = \cos\theta$

B.1.4 Numerical Values

$g = 9.81$, $m = 1$, $l = 1$, $\mu = 0.01$, $u_{max} = 5$, $\omega_{max} = 10$, $s = 1$.

B.2 Acrobot

B.2.1 Variables and Parameters

State variables:

- θ_1: angle of body with respect to the vertical axis

- θ_2: angle of legs with respect to the vertical axis
- $\dot{\theta_1}, \dot{\theta_2}$: their derivatives with respect to time

Control variable:

- u, torque applied to legs ($|u| < u_{max}$)

Problem parameters:

- g: gravity acceleration
- m_1: mass of acrobot body
- m_2: mass of acrobot legs
- l_1: half-length of body
- l_2: half-length of legs
- μ_1: friction coefficient of body
- μ_2: friction coefficient of legs

B.2.2 System Dynamics

Let $\vec{p_i}$, $\vec{v_i}$ and $\vec{a_i}$ be the vectors defined for i in $\{1, 2\}$ by

$$\vec{p_i} = \begin{pmatrix} -\sin\theta_i \\ \cos\theta_i \end{pmatrix},$$
$$\vec{v_i} = \dot{\vec{p}_i} = \begin{pmatrix} -\dot{\theta}_i \cos\theta_i \\ -\dot{\theta}_i \sin\theta_i \end{pmatrix},$$
$$\vec{a_i} = \dot{\vec{v}_i} = \begin{pmatrix} \dot{\theta}_i^2 \sin\theta_i - \ddot{\theta}_i \cos\theta_i \\ \dot{\theta}_i^2 \cos\theta_i + \ddot{\theta}_i \sin\theta_i \end{pmatrix}.$$

Let $\vec{f_1}$ be the force applied to hands and $\vec{f_2}$ the force applied by the body to legs. Let G_1 and G_2 be the centers of mass of the two body parts. The laws of mechanics can be written as

$$\begin{aligned} \vec{f_1} - \vec{f_2} + m_1\vec{g} &= m_1 l_1 \vec{a}_1, \\ \vec{f_2} - m_2\vec{g} &= m_2(2l_1\vec{a}_1 + l_2\vec{a}_2), \\ \frac{l_1^2}{3} m_1 \ddot{\theta}_1 &= \det(\overrightarrow{G_1A_1}, \vec{f_1} + \vec{f_2}) - u - \mu_1\dot{\theta}_1, \\ \frac{l_2^2}{3} m_2 \ddot{\theta}_2 &= \det(\overrightarrow{G_2A_2}, \vec{f_2}) + u - \mu_2\dot{\theta}_2. \end{aligned}$$

Since

$$\begin{aligned}
\det(\overrightarrow{G_1A_1}, \vec{a}_1) &= -l_1\ddot{\theta}_1 \\
\det(\overrightarrow{G_2A_2}, \vec{a}_2) &= -l_2\ddot{\theta}_2 \\
\det(\overrightarrow{G_1A_1}, \vec{a}_2) &= l_1\big(\dot{\theta}_2^2 \sin(\theta_2 - \theta_1) - \ddot{\theta}_2 \cos(\theta_1 - \theta_2)\big) \\
\det(\overrightarrow{G_2A_2}, \vec{a}_1) &= l_2\big(\dot{\theta}_1^2 \sin(\theta_1 - \theta_2) - \ddot{\theta}_1 \cos(\theta_2 - \theta_1)\big)
\end{aligned}$$

these equations can be simplified into

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

with

$$\begin{cases}
a_{11} = (\frac{4}{3}m_1 + 4m_2)l_1^2, \\
a_{22} = \frac{4}{3}m_2 l_2^2, \\
a_{12} = a_{21} = 2m_2 l_1 l_2 \cos(\theta_1 - \theta_2), \\
b_1 = 2m_2 l_2 l_1 \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) + (m_1 + 2m_2) l_1 g \sin\theta_1 - \mu_1\dot{\theta}_1 - u, \\
b_2 = 2m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + m_2 l_2 g \sin\theta_2 - \mu_2\dot{\theta}_2 + u.
\end{cases}$$

B.2.3 Reward

$r(\vec{x}, \vec{u}) = l_1 \cos\theta_1 + l_2 \cos\theta_2$

B.2.4 Numerical Values

$m_1 = m_2 = 1$, $l_1 = l_2 = 0.5$, $\mu_1 = \mu_2 = 0.05$, $u_{max} = 2$

B.3 Cart-Pole

B.3.1 Variables and Parameters

State variables:

- x: cart position ($|x| < L$),
- θ: pole angle with respect to a vertical axis,
- $\dot{x}, \dot{\theta}$: their derivatives with respect to time.

Control variable:

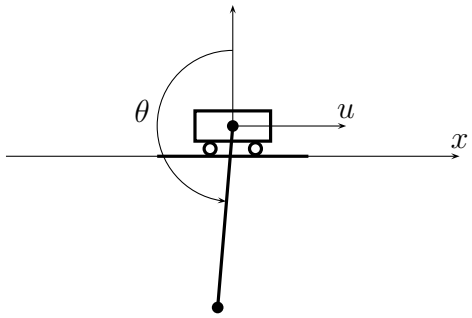


Figure B.3: Cart-Pole

- u, horizontal force applied to the cart ($|u| < u_{max}$).

Problem parameters:

- L: half-length of the track,
- g: gravity acceleration,
- m_c: mass of the cart,
- m_p: mass of the pole,
- l: half-length of the pole,
- μ_c: coefficient of friction of cart on track,
- μ_p: coefficient of friction of pivot.

B.3.2 System Dynamics

The position of a point of the pole at distance λ from the articulation is

$$\begin{pmatrix} x - \lambda \sin\theta \\ \lambda \cos\theta \end{pmatrix}.$$

Thus its velocity is

$$\begin{pmatrix} \dot{x} - \lambda\dot{\theta} \cos\theta \\ -\lambda\dot{\theta} \sin\theta \end{pmatrix},$$

and its acceleration is

$$\begin{pmatrix} \ddot{x} + \lambda(\dot{\theta}^2 \sin\theta - \ddot{\theta} \cos\theta) \\ -\lambda(\dot{\theta}^2 \cos\theta + \ddot{\theta} \sin\theta) \end{pmatrix}.$$

The "dynamic resultant" theorem for the (cart + pole) system can be written as:

$$\begin{aligned} u - \mu_c \operatorname{sign}(\dot{x}) &= m_c\ddot{x} + \int_0^{2l} \big(\ddot{x} + \lambda(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta)\big)\frac{m_p}{2l}d\lambda \\ &= (m_c + m_p)\ddot{x} + l m_p(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta) \end{aligned} \tag{B.3.1}$$

Let G be the center of mass of the pole and J_G be the moment of inertia of the pole at G.

$$J_G = \int_{-l}^{+l} \lambda^2 \frac{m_p}{2l} d\lambda = \frac{l^2}{3} m_p.$$

Let us call $\vec{f}$ the force applied by the cart on the pole and C the articulation point where this force is applied. The "dynamic resultant" theorem, applied to the pole only, gives:

$$m_p\ddot{G} = m_p\vec{g} + \vec{f},$$

hence

$$\vec{f} = m_p\ddot{G} - m_p\vec{g}.$$

The "resultant moment" theorem gives:

$$J_G\ddot{\theta} = det(\overrightarrow{GC}, \vec{f}) - \mu_p\dot{\theta},$$

hence

$$\frac{l^2}{3}\ddot{\theta} = \begin{vmatrix} l\sin\theta & \ddot{x} + l(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta) \\ -l\cos\theta & g - l(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta) \end{vmatrix} - \frac{\mu_p\dot{\theta}}{m_p}$$

$$= -l^2\sin\theta(\dot{\theta}^2\cos\theta + \ddot{\theta}\sin\theta) + gl\sin\theta + l\cos\theta\big(\ddot{x} + l(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta)\big) - \frac{\mu_p\dot{\theta}}{m_p}$$

$$= -l^2\ddot{\theta}\underbrace{(\sin^2\theta + \cos^2\theta)}_{1} - l^2\dot{\theta}^2\underbrace{(\sin\theta\cos\theta - \sin\theta\cos\theta)}_{0} + gl\sin\theta + l\ddot{x}\cos\theta - \frac{\mu_p\dot{\theta}}{m_p}.$$

So

$$\frac{4}{3}l^2\ddot{\theta} = gl\sin\theta + l\ddot{x}\cos\theta - \frac{\mu_p\dot{\theta}}{m_p} \tag{B.3.2}$$

(B.3.1) and (B.3.2) give the set of linear equations to be solved:

$$\begin{pmatrix} \frac{4}{3}l & -\cos\theta \\ l m_p\cos\theta & -(m_c + m_p) \end{pmatrix} \begin{pmatrix} \ddot{\theta} \\ \ddot{x} \end{pmatrix} = \begin{pmatrix} g\sin\theta - \frac{\mu_p\dot{\theta}}{l m_p} \\ l m_p\dot{\theta}^2\sin\theta - u + \mu_c\operatorname{sign}(\dot{x}) \end{pmatrix}$$

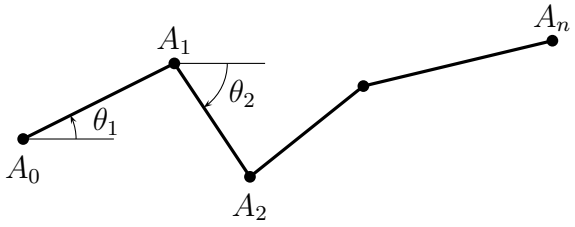


Figure B.4: Swimmer

B.3.3 Reward

$r(\vec{x}, \vec{u}) = \cos\theta$

B.3.4 Numerical Values

$L = 2.4$, $m_c = 1$, $m_p = 0.1$, $l = 0.5$, $g = 9.8$, $u_{max} = 10$, $\mu_c = 0.0005$, $\mu_p = 0.000002$

B.4 Swimmer

B.4.1 Variables and Parameters

State variables:

- A_0: position of first point
- θ_i: angle of part i with respect to the x axis
- $\dot{A}_0, \dot{\theta}_i$: their derivatives with respect to time

Control variables:

- $(u_i)_{i\in\{1...n-1\}}$, torques applied between body parts, constrained by $|u_i| < U_i$

Problem parameters:

- n: number of body parts
- m_i: mass of part i ($i \in \{1...n\}$)
- l_i: length of part i ($i \in \{1...n\}$)
- k: viscous-friction coefficient

B.4.2 Model of Viscous Friction

The model of viscous friction used consists in supposing that a small line of length $d\lambda$ with a normal vector $\vec{n}$ and moving at velocity $\vec{v}$ receives a small force of value $d\vec{F} = -k(\vec{v}\cdot\vec{n})\vec{n}d\lambda$. Let us use this law of friction to calculate the total force and moment applied to a moving part of the swimmer's body. Let us call M a point of this body part at distance λ from the center of mass $G_i = (A_{i-1} + A_i)/2$. $M = G_i + \lambda\vec{p}_i$ with

$$\vec{p}_i = \begin{pmatrix}\cos\theta_i \\ \sin\theta_i\end{pmatrix}.$$

Besides, the normal vector is

$$\vec{n}_i = \begin{pmatrix}-\sin\theta_i \\ \cos\theta_i\end{pmatrix}$$

and $\dot{\vec{p}}_i = \dot{\theta}_i\vec{n}_i$, so $\dot{M} = \dot{G}_i + \lambda\dot{\theta}_i\vec{n}_i$. The total force is equal to

$$\begin{aligned}\vec{F}_i &= \int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\dot{M}\cdot\vec{n}_i)\vec{n}_i d\lambda \\ &= \int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\dot{G}_i\cdot\vec{n}_i)\vec{n}_i d\lambda + \underbrace{\int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\lambda\dot{\theta}_i\vec{n}_i\cdot\vec{n}_i)\vec{n}_i d\lambda}_{0} \\ &= -kl_i(\dot{G}_i\cdot\vec{n}_i)\vec{n}_i.\end{aligned}$$

Let us now calculate the total moment at G_i:

$$\begin{aligned}\mathcal{M}_i &= \int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} \det\left(\lambda\vec{p}_i, -k(\dot{M}\cdot\vec{n}_i)\vec{n}_i d\lambda\right) \\ &= \int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\dot{M}\cdot\vec{n}_i)\lambda d\lambda \\ &= \underbrace{\int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\dot{G}_i\cdot\vec{n}_i)\lambda d\lambda}_{0} + \int_{-\frac{l_i}{2}}^{\frac{l_i}{2}} -k(\lambda\dot{\theta}_i\vec{n}_i\cdot\vec{n}_i)\lambda d\lambda \\ &= -k\dot{\theta}_i\frac{l_i^3}{12}\end{aligned}$$

B.4.3 System Dynamics

Let $\vec{f_i}$ be the force applied by part $i+1$ to part i.

$$\forall i \in \{1, \dots, n\} \quad -\vec{f_{i+1}} + \vec{f_i} + \vec{F_i} = m_i \ddot{\vec{G}}_i$$

These equations allow to express the $\vec{f_i}$'s as a function of state variables:

$$\begin{aligned} \vec{f_0} &= \vec{0} \\ \forall i \in \{1, \dots, n\} \quad \vec{f_i} &= \vec{f_{i-1}} - \vec{F_i} + m_i \ddot{\vec{G}}_i \end{aligned}$$

We end up with a set of $n+2$ linear equations with $n+2$ unknowns:

$$\begin{cases} \vec{f_n} = \vec{0}, \\ m_i \dfrac{l_i}{12} \ddot{\theta}_i = \det(\overrightarrow{G_i A_i}, \vec{f_i} + \vec{f_{i-1}}) + \mathcal{M}_i - u_i + u_{i-1}. \end{cases}$$

Unknowns are $\ddot{\vec{G}}_0$ (or any $\ddot{\vec{G}}_i$) and all the $\ddot{\theta}_i$'s. Solving this set of equations gives state dynamics.

B.4.4 Reward

$r(\vec{x}, \vec{u}) = \dot{G}_x$ (G_x is the abscissa of the center of mass)

B.4.5 Numerical Values

$m_i = 1$, $l_i = 1$, $k = 10$, $U_i = 5$

Appendix C

The K1999 Path-Optimization Algorithm

This appendix is a description of the path-optimization algorithm that is used in the K1999 car driver of the Robot Auto-Racing Simulator [71]. I wrote this very informal document to explain to other competitors how my car works, so it is probably not as formal as a serious research paper should be. In particular, no comparison is made with classical methods. This appendix is simply meant as an illustration of a method that is much more efficient than reinforcement learning. K1999 won the 2000 and 2001 formula one seasons.

C.1 Basic Principle

C.1.1 Path

The path is approximated by a sequence of points $(\vec{x}_i)_{1\leq i\leq n}$. The curvature c_i of the track at each point $\vec{x}_i$ is computed as the inverse of the radius of the circumscribed circle for points $\vec{x}_{i-1}$, $\vec{x}_i$ and $\vec{x}_{i+1}$. The formula is:

$$c_i = \frac{2\det(\vec{x}_{i+1}-\vec{x}_i, \vec{x}_{i-1}-\vec{x}_i)}{\|\vec{x}_{i+1}-\vec{x}_i\|\|\vec{x}_{i-1}-\vec{x}_i\|\|\vec{x}_{i+1}-\vec{x}_{i-1}\|} \tag{C.1.1}$$

This curvature is positive for curves to the left and negative for curves to the right. The $(\vec{x}_i)_{1\leq i\leq n}$ points are initially set to be right in the middle of the track. The method consists in slowly modifying this path by repeating algorithm C.1 several times.

When (If ?) this algorithm has converged then the curvature of the path varies linearly except at points where it touches one side of the track. Note that on the actual implementation, each $\vec{x}_i$ is constrained to stay on a given

Algorithm C.1 Basic algorithm

for $i = 1$ to n **do**
  $c_1 \leftarrow c_{i-1}$
  $c_2 \leftarrow c_{i+1}$
  set $\vec{x}_i$ at equal distance to $\vec{x}_{i+1}$ and $\vec{x}_{i-1}$ so that $c_i = \frac{1}{2}(c_1 + c_2)$
  if $\vec{x}_i$ is out of the track **then**
    Move $\vec{x}_i$ back onto the track
  end if
end for

line that crosses the track. As a consequence, $\vec{x}_i$ can not be set "at equal distance to $\vec{x}_{i+1}$ and $\vec{x}_{i-1}$". $c_i = \frac{1}{2}(c_1 + c_2)$ is replaced by a sum weighted by distance.

C.1.2 Speed Profile

Once the path has been computed, the target speed v_i all along this path can be obtained by supposing that acceleration is limited by tyre grip to having a norm inferior to a_0. This is done in two passes. First, s_i is initialized with an "instantaneous" maximum speed with algorithm C.2 (ε is a small enough positive constant). Then, algorithm C.3 consists in anticipating braking. This second pass is iterated a dozen times so that it reaches convergence. I am not particularaly happy by this technique, but it works. A more clever solution could be found that does not need iteration at all. But it does not matter much since it is very fast and gives good results.

Algorithm C.2 Pass 1

for $i = 1$ to n **do**
  if $|c_i| > \varepsilon$ **then**
    $s_i \leftarrow \sqrt{a_0/|c_i|}$
  else
    $s_i \leftarrow \sqrt{a_0/\varepsilon}$
  end if
end for

Algorithm C.3 Pass 2

for $i = n$ to 1 **do** {Note the descending order}
  $N \leftarrow \frac{1}{2}(c_{i-1} + c_i)v_i^2$ {normal acceleration}
  $T \leftarrow \sqrt{\max(0, a_0^2 - N^2)}$ {tangential Acceleration}
  $v \leftarrow \frac{1}{2}(v_i + v_{i-1})$
  $D \leftarrow kv^2$ {air drag}
  $t \leftarrow \|\vec{x}_i - \vec{x}_{i-1}\|/v$
  $v_{i-1} \leftarrow \min\big(s_{i-1}, v_i + t(T + D)\big)$
end for

C.2 Some Refinements

C.2.1 Converging Faster

To get a high accuracy on the path, a large number of points is necessary (1000 or more, typically). This makes convergence very slow, especially on wide tracks with very long curves like `wierd.trk`. The basic algorithm can be sped up considerably (a dozen times) by proceeding as shown in algorithm C.4. This will double the number of points at each iteration and converge much more rapidly. Note that the actual implementation I programmed in `k1999.cpp` is uselessy more complicated. This algorithm takes about 2-3 seconds of CPU time for a track on a 400MHz celeron PC.

Algorithm C.4 Speed optimization

Start with a small number of $\vec{x}_i$ (typically 2-3 per segment)
while there are not enough $\vec{x}_i$'s **do**
  Smooth the path using the basic algorithm
  Add new $\vec{x}_i$'s between the old ones
end while

C.2.2 Security Margins

The path-optimization algorithm must take into consideration the fact that the simulated car will not be able to follow the computed path exactly. It needs to take security margins with respect to the side of the track into consideration. These security margins should depend on the side of the track and the curvature of the path. For instance, if the curvature is positive (the path is turning left), it is much more dangerous to be close to the right side of the track than to the left side.

C.2.3 Non-linear Variation of Curvature

The statement

set $\vec{x}_i$ so that $c_i = \frac{1}{2}(c_1 + c_2)$

gives a linear variation of the curvature. But this choice was arbitrary and better laws of variation can be found. One cause of non-optimality of a linear variation of the curvature is that the car acceleration capabilities are not symmetrical. It can brake much more than it can accelerate. This means that accelerating early is often more important than braking late. Changing the law of variation to get an exponential variation with

if $|c_2| < |c_1|$ **then**
  set $\vec{x}_i$ so that $c_i = 0.51c_1 + 0.49c_2$
else
  set $\vec{x}_i$ so that $c_i = 0.50c_1 + 0.50c_2 n$
end if

can give a speed-up on some tracks.

C.2.4 Inflections

Another big cause of non-optimality is inflection in S curves. The curvature at such a point should change from $-c$ to $+c$ and not vary smoothly. Results can be significantly improved on some tracks by using the change described in algorithm C.5. This can be a big win on some tracks like `suzuka.trk` or `albrtprk.trk`.

C.2.5 Further Improvements by Gradient Descent

Although it is good, the path computed using this algorithm is not optimal. It is possible to get closer to optimality using algorithm C.6.

This algorithm is repeated all long the path for different values of i_0. It is more efficient to try values of i_0 which are multiple of a large power of two because of the technique described in §C.2.1. The details of the algorithm are in fact more complicated than this. Take a look at the source code for a more precise description. This technique takes a lot of computing power, but works very well. The typical gain is 1%. It was programmed as a quick hack and could be very probably improved further.

One of the main changes it causes in paths, is that they are more often at the limit of tyre grip (and not engine power), which is more efficient.

This technique also gives improvements that could be called "making short term sacrifices for a long term compensation". This effect is particulary spectacular on the Spa-Francorchamps track (`spa.trk`) where the computed

Algorithm C.5 Inflections

for $i = 1$ to n **do**
  $c_1 \leftarrow c_{i-1}$
  $c_2 \leftarrow c_{i+1}$
  if $c_1 c_2 < 0$ **then**
    $c_0 \leftarrow c_{i-2}$
    $c_3 \leftarrow c_{i+2}$
    if $c_0 c_1 > 0$ and $c_2 c_3 > 0$ **then**
      if $|c_1| < |c_2|$ and $|c_1| < |c_3|$ **then**
        $c_1 \leftarrow -c_1$
      else
        if $|c_2| < |c_1|$ and $|c_2| < |c_0|$ **then**
          $c_2 \leftarrow -c_2$
        end if
      end if
    end if
  end if
  set $\vec{x}_i$ at equal distance to $\vec{x}_{i+1}$ and $\vec{x}_{i-1}$ so that $c_i = \frac{1}{2}(c_1 + c_2)$
  if $\vec{x}_i$ is out of the track **then**
    Move $\vec{x}_i$ back onto the track
  end if
end for

Algorithm C.6 Rough and Badly Written Principle of Gradient Descent

Choose an index i_0 and a fixed value for $\vec{x}_{i_0}$
Run the standard algorithm without changing $\vec{x}_{i_0}$
Estimate the lap time for this path
Change $\vec{x}_{i_0}$ a little
Run the standard algorithm again and estimate the new lap time
while It is an improvement **do**
  Continue moving $\vec{x}_{i_0}$ in the same direction
end while

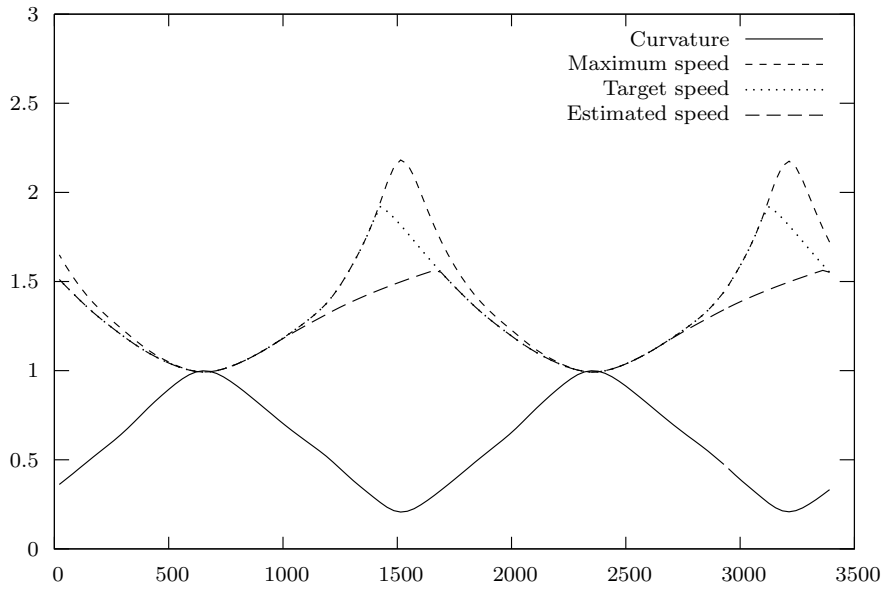


Figure C.1: Path data for oval2.trk before gradient descent

path looses contact with the inside the long curve before the final straight so that the car can accelerate earlier and enter the long straight with a higher speed.

C.3 Improvements Made in the 2001 Season

I had to improve my path optimization further in 2001 to strike back after Tim Foden's Dodger won the first races. His driver was inspired by some of the ideas I described in the previous sections, with some clever improvements that made his car faster than mine. I will not go into the details of these. One of the most important improvements in his path-optimization algorithm were a change that makes corners more circular, and a change that makes the inflection algorithm more stable.

C.3.1 Better Variation of Curvature

Both of these improvements by Tim Foden were ways to provide a better variation of curvature. I thought a little more about this and came to the conclusion that an important principle in finding the optimal path is that it should be at the limit of tire grip as long as possible. This is what guided the idea of the non-linear variation of curvature (Section C.2.3). Results

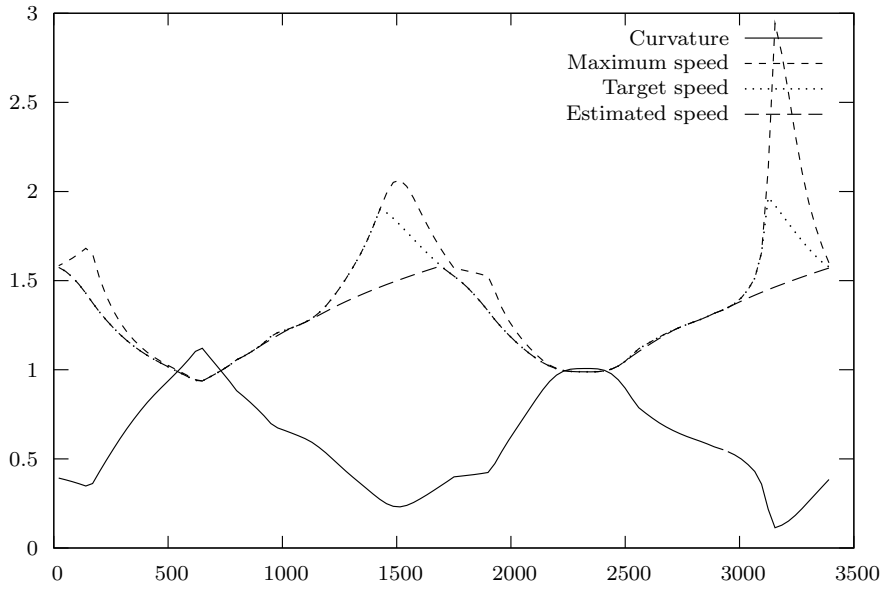


Figure C.2: Path data for oval2.trk after gradient descent

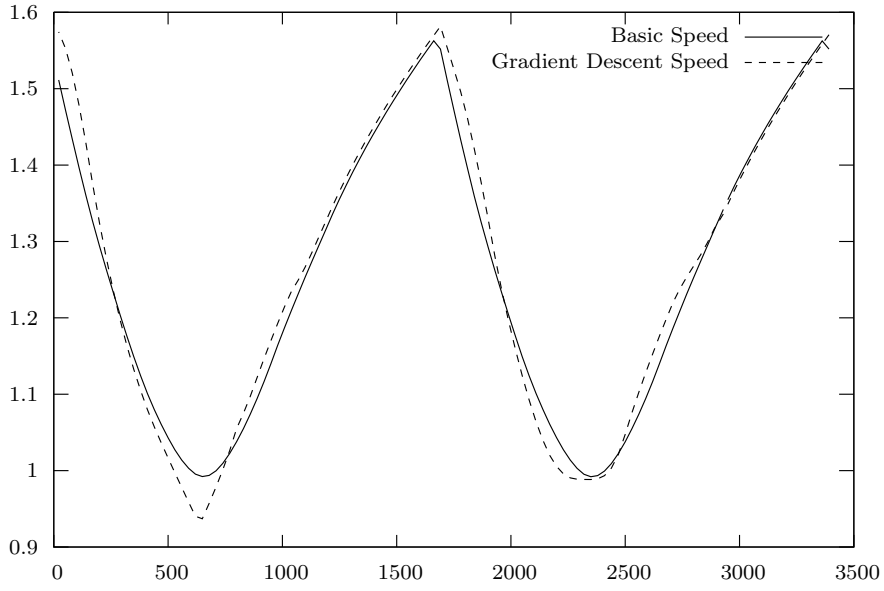


Figure C.3: Comparison before/after gradient descent

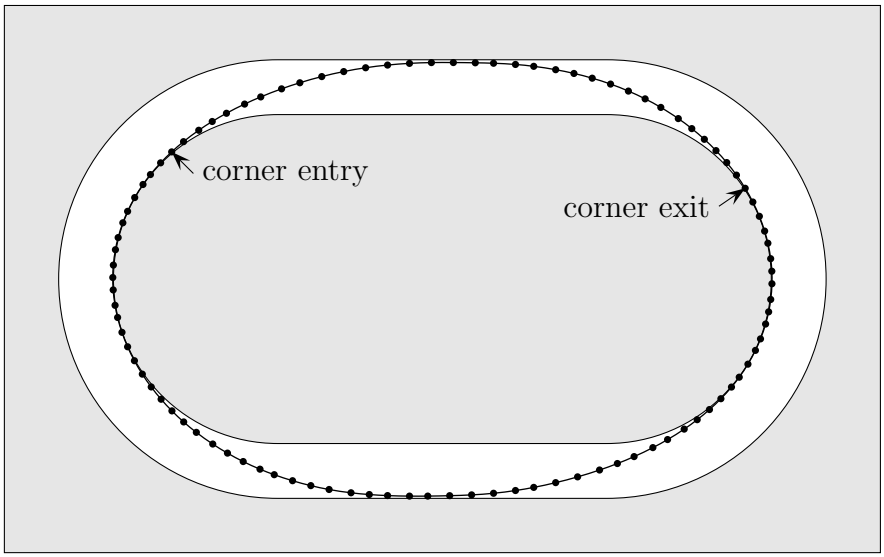


Figure C.4: Path for oval2.trk (anti-clockwise). The dotted line is the path after gradient descent. It is visibly asymmetric.

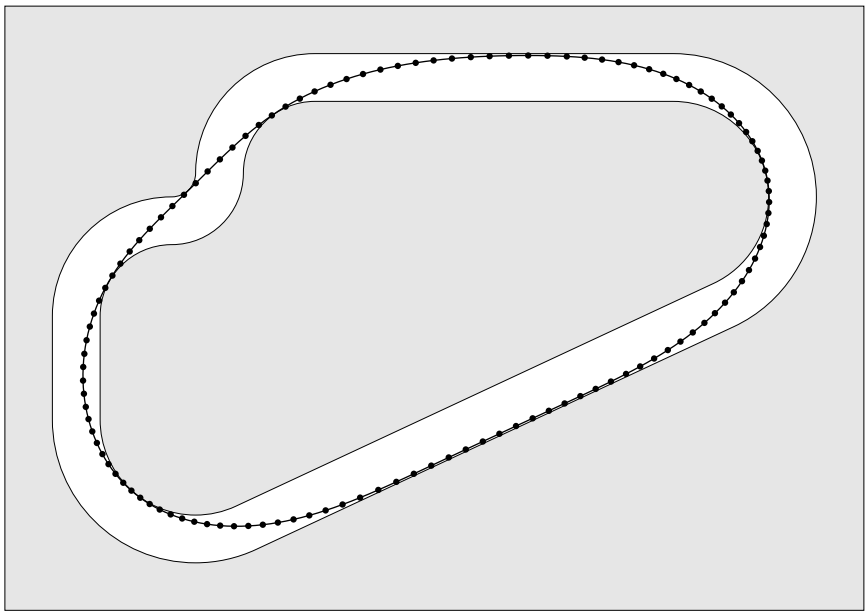


Figure C.5: Path for clkwis.trk

of gradient descent showed this too. Dodger's circular corners had also the consequence of letting the car be longer at the limit. So I modified the original algorithm into algorithm C.7. In this algorithm, v_i is an estimation of the velocity of the car at point $\vec{x}_i$ for the current shape of the path. Velocities are in feet per second.

Algorithm C.7 Better Variation of Curvature

for $i = 1$ to n **do**
  $c_1 \leftarrow c_{i-1}$
  $c_2 \leftarrow c_{i+1}$
  if $c_1 c_2 > 0$ **then**
    $v \leftarrow \sqrt{2a_0/(|c_1| + |c_2|)}$ {maximum velocity for current curvature}
    if $v > v_i + 8$ **then** {if estimated velocity inferior to limit}
      if $|c_1| < |c_2|$ **then** {try to bring it closer}
        $c_1 \leftarrow c_1 + 0.3 \times (c_2 - c_1)$
      else
        $c_2 \leftarrow c_2 + 0.3 \times (c_1 - c_2)$
      end if
    end if
  end if
  set $\vec{x}_i$ at equal distance to $\vec{x}_{i+1}$ and $\vec{x}_{i-1}$ so that $c_i = \frac{1}{2}(c_1 + c_2)$
  if $\vec{x}_i$ is out of the track **then**
    Move $\vec{x}_i$ back onto the track
  end if
end for

This change in the algorithm proved to be a significant improvement in terms of path optimization (about 1% on most tracks, much more on tracks with very fast curves like `indy500.trk` or `watglen.trk`). It is also a significant simplification since it handles both non-linear variation ideas and inflection ideas in a unified way.

Since the estimation of the velocity takes tyre grip and engine power into consideration, this algorithm generates different paths depending on these characteristics of the car (Figures C.7 and C.8).

C.3.2 Better Gradient Descent Algorithm

I first thought that after the improvement of the algorithm described previously, gradient descent would be less efficient than before. This is wrong. It still gets about 1% improvement on most tracks. I also improved it a bit. If you want to know more, then can take a look at the code or run the program

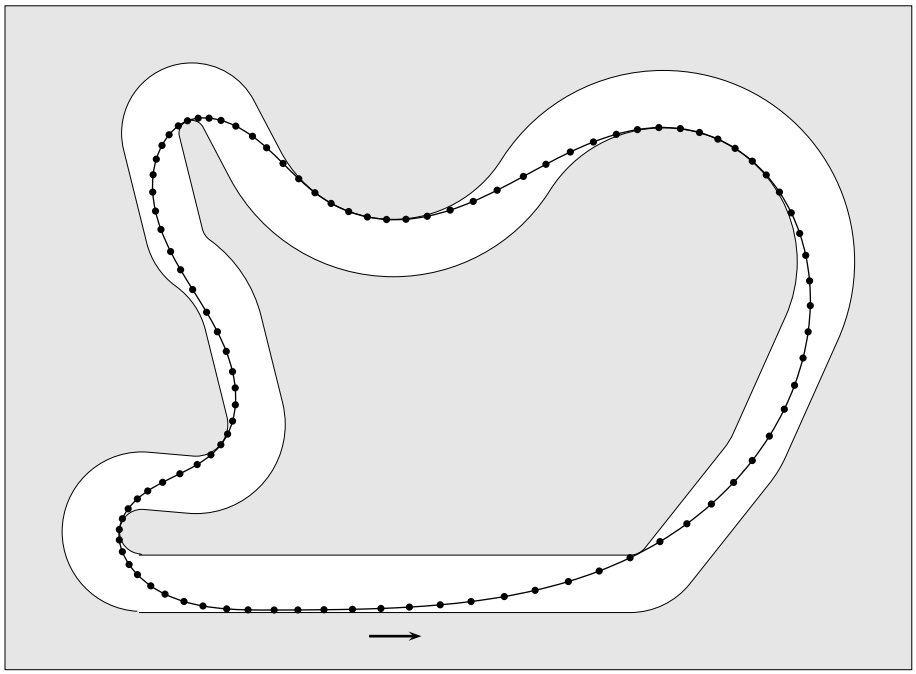


Figure C.6: linear curvature: 68.24 mph with -s1, 82.69 mph with -s2

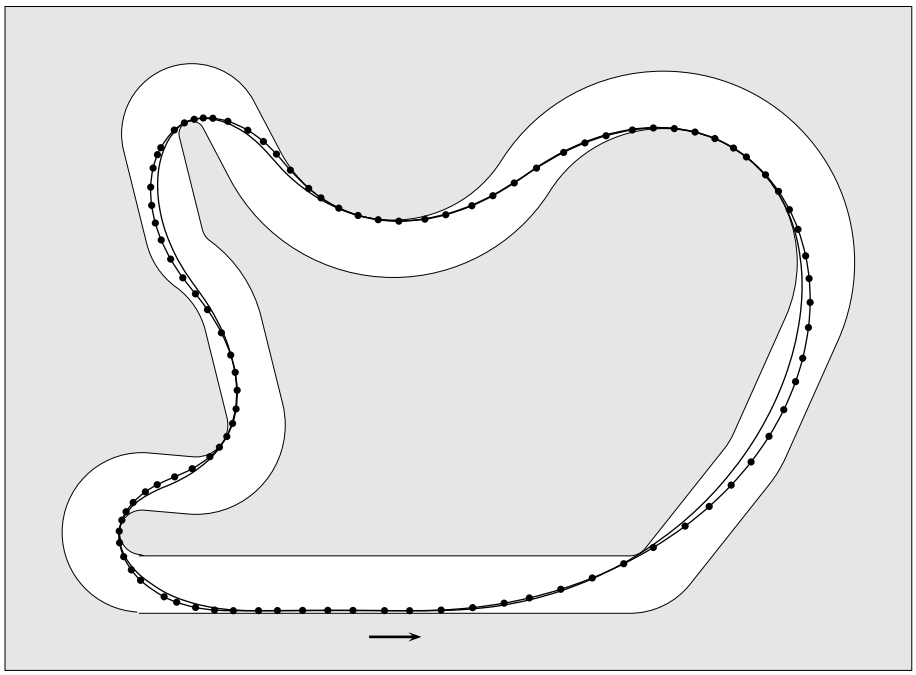


Figure C.7: Algorithm C.7 with standard tire grip (-s1) before (68.30 mph, dotted) and after (69.37 mph, not dotted) gradient descent

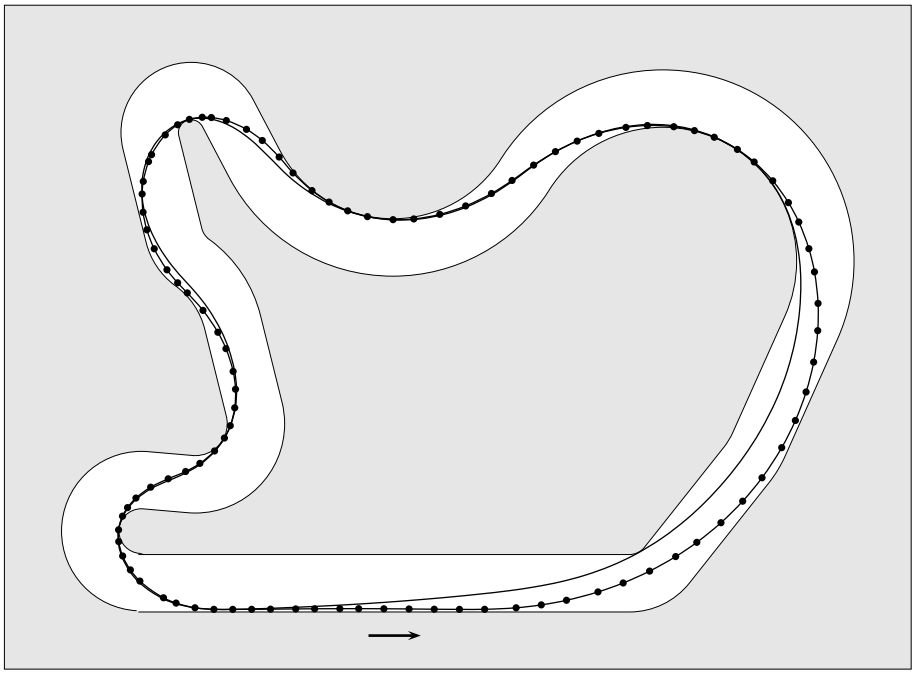


Figure C.8: Algorithm C.7 with more tire grip (-s2) before (82.87 mph, dotted) and after (84.29 mph, not dotted) gradient descent

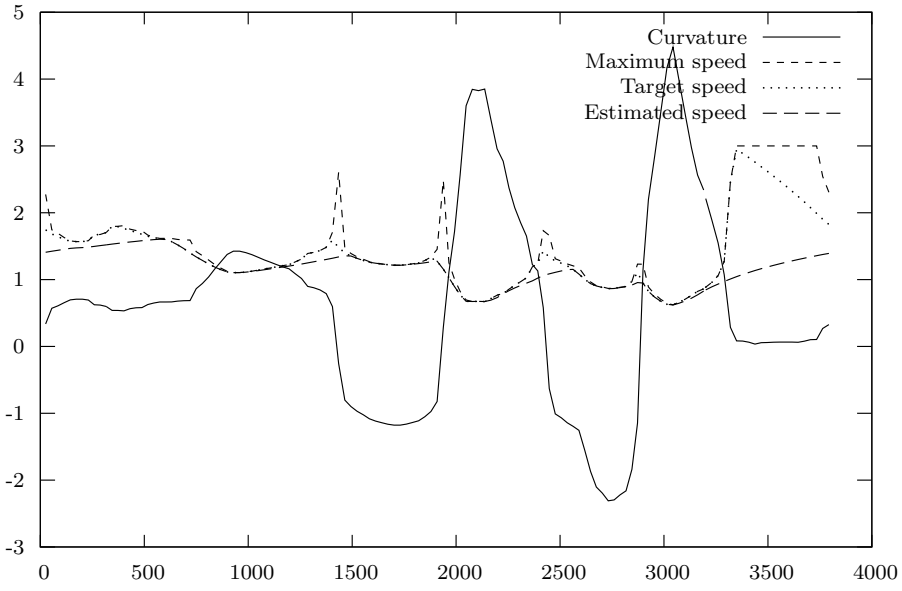


Figure C.9: Path data for stef2 after gradient descent (-s2)

to see how it works. Most of the time, the effect of gradient descent consists in making the path a little bit slower, but shorter (see Figures C.7 and C.8).

C.3.3 Other Improvements

A significant amount of additional speed was obtained with a better servo control system (up to 1% on some tracks). A better pit-stop strategy helped a little too. Passing was improved slightly as well. I also programmed a team mate for K1999. I will not go into the details of these improvement since this document is mainly about methods to find the optimal path.

Bibliography

[1] James S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). In *Journal of Dynamic Systems, Measurement and Control*, pages 220–227. American Society of Mechanical Engineers, September 1975.

[2] Charles W. Anderson. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114, Irvine, CA, 1987. Morgan Kaufmann.

[3] Charles W. Anderson. Approximating a policy can be easier than approximating a value function. Technical Report CS-00-101, Colorado State University, 2000.

[4] Christopher G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, 1994.

[5] Christopher G. Atkeson and Juan Carlos Santamaría. A comparison of direct and model-based reinforcement learning. In *International Conference on Robotics and Automation*, 1997.

[6] Leemon C. Baird. Advantage updating. Technical Report WL-TR-93-1146, Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1993. Available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145.

[7] Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Proceedings of the Twelfth International Conference*. Morgan Kaufman, 1995.

[8] Leemon C. Baird and A. Harry Klopf. Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-1147,

Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1993. Available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145.

[9] Andrew R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, May 1993.

[10] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[11] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13:835–846, 1983.

[12] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Experiments in parameter learning using temporal differences. *ICCA Journal*, 21(2):84–89, June 1998.

[13] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[14] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[15] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[16] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[17] Gary Boone. Efficient reinforcement learning: Model-based acrobot control. In *1997 International Conference on Robotics and Automation*, pages 229–234, Albuquerque, NM, 1997.

[18] Gary Boone. Minimum-time control of the acrobot. In *1997 International Conference on Robotics and Automation*, pages 3281–3287, Albuquerque, NM, 1997.

[19] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*. MIT Press, 1995.

[20] Robert H. Crites and Andrew G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262, 1998.

[21] Peter Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8:341–362, 1992.

[22] Peter Dayan and Terrence J. Sejnowski. TD(λ) converges with probability 1. *Machine Learning*, 14:295–301, 1994.

[23] Kenji Doya. Temporal difference learning in continuous time and space. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1073–1079. MIT Press, 1996.

[24] Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12:243–269, 2000.

[25] Stanislav V. Emelyanov, Sergei K. Korovin, and Lev V. Levantovsky. Higher order sliding modes in binary control systems. *Soviet Physics, Doklady*, 31(4):291–293, 1986.

[26] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie-Mellon University, 1988.

[27] A. F. Filippov. Differential equations with discontinuous right-hand side. *Trans. Amer. Math. Soc. Ser. 2*, 42:199–231, 1964.

[28] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In *Proceedings of 12th Australian Joint Conference on Artificial Intelligence*, Sydney, Australia, 1999. Springer Verlag.

[29] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In A. Prieditis and S. Russel, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 261–268, San Francisco, 1995. Morgan Kaufmann.

[30] M. Hardt, K. Kreutz-Delgado, J. W. Helton, and O. von Stryk. Obtaining minimum energy biped walking gaits with symbolic models and numerical optimal control. In *Workshop—Biomechanics meets Robotics, Modelling and Simulation of Motion*, Heidelberg, Germany, November 8–11, 1999.

[31] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[32] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.

[33] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[34] Yasuharu Koike and Kenji Doya. Multiple state estimation reinforcement learning for driving model—driver model of automobile. In *IEEE International Conference on System, Man and Cybernetics*, volume V, pages 504–509, 1999.

[35] R. Lachner, M. H. Breitner, and H. J. Pesch. Real-time collision avoidance against wrong drivers: Differential game approach, numerical solution, and synthesis of strategies with neural networks. In *Proceedings of the Seventh International Symposium on Dynamic Games and Applications*, Kanagawa, Japan, December 16–18 1996. Department of Mechanical Systems Engineering, Shinsyu University, Nagano, Japan.

[36] Yann Le Cun. Learning processes in an asymmetric threshold network. In *Disordered Systems and Biological Organization*, pages 233–240, Les Houches, France, 1986. Springer.

[37] Yann Le Cun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*. Springer, 1998.

[38] Jean-Arcady Meyer, Stéphane Doncieux, David Filliat, and Agnès Guillot. Evolutionary approaches to neural control of rolling, walking, swimming and flying animats or robots. In R.J. Duro, J. Santos, and M. Graña, editors, *Biologically Inspired Robot Behavior Engineering*. Springer Verlag, 2002.

[39] Martin F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[40] John Moody and Christian Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294, 1989.

[41] Jun Morimoto and Kenji Doya. Hierarchical reinforcement learning of low-dimensional subgoals and high-dimensional trajectories. In *Proceedings of the Fifth International Conference on Neural Information Processing*, pages 850–853, 1998.

[42] Jun Morimoto and Kenji Doya. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. In *Proceedings of 17th International Conference on Machine Learning*, pages 623–630, 2000.

[43] Rémi Munos. A convergent reinforcement learning algorithm in the continuous case based on a finite difference method. In *International Joint Conference on Artificial Intelligence*, 1997.

[44] Rémi Munos. *L'apprentissage par renforcement, étude du cas continu.* Thèse de doctorat, Ecole des Hautes Etudes en Sciences Sociales, 1997.

[45] Rémi Munos, Leemon C. Baird, and Andrew W. Moore. Gradient descent approaches to neural-net-based solutions of the Hamilton-Jacobi-Bellman equation. In *International Joint Conference on Artificial Intelligence*, 1999.

[46] Rémi Munos and Andrew Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *International Joint Conference on Artificial Intelligence*, 1999.

[47] Ralph Neuneier and Hans-Georg Zimmermann. How to train neural networks. In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*. Springer, 1998.

[48] Genevieve B. Orr and Todd K. Leen. Weight space probability densities in stochastic learning: II. transients and basin hopping times. In S. Hanson, J. Cowan, and L. Giles, editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, San Mateo, CA, 1993.

[49] Genevieve B. Orr and Todd K. Leen. Using curvature information for fast stochastic search. In *Advances in Neural Information Processing Systems 9*. MIT Press, 1997.

[50] Michiel van de Panne. Control for simulated human and animal motion. *IFAC Annual Reviews in Control*, 24(1):189–199, 2000. Also published in proceedings of IFAC Workshop on Motion Control, 1998.

[51] Stefan Pareigis. Adaptive choice of grid and time in reinforcement learning. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1036–1042. MIT Press, Cambrdige, MA, 1998.

[52] Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–160, 1994.

[53] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C—The Art of Scientific Computing*. Cambridge University Press, 1992.

[54] Jette Randløv and Preben Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML'98)*. MIT Press, 1998.

[55] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[56] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, Cambridge, MA, 1986.

[57] Juan Carlos Santamaría, Richard S. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–218, 1998.

[58] Warren S. Sarle, editor. *Neural Network FAQ*. Available via anonymous ftp from `ftp://ftp.sas.com/pub/neural/FAQ.html`, 1997. Periodic posting to the usenet newsgroup `comp.ai.neural-nets`.

[59] Stefan Schaal and Christopher G. Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, 14:57–71, 1994.

[60] Nicol N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *Proceedings of the 9th International Conference on Artificial Neural Networks*, London, 1999. IEE.

[61] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, August 1994. Available on the World Wide Web at `http://www.cs.cmu.edu/~jrs/jrspapers.html`.

[62] Karl Sims. Evolving 3D morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV Proceedings*, pages 28–39. MIT Press, 1994.

[63] Karl Sims. Evolving virtual creatures. In *Computer Graphics, Annual Conference Series, (SIGGRAPH '94 Proceedings)*, pages 15–22, July 1994.

[64] Russel L. Smith. *Intelligent Motion Control with an Artificial Cerebellum*. PhD thesis, University of Auckland, New Zealand, July 1998.

[65] Mark Spong. The swingup control problem for the acrobot. *IEEE Control Systems Magazine*, 15(1):49–55, February 1995.

[66] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[67] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press, 1996.

[68] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[69] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*. MIT Press, 1999.

[70] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[71] Mitchell E. Timin. The robot auto racing simulator, 1995. http://rars.sourceforge.net/.

[72] John N. Tsitsiklis. On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3:59–72, July 2002.

[73] John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[74] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[75] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[76] Thomas L. Vincent and Walter J. Grantham. *Nonlinear and Optimal Control Systems*. Wiley, 1997.

[77] Scott E. Weaver, Leemon C. Baird, and Marios M. Polycarpou. An analytical framework for local feedforward networks. *IEEE Transactions on Neural Networks*, 9(3):473–482, 1998. Also published as Univeristy of Cincinnati Technical Report TR 195/07/96/ECECS.

[78] Scott E. Weaver, Leemon C. Baird, and Marios M. Polycarpou. Preventing unlearning during on-line training of feedforward networks. In *Proceedings of the International Symposium of Intelligent Control*, 1998.

[79] Norbert Wiener. *Cybernetics or Control and Communication in the Animal and the Machine*. Hermann, 1948.

[80] Junichiro Yoshimoto, Shin Ishii, and Masa-aki Sato. Application of reinforcement learning to balancing of acrobot. In *1999 IEEE International Conference on Systems, Man and Cybernetics*, volume V, pages 516–521, 1999.

[81] Wei Zhang and Thomas G. Dietterich. High-performance job-shop scheduling with a time-delay TD(λ) network. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1024–1030. MIT Press, 1996.

Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur

Cette thèse est une étude de méthodes permettant d'estimer des fonctions valeur avec des réseaux de neurones feedforward dans l'apprentissage par renforcement. Elle traite plus particulièrement de problèmes en temps et en espace continus, tels que les tâches de contrôle moteur. Dans ce travail, l'algorithme TD(λ) continu est perfectionné pour traiter des situations avec des états et des commandes discontinus, et l'algorithme vario-η est proposé pour effectuer la descente de gradient de manière efficace. Les contributions essentielles de cette thèse sont des succès expérimentaux qui indiquent clairement le potentiel des réseaux de neurones feedforward pour estimer des fonctions valeur en dimension élevée. Les approximateurs de fonctions linéaires sont souvent préférés dans l'apprentissage par renforcement, mais l'estimation de fonctions valeur dans les travaux précédents se limite à des systèmes mécaniques avec très peu de degrés de liberté. La méthode présentée dans cette thèse a été appliquée avec succès sur une tâche originale d'apprentissage de la natation par un robot articulé simulé, avec 4 variables de commande et 12 variables d'état indépendantes, ce qui est sensiblement plus complexe que les problèmes qui ont été résolus avec des approximateurs de fonction linéaires.

Reinforcement Learning Using Neural Networks, with Applications to Motor Control

This thesis is a study of practical methods to estimate value functions with feedforward neural networks in model-based reinforcement learning. Focus is placed on problems in continuous time and space, such as motor-control tasks. In this work, the continuous TD(λ) algorithm is refined to handle situations with discontinuous states and controls, and the vario-η algorithm is proposed as a simple but efficient method to perform gradient descent. The main contributions of this thesis are experimental successes that clearly indicate the potential of feedforward neural networks to estimate high-dimensional value functions. Linear function approximators have been often preferred in reinforcement learning, but successful value function estimations in previous works are restricted to mechanical systems with very few degrees of freedom. The method presented in this thesis was tested successfully on an original task of learning to swim by a simulated articulated robot, with 4 control variables and 12 independent state variables, which is significantly more complex than problems that have been solved with linear function approximators so far.

Spécialité
Sciences Cognitives

Mots Clés
Apprentissage par renforcement, réseaux de neurones, contrôle moteur, commande optimale

Laboratoire
Laboratoire Leibniz-IMAG, 46 avenue Félix Viallet, 38000 Grenoble