Planning with Neural Networks and Reinforcement Learning

Gianluca Baldassarre

A thesis submitted for the degree of
Ph.D. in Computer Science

Department of Computer Science,
University of Essex,
Colchester, United Kingdom, 2002

Abstract

This thesis presents the design, implementation and investigation of some predictive-planning controllers built with neural-networks and inspired by Dyna-PI architectures (Sutton, 1990). Dyna-PI architectures are planning systems based on actor-critic reinforcement learning methods and a model of the environment. The controllers are tested with a simulated robot that solves a stochastic path-finding landmark navigation task.

A critical review of ideas and models proposed by the literature on problem solving, planning, reinforcement learning, and neural networks precedes the presentation of the controllers. The review isolates ideas relevant to the design of planners based on neural networks.

A "neural forward planner" is implemented that, unlike the Dyna-PI architectures, is taskable in a strong sense. This planner is capable of building a "partial policy" focussed on around efficient start-goal paths, and is capable of deciding to re-plan if "unexpected" states are encountered. Planning iteratively generates "chains of predictions" starting from the current state and using the model of the environment. This model is made up by some neural networks trained to predict the next input when an action is executed.

A "neural bidirectional planner" that generates trajectories backward from the goal and forward from the current state is also implemented. This planner exploits the knowledge (image) on the goal, further focuses planning around efficient start-goal paths, and produces a quicker updating of evaluations.

In several experiments the generalisation capacity of neural networks proves important for learning but it also causes problems of interference. To deal with these problems a modular neural architecture is implemented, that uses a mixture of experts network for the critic, and a simple hierarchical modular network for the actor.

The research also implements a simple form of neural abstract planning named "coarse planning", and investigates its strengths in terms of exploration and evaluations' updating. Some experiments with coarse planning and with other controllers suggest that discounted reinforcement learning may have problems dealing with long-lasting tasks.

To my family,
To Simona,
To every curious spirit intrigued by the beauty and mystery of this world.

Acknowledgements

I thank my supervisor Prof. Jim Doran for his support in the difficult moments of the PhD, for his hard criticism that helped me to trace interesting problems, for his precious contribution of ideas, and for his continuous stimulation to rethink implicit assumptions and abandon preconceptions. This thesis would not exist without him.

I thank the members of my supervisory board, Jeff Reynolds, Paul Scott, and Sam Steel for their criticism, their suggestions about interesting direction of research to develop and for their efforts to keep my work on schedule. Special thanks go to Paul Scott (my supervisor for the last two months of my PhD) for having helped me with the corrections of the final version of the thesis.

I thank the Heads of Department during my PhD, Ann De Roeck and Martin Henson, and the Department Staff and the administrative personnel for having guaranteed me a friendly and stimulating environment while doing my research.

I thank the Department and all the people that have had a role in the decision process that brought to the assignment of a full time scholarship for my research.

I thank James Adam and David Hales for the stimulating conversations we have had together, and for their precious help in the preparation of this thesis. I thank all the other friends in the Department for their warm support.

I thank the friends working at the Italian National Research Council, and in particular Prof. Domenico Parisi, Stefano Nolfi, Raffaele Calabretta, and Andrea Di Ferdinando, for their precious contribution of ideas, help and support.

I thank my family and Simona's family, and in particular my mother, father and sister, for their unlimited help, support, and love. I thank all my relatives, spread in Italy and in the world, for their affection.

I thank my friends of Italy, the “friends of flat 8” and the other friends at Essex University, for the fun we have had together, and for their friendship and support during the PhD.

I thank Simona for her simple and pure love.

Table of contents

List of Mathematical Symbols

β	Termination function of options (theory of options)
$\Re$ $\aleph$	Sets of real and natural numbers
$\exists, \forall, \in, \subset, \subseteq$	Set theory: "there is at least one…", "for every…", "is an element of …", "is contained in…", "is strictly contained in…".
$\sigma[.], \sigma'[.]$	Sigmoidal function and its derivative (argument: unit's activation potential)
$: , \rightarrow$	"Such that…", functional relation between two sets
[., .] (., .)	Intervals, including or excluding the interval extreme values
[.]	Square brackets used to indicate the arguments of a function or operator
{., ., ., .}	List of elements of a set
|	Operator used to express conditional probabilities, as in Pr[a | b]
$\partial f[x]/\partial x$	Derivative of function f[.] with regard to the variable x
A	Set of actions
a, a_t, a_{win}	An action, action at time t, selected action
argmax [.]	Argument of maximisation operator (arguments: variable, function)
b_l	Actor's gating network: activation potential relative to output unit l
c_k	Correctness of evaluator's expert k
d_{lq}	Actor's expert l: activation potential relative to action unit q
E[.]	Expectation operator (argument: a stochastic variable)
E_k	Output error of a neural network for the training pattern k
e_t	TD-error relative to the estimate V'[.] formulated at time t
f, f[.]	Pseudo-variable or transfer function of a neural unit (argument: activation potential)
g_k	A-priori weight (or "probability") of modular evaluator's expert k
h_k	A-posteriori weight (or "probability") of modular evaluator's expert k
I	Input set (theory of options)
k	Index of time when an option terminates (theory of options), or index of training pattern (feed-forward neural networks), or index of expert (mixture of experts neural network)
l[.], L[.]	Likelihood function and log-likelihood function (argument: an event)
ln[.]	Natural logarithm function (argument: a positive real number)
max.[.]	Maximisation operator (arguments: variable, function)
m_{lq}	Actor's expert l: action merit relative to action unit q
m_q	Actor: merit of action q
MTP MR	Model of environment: state transition function and reward function
n	Often used to indicate the number of elements of a set
n_l	Actor's gating network: activation of output unit l
o	An option (theory of options)
O[.]	Complexity asymptotic analysis of algorithms: number of steps taken by the algorithm to find a solution. Argument: parameter(s) that characterises the size of the input
o_k	Evaluator's gating network: output of unit k
p	Activation potential of one unit of neural network
$P^a_{ss'}$	Probability of getting to state s' if action a is executed at state s
Pr[.]	Probability (argument: an event)
Q*[.]	Optimal Q function (argument: a state of the world or perception)

$Q[.]$	Q function (argument: a state of the world or perception)
$Q'[.]$	Estimate of Q function (argument: a state of the world or perception)
$r^a_{ss'}$	Average reward obtained by executing action a at state s and passing to state s'
r_t	Reward at time t
s, s'	Two particular states
S, S_g	Set of states, set of goal states
s_g s_i	Goal state and start/initial state
s_t	State at time t
t	Index of time
u_{lj}	Actor's gating network: weight relative to output unit l and feature unit j
v	Vector of output units
$V^*[.]$	Optimal value function (argument: a state of the world or perception)
$V[.]$	Evaluation function (argument: a state of the world or perception)
$V'[.]$	Estimate of evaluation function (argument: a state of the world or perception)
v^d	Evaluator's expert k: desired output
v_k	Evaluator's expert k: activation of output unit k.
v_q	Feed-forward network: activation of output unit q
$V^\pi[.]$	Evaluation function depending on the action policy π (argument: a state of the world or perception)
w	Vectors or array of weights
w_{ji}	Weight between input unit I and feature (or hidden) unit j
w_{kj}	Evaluator's expert k: weight relative to feature unit j
X	The set of possible input vectors
x, $\mathbf{x}_t$	An activation vector of input units, activation vector at time t
x_I	Activation of input unit I
y	Activation vector of feature (or hidden) units
y_j	Activation of feature (or hidden) unit j
z	Vector or array of weights
z_{kj}	Evaluator's gating network: weight relative to gating unit k and feature unit j
γ	Discount coefficient
η, ζ, ξ, ν	Learning rates
π, π^*	A given policy (or the policy of an option within the theory of opitons), optimal policy
$\pi[. , .]$	Policy function: probability of action policy (arguments: state and action)
$\Sigma_f[.]$	Sum in f
μ	Global policy (theory of options)

1 Introduction

This introduction illustrates the objective of the thesis' research and the motivations behind it (section 1.1). It also introduces the specific issues and problems addressed in the thesis and presents an outline of the controllers proposed (section 1.2).

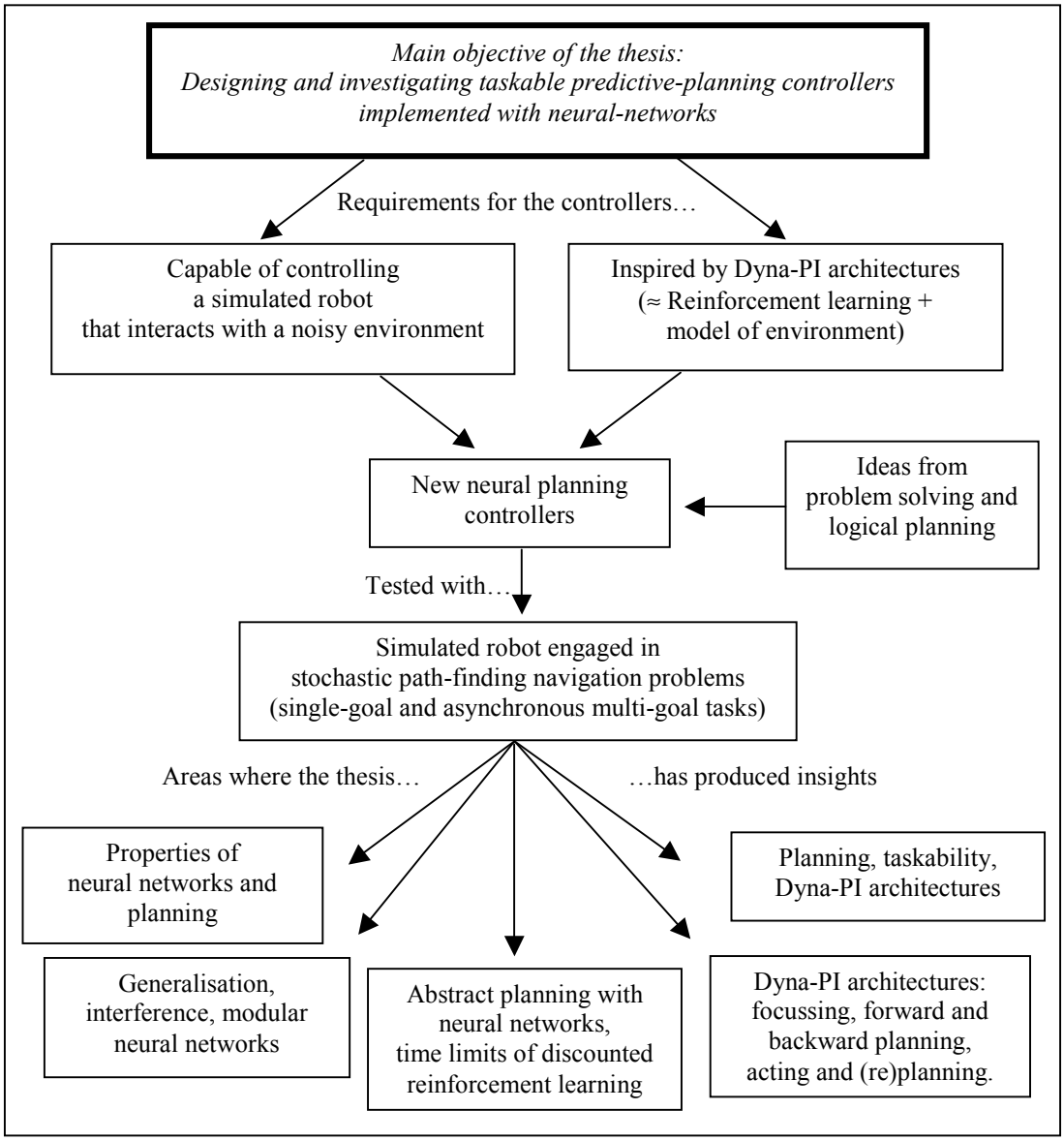


Figure 1.1: Development of the ideas presented in the thesis.

1.1 The Objective of the Thesis

Figure 1.1 presents a graphical summary of the ideas and problems investigated in the thesis. The main objective of the thesis can be defined as follows: *Designing and investigating taskable predictive-planning controllers implemented with neural-networks.*

The terms used in the definition of the thesis's objective are going to be precisely specified in s. 4.1 (neural networks) and 5.1 (taskable predictive planning). However, at least some approximate definitions are necessary for this introduction. "Predictive planning" can be defined as any information processing carried out by an agent, that constrains the agent's future course of action and is based on the agent's capacity to predict the consequences of its actions. A controller is "taskable" if it is capable of pursuing a goal ("desired state") assigned to it for the first time, on the basis of the use of previously acquired goal-independent information (cf. s. 5.1). "Neural networks" are systems made of simple units that exchange signals in parallel through a network of connections, and process the signals received in simple ways (cf. s. 4.1).

In pursuing the thesis' objective it has been decided to try to satisfy two requirements:

- The controllers should be capable of guiding a simulated robot that interacts with noisy environments.
- The controllers should be inspired by reinforcement learning and dynamic programming methods.

The following subsections first present the motivations for the choice of the thesis's objective and the motivation for adopting the requirement of a simulated robot interacting with a noisy environment. Then they present a brief review of the reinforcement learning framework and its possible use for planning. Successively they introduce some ideas drawn from problem solving and logical planning that inspired some aspects of the controllers implemented here, and present the reasons for which reinforcement learning, and in particular Dyna-PI architectures, have been adopted as a framework to tackle the research's objective (second requirement). Finally they briefly present the tasks that have been used to test the controllers.

1.1.1 Why Neural-Network Planning Controllers?

When used for control, neural networks usually play the role of "reactive devices" that yield a behaviour by directly mapping input sensorial patterns into output behavioural patterns (c.f. several works in Miller et al., 1990). Are neural networks suitable to implement "deliberative processes" where the input-output association is indirect, i.e. interleaved by some sophisticated information processing? Planning is a suitable candidate to attempt to answer this question. In fact, planning requires a "looping" information processing. This processing is necessary because planning implies that the prediction of the effects of actions' execution, produced by the system, is fed back into the system itself. Moreover, there is much literature on planning since it has been extensively studied by artificial intelligence since its birth (Fikes and Nilsson, 1971). Finally, planning is also very important for control (Russell and Norvig, 1995; Arkin, 1998).

Another issue that motivated the thesis's object, which is closely related to the previous one, is as follows. Many classic artificial intelligence planning systems are based on logical information representations that are set a-priori by the researcher (cf. several examples in Allen et al., 1990). Originally (e.g. the first version of the robot Shakey in 1969, cf. Russell and Norvig, 1995, pp. 787), when these planning systems were used to control robots, the sensors' readings were converted into logic representations, the control was implemented in terms of manipulations of these representations, and then the outcome of this processing was

converted into effectors' commands (cf. top part of Figure 1.2). This approach has difficulties, as the time-consumption of logical reasoning about the effects of low-level actions is too expensive to generate real-time behaviour (Russell and Norvig, 1995, p. 788).

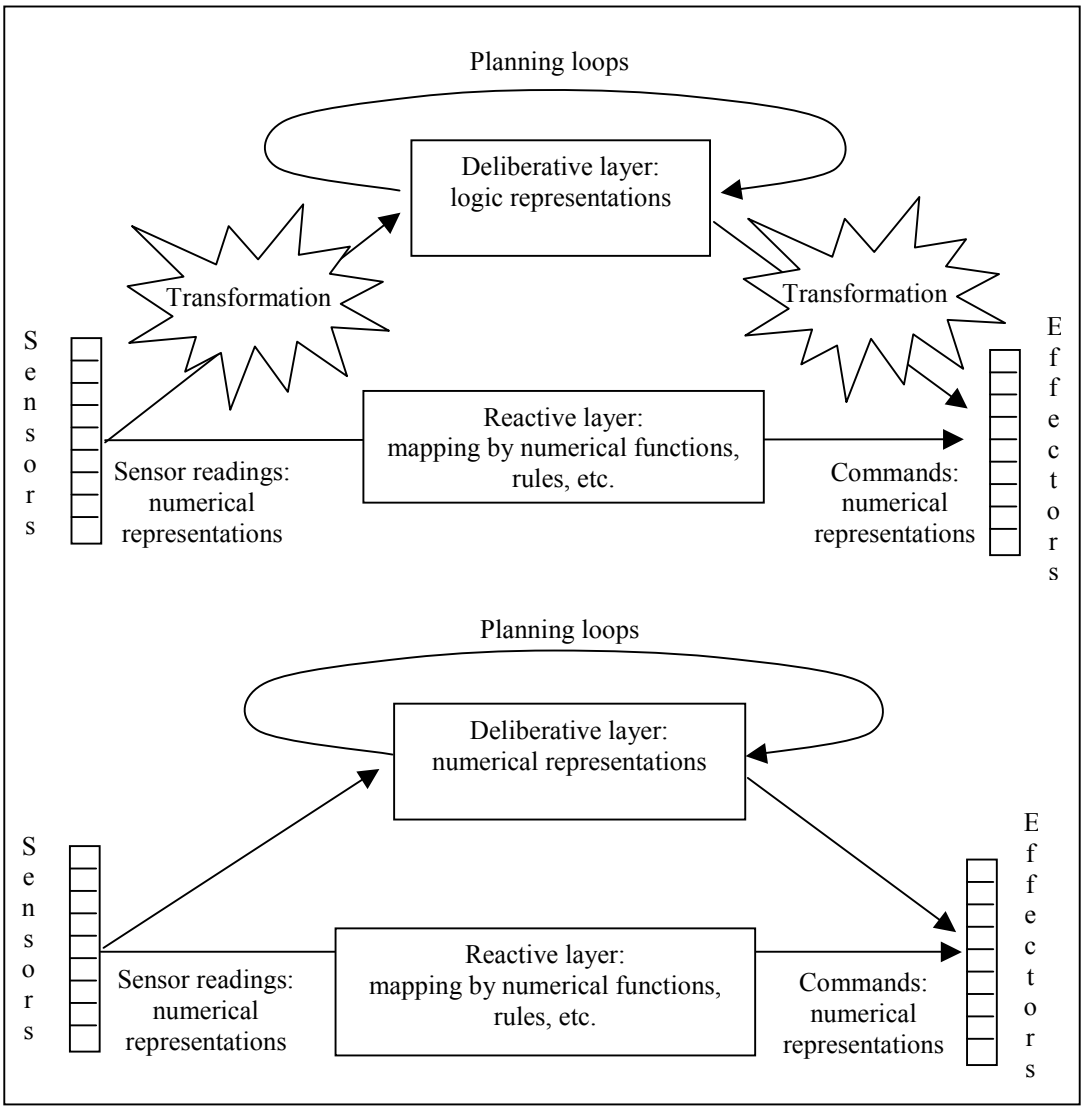


Figure 1.2: Top: Planning with logical information representation requires a double transformation of the numerical patterns yielded by the sensors into a logical format, and then a second transformation of the logical information processed into the numerical patterns for the effectors' commands. Bottom: A planning system that uses numerical representations both to plan and to react does not need the double transformation.

A different approach, the "behaviour-based approach", has proposed eliminating logical deliberation and planning from control altogether. At least initially, this approach has attempted to define control in terms of full reactive behaviour. This was implemented through numerical functions and/or rules that "directly" linked the sensors' numerical patterns to the effectors' numerical patterns (Brooks, 1986; Arkin, 1998; Murphy, 2000). However the pure behaviour based approach, and other "reactive" approaches, have shown to have important limitations. In fact exclusively-reactive systems are not capable of fully exploiting eventual

strong invariances of the environment, are not very flexible, and are non-taskable in a strong sense (Russell and Norvig, 1995, p. 790; Arkin, 1990, p. 206; cf. also s. 5.1).

The limitations of these approaches has led to the proposal of several "hybrid architectures" where a "reactive layer" uses quantitative/rule representations and the "deliberative layer" (planning) uses logic-like and abstract representations (Mitchell, 1990; Gatt, 1992; Morasso et al., 1992; Noreils and Chatila, 1995; Arkin, 1989, pp. 205-235, for a review). Unfortunately, also these systems have a significant limitation: they imply a repeated double recoding of information from a numerical format to a logic format and vice versa (cf. top of Figure 1.2). Indeed, the interface between the planning and the reactive components of these systems is difficult to implement, slow, and prone to errors (cf. Arkin, 1990).

The motivation of this thesis is to build reactive and planning systems that rely only on numerical representations. This novel approach should allow coping with noisy and unpredictable environments through reactive behaviours, having the flexibility of planning, and avoiding the problem of the interface between different information representation formats (see bottom part of Figure 1.2). Given the level of development that they have reached (cf. Haykin, 1999), neural networks have been chosen to pursue this goal (cf. s. 4.5 for a review on existing planning systems based on neural networks). This attempt is a challenge that is hard and interesting at the same time. In fact its solution implies to answer a number of questions of this type: How implementing the "looping" information processing required by planning with neural networks? What kind of information representations can be used to plan with neural networks? Can the neural system acquire the information needed to plan by experience? How can the neural planning process be used to influence future action? What are the advantages and disadvantages of using neural networks versus logic-based algorithms to implement planning?

1.1.2 Why a Robot and a Noisy Environment? Why a *simulated* robot?

As shown in Figure 1.1, it has been decided that the controllers designed should satisfy two requirements. The first requirement is that the planning controllers should be capable of guiding a simulated *robot interacting with a noisy environment*. This requirement is explained in this subsection. The second requirement is that the planning controllers should be based on the ideas of Dyna architectures (actually this is more a "decision" about where to look for solutions than a proper "requirement"). This requirement is illustrated in section 1.1.5.

Before illustrating the first requirement, the attention of the reader is drawn on the fact that the emphasis that this section puts on the use of a simulated robot for testing the controllers should not give rise to the idea that the thesis is mainly about robotics. In fact, as stated in section 1.1, the focus of the thesis is the development, implementation and testing of planning controllers based on neural networks.

The first constraint has been chosen because the author was interested in studying planning in a difficult context: one where a simulated robot perceives the environment through noisy and limited sensors and interacts with the environment with noisy effectors (cf. Brooks, 1986; Steels and Brooks, 1995). Such type of domains present problems that are different from those that arise from domains where planning agents interact with "engineered environments", such as the Internet. For example, as we shall see in s. 1.1.5, the fact that the effects of the actions are noisy and only partially predictable has fundamental implications for the principles and methods that can be employed for planning.

It has also been decided to study planning with a *simulated* robot and environment instead of a *real* robot. Simulations have disadvantages and advantages. The disadvantages are that simulations are usually simplified and their results do not entirely match the results

that can be obtained with physical experimental sets because simulations rarely capture the "full and infinite complexity" of reality (Steels, 1994; Nolfi et al., 1994; Jakobi et al., 1995; Steels and Brooks, 1995; Lee et al., 1998; Nehmzow, 2001). The advantages are that simulations are fast and cheap, allow the researcher to easily vary the robot's "body" (sensors, effectors, etc.), facilitate repeated experiments under identical conditions, and allow running experiments that cannot be easily executed with physical devices (Lee et al., 1998; Miglino et al., 1995). These advantages usually make simulations preferable *in the initial phase* of development of innovative controllers. In fact, in this phase it is usually necessary to change the physical properties of the robot to explore a wide range of scenarios and tasks, and to run a considerable number of experiments in a short period of time.

An example of the role played by simulations in developing controllers is reinforcement learning, the framework adopted in the thesis. The majority of new reinforcement learning techniques have been developed and tested with simulated agents and worlds. For example this has been true for: the "actor-critic methods" (Barto et al., 1983); the "Dyna-PI architectures" (Sutton, 1990); "prioritised sweeping" (Moore and Atkeson, 1993); "temporal abstract reinforcement learning" (Sutton et al., 1998); cf. Sutton and Barto (1998) for many other examples. This has not prevented reinforcement learning techniques from becoming some of the most widely used techniques to control physical robots (cf. the proceedings edited by Meyer et al., 2000).

Given the advantages rendered by simulations in the initial phase of development of new controllers, here it has been decided to use a simulated robot and simulated environments. These simulated robot and environments try to maintain the interesting problems raised by the control of real robots in real environments, such as continuous interaction with the environment, perception through noisy and limited sensors, action through noisy effectors. This should facilitate the further development and application of these same controllers to real robots.

1.1.3 Reinforcement Learning, Dynamic Programming and Dyna Architectures

As mentioned, a second requirement has been chosen for the controllers: they should be based on the reinforcement learning framework. This subsection briefly introduces the major aspects of this framework (cf. chapter 3 for details), while s. 1.1.5 explains why this framework has been adopted.

The expression "reinforcement learning framework" is used here, as in the title of the thesis, to refer to a constellation of methods and theories that have been developed independently, but then have been shown to have important common principles (cf. chapter 3). These methods and theories are: "Markov Decision Processes" (MDP), "Reinforcement Learning" (RL), "Dynamic Programming" (DP) and "Dyna-PI architectures" (the acronyms are indicated because they are often used in the literature, but the extended terminology is used here to ease the reading).

Markov decision processes (Puterman, 1994) and reinforcement learning (Barto et al. 1983; Barto et al., 1990; Kaelbling et al., 1996; Sutton and Barto, 1998) are among the best theories currently available to frame sequential decision problems under uncertainty (Russell and Norvig, 1995, p. 498; they are also referred to as "Markov decision problems" or "reinforcement learning problems"). They assume that an agent knows exactly the current state of the world ("Markov assumption") and has to select and execute an action among a set of available actions. As a consequence the environment returns a new state according to some given probabilities distributed over the possible states, called "transition probabilities". Positive or negative "rewards" (or "utilities") are assigned to some states and a 0 reward is

assigned to the remaining states (e.g. in the stochastic path-finding problems, cf. s. 1.1.6, the "goal-state" is assigned reward +1, while all the other states are assigned reward 0).

Within this stochastic context the concept of "plan" (a sequence of "operators" to execute) employed in the classic artificial intelligence searching and planning literature, is substituted by the concept of "policy". A policy associates an action probability distribution with each state. This distribution determines the probabilities that the agent will select each action. The strength of the idea of "policy" is that whatever the consequences of an action are in terms of the new state reached after its execution, the agent is not committed by any previous decision and can decide what to do on the basis of the new state itself. The agent's task is to find an optimal (or near-optimal) policy, i.e. a policy that (in the most popular formulation of reinforcement learning tasks) maximises the expected future discounted rewards starting from each possible state. In the case of the stochastic path-finding problems this means that the agent has to find a policy that leads from the start state to the goal state following the most direct path, notwithstanding the perturbations caused by noise.

Now let us consider dynamic programming methods. Given a reinforcement learning problem, if a *model of the environment* is available, dynamic programming methods (Ross, 1983; Bertsekas, 1995) are capable of generating optimal policies by using it. A model of the environment is made up of two components. The first component, the "state transition function", returns the state achieved after selecting a particular action at a given state on the basis of the transition probabilities. The second component, the "reward function", returns the reward associated with the execution of a given action in a given state. Dynamic programming methods are based on the generation of a gradient field of "evaluations", associated with the states, that are higher for states closer to states with high positive rewards (e.g. the goal state). The evaluations of all the states are computed iteratively and in parallel ("full sweep"). The evaluation of one state is updated on the basis of the approximate evaluation of all the states reachable from it and on the basis of the transition probabilities ("full back-up"). At execution time, dynamic programming selects the actions that ascends the gradient field along the steepest direction (policy). Clearly, dynamic programming implements a form of planning since it uses a model of the environment to guide the course of action.

Given a reinforcement learning problem, if a model of the environment is not available reinforcement learning algorithms (Sutton and Barto, 1998) are capable of finding a policy by using a trial-and-error process directly executed in the environment. Similarly to dynamic programming, reinforcement learning algorithms compute state evaluations and action policies based on those evaluations. However, unlike dynamic programming, they update the state evaluations and the action policies by executing actions in the environment and by observing the consequences in terms of states reached and rewards obtained ("sample back-up"). As a consequence, the updating of evaluations and policy affects only the states actually visited ("focussing"; however the convergence of reinforcement learning algorithms usually requires that all states are visited an infinite number of times, cf. Sutton and Barto, 1998). With more experience, the state evaluations become more accurate and the policy changes towards the optimal one.

In reinforcement learning the policy is usually generated dynamically from the evaluations each time that the agent needs to select an action (cf. the popular Q-Learning algorithm, Watkins, 1989, and Watkins and Dayan, 1992, reviewed in s. 13.2.3). The controllers designed and implemented here are based on the "actor-critic reinforcement learning methods" (Barto et al., 1983; Sutton and Barto, 1998). Actor-critic methods are characterised by two memory structures, one to store the evaluations, and one to store the

policy probabilities. This research has chosen to adopt actor-critic reinforcement learning instead of other reinforcement-learning methods for the following reasons:

- Convergence of reinforcement learning that uses a differentiable function approximation, that satisfy some particular conditions, has been demonstrated only for the case of actor-critic methods (i.e. methods that use "Policy Iteration", see Sutton et al., 2000).
- The best stochastic policy can be better than the best deterministic policy in Partially Observable Markov Decision Processes (Singh et al, 1994; Jaakkola et al., 1995; cf. s. 13.2.2).
- The author is interested in actor-critic models because they are more biologically plausible than other reinforcement learning models (Sutton and Barto, 1990; Houk et al., 1994; Sutton and Barto, 1998; Baldassarre and Parisi, 2000; Baldassarre, 2001b; Baldassarre, 2001e).

Sutton (1990) has integrated dynamic programming and reinforcement learning into a class of architectures called "Dyna" ("Dyna" stands for "dynamic programming"). When the Dyna architectures are based on actor-critic reinforcement learning methods they are called "Dyna-PI architectures" where "PI" stands for "policy iteration", the key process at the base of actor-critic methods (cf. s. 13.2.3). The basic idea of Dyna architectures is to have a reinforcement learning architecture that is trained in the environment but also through a model of the environment used to generate "simulated" extra experience, similarly to what is done in dynamic programming.

The new controllers presented in the thesis are inspired by Dyna-PI architectures. These controllers overcome some drawbacks of Dyna architectures concerning predictive planning and taskability (cf. s. 5.1 and 8.3.2). Once this is done, these controllers are used to investigate the advantages and disadvantages of implementing planning with neural networks (s. 1.2 presents an overview of the issues explored).

1.1.4 Ideas from Problem Solving and Logical Planning

The field of problem solving and search strategies (Korf, 1988; Russell and Norvig, 1995, pp. 55-121; cf. s. 2.2 for a review) tackles problems where an agent has to find a sequence of states that lead from a starting state to a goal state. In some problems this is done on the basis of information about the approximate (usually optimistic, or "admissible") "heuristic", i.e. the estimate of the cost from each state to the goal. Interestingly the concept of state evaluations on which dynamic programming and Dyna architectures are based is related to the concept of "heuristic" employed within problem solving. For example, it has been shown that a particular form of dynamic programming, namely "trial-based real-time asynchronous dynamic programming", is equivalent to a particular form of heuristic search, namely "learning real-time A*" (Barto et al., 1995). As we shall see in s. 13.2.11 these two methods are equivalent because the evaluations and the heuristic values that they respectively *learn*, have a close correspondence.

This research has isolated some ideas developed within problem solving that could inspire the design of the neural planning controllers presented later. Notice that the original ideas from problem solving could not be directly applied in the new context, mainly because problem solving has been developed for deterministic environments, while the neural planners considered here should be capable of dealing with stochastic environments. The following ideas proposed by the problem solving literature have been relevant to develop the controllers proposed here:

- Achieving a full taskability of the neural planners through the use of a neural network that can establish if the current state is similar enough to the current state (analogously to the idea of "goal test" used in problem solving, cf. s. 2.1).
- Implementing planning in terms of an exploration of the model of the environment from the start and from the goal (analogously to the idea of "bidirectional search", cf. again s. 2.1).
- Executing iterative deepening explorations of the model of the environment (analogously to the idea "iterative deepening search", cf. again s. 2.1).

Planning is the field of artificial intelligence that has traditionally tackled the problem of deciding the future course of action on the basis of the agent's capacity to predict its consequences. Planning is applied to problems similar to those of problem solving and is closely related to it. Planning differs from problem solving because it "breaks" the "monolithic" representation of state used in problem solving into logical statements ("STRIPS representation"). This operation allows planning to gain in efficiency when looking for a solution of the problem (cf. s. 2.3 for details). Unluckily this strategy cannot be directly followed when using neural networks (cf. s. 2.3.3 for details). Notwithstanding this, this research has attempted to suitably transform and transfer some ideas developed within planning to the design of the neural planning controllers based on reinforcement learning. In particular the following ideas have been considered for this research:

- Importance of planners being able to deal with uncertain outcomes of actions (cf. s. 2.4.1).
- Necessity of finding a balance between "conditional planning" (this kind of planning is close to the idea of "policy"; cf. s. 2.4.1) and "re-planning" (this is a kind of planning that stops the action to improve/formulate a new plan in particular circumstances; cf. s. 2.4.2). The planners designed and implemented here offer a solution to this problem, based on the "confidence in action" of the agent (cf. s. 8.3.2).
- Non-scalability of "universal planners" (cf. s. 2.5.2): this same problem applies to the concept of "policy" (cf. s. 2.5.5).
- Importance of "abstract planning": here a controller that implements a simple form of abstract planning is proposed (cf. s. 2.4.3 and chapter 11).

1.1.5 Why Dyna-PI Architectures (Reinforcement Learning + Model of the Environment)?

In s. 1.1.3 the main concepts at the basis of reinforcement learning and Dyna architectures have been introduced. Why has this framework been chosen to build the neural network planners presented here? This subsection answers this question. Dealing with agents that act in noisy environments has important implications for planning. The first STRIPS planners (Fikes et al., 1971) had several problems in dealing with these kinds of environments. In fact they decided the course of action a-priori on the basis of a model of the environment that was assumed to be perfect, and then they executed the actions in the environment in a "blind" way, i.e. without monitoring the effects of the execution of actions. The problems derived from the fact that the execution of actions often resulted in effects different from the expected ones (Russell and Norvig, 1995, p. 392 and p. 787). Later versions of these planners offered some solutions to deal with these problems. For example they contemplated a-priori different possible outcomes of actions (e.g. cf. "conditional planning", Warren, 1976; see s. 2.4.1) or monitored the effects of actions' execution (e.g. IPEM, Ambros-Ingerson and Steel, 1988; cf. s. 2.4.2).

This approach has gone even further. If an agent has an "action function" (e.g. a look-up table or a set of condition-action rules) that specifies what action to select in correspondence to a given state, it does not need to worry about unexpected developments in the environment. All it has to do is to execute whatever action the action function recommends for the state in which it finds itself. The field of "reactive planning" aims at taking advantage of this fact, thereby avoiding the complexities of planning in dynamic, inaccessible environments. "Universal plans" (Shoppers, 1987; Shoppers, 1989; cf. s. 2.5.2) were developed as a general scheme for reactive planning. A universal plan is a function f that maps the set of states S into the set of action A, i.e. f: S $\rightarrow$ A. In an initial phase the planning process "compiles" information into the universal plan. Successively the universal plan is used to act in the world in a reactive fashion. Interestingly reactive planning turned out to be a rediscovery of the idea of policy of Markov decision processes, used throughout this research (Russell and Norvig, 1995, 411).

Are Markov decision processes and the concept of policy the final solution for planning in stochastic environments? Probably not. This approach, like universal planning, also has important limitations (Ginsberg, 1989). The most important ones derive from this fact: the system has to know what to do for *every* possible state. This implies two problems if the state space is big, as in the majority of realistic problems. The first problem is that a lot of time is needed to prepare ("compile") the plan since the agent needs to decide what to do for every possible state. The second is that a big memory structure is needed to store the "compiled" plan. Using function approximation methods such as neural networks can alleviate these problems (Sutton and Barto, 1998, p. 193; cf. s. 13.2.7 and 4.4.1). However, these problems are still central for planning since one of the strengths of planning is precisely that it allows agents to quickly prepare a plan focussed on few important states (cf. s. 2.4 and 8.3).

Given these considerations, it is likely that the optimal solution is between the two extremes of having rigid plans (plus re-planning in the case of failure) and universal planning (cf. Russell and Norvig, 1995, pp. 407-409). This means that a good strategy would be to plan when necessary, to build plans with a certain degree of robustness that can deal with some unexpected but likely outcomes (i.e. to focus planning on states that are likely to be visited), and to re-plan when actions' outcomes are different from those expected. As we shall see the controllers implemented in this thesis have such properties. In fact they are inspired by the Dyna-PI architecture (the basic version of this is an instance of universal planners) but they also incorporate some aspects of re-planning and planning focussed on relevant states.

A last reason (quite different from the previous ones) for which reinforcement learning and Dyna architectures have been chosen to implement planning, is one particular interest of the author related to the development of animals' brain during natural evolution: what is the minimal "machinery" that needs to be added to a reactive learning controller to obtain a planning controller? Though interesting, this aspect has not been developed in the thesis to preserve its focus on computational issues (but cf. Baldassarre and Parisi, 2000; Baldassarre, 2001b; Baldassarre, 2001e; Baldassarre, 2002).

1.1.6 Stochastic Path-Finding Landmark Navigation Problems

This section describes the type of tasks that have been used to test the controllers designed and implemented here (cf. s. 3.1 for a formal formulation). We have already seen the nature of reinforcement learning problems. Broadly speaking, these problems are characterised by fact that they do not have a termination. Several states of the space problem have a positive reward associated with them, and the agent has to behave so that in each state it maximises the sum of the expected discounted future rewards (Sutton and Barto, 1998, p. 60-61). In s. 5.1.4 we will

see that Dyna architectures, that are capable of dealing with this broad category of problems, are not "taskable" when planning. In fact each *new* goal implies a new reward function, and this reward function has either to be learned by the agent by experiencing the goal itself, or it has to be provided by the designer. We will see that a solution of this problem is to restrict the category of problems that the controllers can tackle to "stochastic path-finding problems". Stochastic path-finding problems are similar to the problems of "problem solving": the agent has to find the most direct path from a start state to a goal state.

The particular stochastic path-finding problems used in this thesis are "landmark navigation problems". In these problems the agent has to reach a goal position from a start position in a two (or n) dimensional space, by referring to the *view* (or other similar information) of some landmarks spread in the environment. In the problems considered here a simulated robot moves in a continuous two-dimension space where there are few landmarks (with the exception of chapter 11, in the simulations considered here the robot can cover one side of the arena in 20 moves).

It is important to consider why the use of a simulated robot makes the problem more interesting and complex in comparison to the problems usually considered within problem solving. To understand this complexity it is useful to look at the problem from the point of view of the simulated robot employed here. The simulated robot perceives the environment through a horizontal one-dimensional binary retina, always aligned with the simulated magnetic north. The retina returns a vector of 50 bits corresponding to the activation of its pixel sensors (2^{50} possible configurations). The simulated robot can move in the arena by executing one step in the 8 compass directions (north, northeast, east, etc.). The simulated robot's task is to move to a position where the retina is activated in a way similar to a given template assigned to the robot itself (goal). Both actions and perceptions are affected by noise. In contrast to problem solving tasks, no information is furnished to the simulated robot about which state is a neighbour of which state. If the agent wants to plan, it first needs to "learn" this relationship between states in terms of "which state results from which action executed in which state" (model of the environment). Moreover, when it "plans" it has to consider that its model is imperfect because of noise, and that the execution of an action in identical conditions may produce different outcomes because of noise.

Notice that in the task just described no particular landmark or signal emitter marks the goal position. The goal position is recognisable only when it is reached, because it matches the given template. The literature considers this type of problem (e.g. McGovern et al., 1997; Trullier and Meyer 1998) much more challenging than path-finding problems where there are signal emitters or special markers at the goal position (e.g. Rummery and Niranjan, 1994; Sun and Peterson, 1998). In fact in the problems considered here the simulated robot has to be capable of selecting a particular action in correspondence with a *view* of the environment. Instead, in the easier case where there is an emitter or special marker at the goal position the *direction* of the goal is available at each position visited, so the agent can solve the problem by simply aligning the direction of its movement with the direction of the goal (while "local" information is usually used to avoid obstacles).

Although the controllers have been tested only with landmark navigation problems, efforts have been made to design controllers that are general and also applicable to other problem domains. In the course of the thesis it will be stated clearly when this has not been possible, i.e. when some aspects of the controllers rely upon specific features of landmark navigation problems. The thesis will also illustrate some results related to the particular nature of landmark navigation tasks when tackled with reinforcement learning and neural networks.

1.2 Overview of the Controllers and Outline of the Thesis

The thesis is divided in two parts. The first part (chapters 2 to 5) presents a critical literature review and an analysis of the ideas and principles relevant for neural planning. The second part (chapters 6 to 11) presents some new neural-network planners and an empirical analysis of them through simulations. The thesis is also divided in chapters, sections and subsections (for ease of reference, sections and subsections are referred to as "sections" or "s." for short, for example "cf. s. 1.2").

1.2.1 Overview of the Controllers Implemented in this Research

The continuity of the thesis is guaranteed by two elements. The first is that the controllers presented in the thesis are built incrementally through the chapters, i.e. each new controller is built by adding new components to the previous controller. The second is the nature of the problems addressed. The first aspect is discussed here, while the second is discussed in s. 1.2.2. The whole *final* controller is reported in Figure 1.3. The reader can read the label of this figure to gain an idea of the architecture of the controllers, and skip the rest of this section, or read the general description of them that follows.

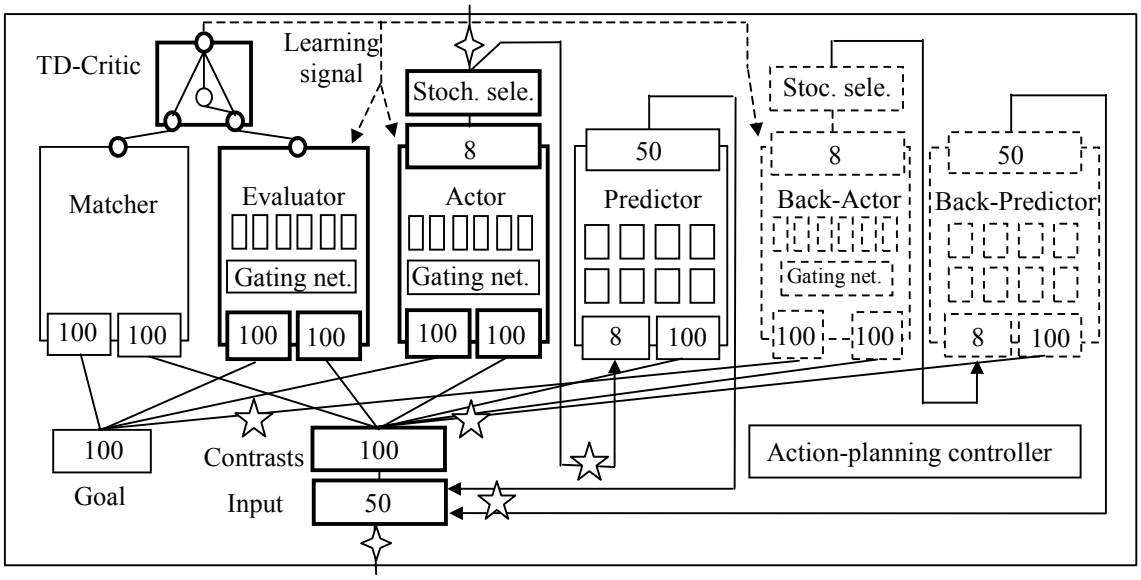


Figure 1.3: The final controller presented in the thesis developed incrementally through the chapters. Networks with a bold border implement a simple reinforcement learning controller (chapter 6), or a modular neural reinforcement learning controller (chapter 7). Networks with a thin border implement forward planning (chapter 8). Networks with a dashed border implement bidirectional planning (chapter 9). The whole system implements bidirectional planning with modular neural networks (chapter 10). The whole system without modularity has been used to implement forward and backward planning with a simple kind of abstraction called "coarse planning" (chapter 11). Numbers indicate the number of units of each neural layer. Stars indicate the points where the information flows between the modules are enhanced or blocked during acting or planning.

Figure 1.3 shows that the controllers are mainly composed of two-layer feed-forward neural networks connected in a direct or re-entrant way between them, or with the environment. The core of the algorithm is made up by the actor, the evaluator and the TD-critic networks that together implement an actor-critic reinforcement learning system. The

evaluator has the role of learning the evaluations of the states, and hence of producing the gradient field of evaluations, and the actor has the role of learning the policy on the basis of such a gradient field, and hence of selecting the actions. This basic model has been tested in two versions: one where the actor and evaluator are simple "monolithic" networks (chapter 6) and one where the evaluator and actor are "modular networks" (chapter 7). Both these models learn by interacting with the environment.

The "neural forward planner" (chapter 8) is built by adding two further networks to this basic model. The first network is the matcher, a network capable of deciding if the current state is or is not the goal. This network produces the reward signal used to train the evaluator and actor. It corresponds to the "reward function" part of the model of the environment. The second network is the predictor, a network capable of predicting the next state that will follow the execution of an action in correspondence to a given state. It corresponds to the "state transition function" part of the model of the environment. A hardwired algorithm, the "action-planning controller" showed in Figure 1.3, decides when to act and when to plan. This algorithm also controls the flows of information between the networks of the system, the sensors, and the effectors, by "opening and closing" the flows at the level of the stars indicated in the figure.

Given that planning is central to the thesis, it is useful to briefly anticipate how the neural planning process works in these systems. In the forward planner, planning takes place through reinforcement learning *executed within the model of the environment* rather than in the actual environment. When the system plans, it generates repeated sequences of projections into the future starting from the current state. To this end, the actor selects an action, the predictor predicts the next state on the basis of the action and the current state; this predicted state is sent back to the actor and the predictor (re-entrant connections); the actor selects another action and the predictor generates another predicted state, so the process goes on generating a "trajectory" into the future. While many trajectories are generated, the evaluator and actor are trained as it would be done in the case of training in the actual environment. When the controller is "confident" enough, i.e. it assigns high probabilities to one or few actions in correspondence to the current state, the policy expressed by the actor is executed in the environment.

The "neural bidirectional planner" (chapter 9) works similarly to the forward planner. The main difference is that it generates sequences of actions and predictions both forward from the current state and backward from the goal. This bidirectional planner is built by adding two neural networks to the forward planner. The first neural network, the "back-actor", is capable of "guessing" which was the action that led to a particular state. The second neural network, the "back-predictor", is capable of "back-predicting" what could have been the state that brought to a given state by executing the action yielded by the back-actor in correspondence to this given state itself.

Both the forward and bidirectional planners are also endowed with a modular evaluator and actor and tested with a multi-goal task (chapter 10). Finally, the forward planner is used to implement a simple form of abstract planning based on the repetition of actions of the same kind (e.g. "north, north, north", chapter 11).

1.2.2 Outline of the Thesis and Problems Addressed Chapter by Chapter

The second element of continuity across the chapters is the nature of the problems addressed. These are all related to the objective of the thesis: developing taskable predictive-planning neural controllers inspired by the Dyna-PI architecture. These problems are now introduced by presenting an outline of the thesis chapter by chapter.

First Part. The first part, from chapter 2 to 5, has two objectives. The first is to present a literature review of the frameworks within which the controller have been developed. The second is to present a critical analysis of the strengths and weaknesses of these contributions with the aim of isolating ideas and principles that can be useful to implement planning with neural networks.

Chapter 2. Chapter 2 first presents the algorithms used by blind and heuristic search, and refers the reader to the appendices for details. Heuristic search is particularly important because it has strong connections with dynamic programming and reinforcement learning methods in general. The chapter continues by presenting a review of planning. Important ideas, as "re-planning" and "conditional planning", are discussed here.

Chapter 3. This chapter introduces the Markov decision processes, the reinforcement learning problem, the actor-critic models, dynamic programming, the correspondences between dynamic programming and heuristic search, and refers the reader to the appendices for the mathematical details. The chapter also presents the details of path-finding problems (the class of problems dealt with in the thesis), Dyna and Dyna-PI architectures, and the issue of how focussing planning on limited areas of the state space. The review of these techniques is accompanied by comments on their relevance for neural planning.

Chapter 4. This chapter starts by defining neural networks and by analysing their properties relevant for planning. Then the reader is referred to the appendices for a mathematical presentation of the feed-forward neural networks, the back-propagation algorithm, and the "mixture of experts networks", that are building blocks used in the controllers presented here. The chapter ends by reviewing some of the most important existing planning systems based on neural networks.

Chapter 5. This chapter presents a formalisation of learning of behaviour and taskable planning, and uses it to show that the Dyna-PI planner is not taskable in a strong sense. Then it presents a unified view of some heuristic search methods, some reinforcement learning and dynamic programming methods, and the activation diffusion planning method. This unified view is centred on the idea of evaluation gradient field. Finally it summarises the nature, advantages and disadvantages of "plans" in comparison to "policies".

Second Part. The second part of the thesis, from chapter 6 to chapter 11, illustrates, implements and empirically tests some neural reactive and planning controllers based on the ideas presented in the first part.

Chapter 6. This chapter introduces the simulation scenarios used throughout the research. Then it presents the neural controller based on the actor-critic model, which is at the core of all the controllers implemented in the following chapters. Some simulations investigate the functioning, the strengths (such as the generalisation capacity) and the drawbacks (such as the aliasing problem) of this controller, and provide some data useful for interpreting the results of the succeeding chapters. Some other simulations suggest that discounted reinforcement learning has problems in dealing with long periods of time. This issue is very important for planning, as planning expresses its full power when it deals with long periods of time.

Chapter 7. This chapter addresses the problem of interference. This is an important problem encountered when planners have to accomplish several tasks. To tackle this problem, a multi-goal version of the landmark navigation problem is introduced, where the simulated robot has to pursue different goals at different times. A new controller is designed and implemented that deals with the problems caused by interference through "emergent functional modularity". In this controller a "mixtures of experts network" is employed to implement the evaluator (evaluation function) and a novel hierarchical network to implement the actor (policy).

Chapter 8. This chapter deals with the problems of taskability of the Dyna-PI architecture, and with the problems of focussing planning around relevant states and interleaving acting and planning. Afterwards, the chapter presents a "neural forward planner" that offers a solution to these problems. In particular the forward planner is taskable, is capable of focussing planning on relevant states, and is capable of deciding when to act and when to plan. In comparison with the basic version of the Dyna-PI architecture and the classic artificial intelligence planners, the controller is new in that:

- Most of its components are implemented with neural networks.
- It generates the reward internally by comparing the states visited with the goal state through the "matcher". The matcher allows the controller to be taskable in a strong sense. This differs from the Dyna-PI architectures where the reward function has to be learned for each new goal assigned to the agent or has to be furnished by the user/designer.
- The controller decides whether to plan or to act on the basis of its "confidence" in action. The confidence is computed on the basis of the probabilities of selecting the different actions.
- A new algorithm is designed and implemented to guide the "simulated experiences" while planning. As mentioned previously, this algorithm iteratively explores the model of the environment starting from the current position in search for the goal. The length of these explorations increases with failure, and decreases with success, at achieving the goal.
- Contrary to classic artificial intelligence planners, the "model of the environment" is learned through a modular neural-network, the "predictor". The experiments show that the predictor has an interesting capacity to maintain the coherence between the states predicted while planning and real states, probably because the patterns that correspond to real states are a sort of "attractor" for the predicted states.

Chapter 9. The problem addressed in this chapter is how to further focus planning on relevant regions of the state space. While planning the controller presented here, the "neural bidirectional planner", carries out both forward searches from the current state, and backward searches from the goal. In comparison to the forward planner of chapter 8, this controller shows a better capacity of focussing "planning search" around the goal and quickly propagating the evaluations backward from the goal. The chapter also shows that the drawbacks of the controller are its architectural complexity and the need to generate simulated experience backward from the goal.

Chapter 10. This chapter addresses the problem of generalisation, interference and modularity, introduced in chapter 7, within planning. To this purpose it compares the performance of the reactive controller, forward planner, and bidirectional planner within the asynchronous multi-goal task. This test is important since planning tends to exacerbate interference problems because it focuses search on the same goal for a long time.

Chapter 11. This chapter addresses the problem of how implementing abstract planning with neural network systems. In particular it explores a simple form of abstraction, called "coarse planning" for reference. Coarse planning is based on planning with "coarse actions", i.e. small sequences of primitive-actions of the same kind, and acting with "primitive-actions". Some simulations also investigate the problem of the temporal limits of discounted reinforcement learning, introduced in chapter 6, within the context of coarse planning.

Chapter 12. This chapter summarises the main passages of the thesis, highlights the most important insights achieved, and suggests the future work to do to continue the research started here.

PART 1

CRITICAL LITERATURE REVIEW AND ANALYSIS OF CONCEPTS USEFUL FOR NEURAL PLANNING

This first part of the thesis has two objectives:

- To offer a review of the results of four fields of artificial intelligence relevant for this research. These fields are "blind" and "heuristic" search, planning, reinforcement learning and dynamic programming, and finally neural networks.
- To present critical observations that evaluate the relevance, strength and drawbacks of the different approaches/algorithms/systems presented, in order to isolate some ideas and principles that might be useful to design neural planners.

To highlight the original elaboration of the material, the review part will be contained in standard sections, while the critical observations will be presented in sections titled "critical observations" (when observations presented by these sections are not original, references will be given).

2 Problem Solving, Search, and STRIPS Planning

Broadly speaking, problem solving and planning are two fields of artificial intelligence that study agents whose task is to "search" a "sequence of actions" ("operators") that bring from an "initial state" to a "goal state" through the "state space". As we shall see the main difference between the two fields is that problem solving treats goal, states and actions as a whole, while planning decomposes them into parts each represented by a logical description (Russell and Norvig, 1995, p. 339). This opens up a number of possibilities that make the search of a solution much more efficient (cf. s. 2.3).

2.1 Planning as a Searching Process: Blind-Search Strategies

The most important aspects involved in the solution of a problem within "problem solving" are these (Russell and Norvig, 1995, p. 60):

- State space. The set of all possible states of the environment.
- Initial state. The state from which the agent starts to solve the problem.
- Goal and goal test. A goal is a state that is "desirable" for the agent. There might be more than one goal. The goals can be explicitly listed or identified by abstract properties. A goal test is a procedure directed to verify if a particular state corresponds to a goal.
- Actions (operators). The actions with which the agent interacts with the environment. The term "operator" is used to denote an action in terms of the state that will be reached by carrying out the action in a particular state. In problem solving little importance is given to the details of execution of the actions, so the stress is put on the next states achievable from one particular state. The execution of an operator is often associated with a cost (often equal to 1 for all operators).
- Path and solution. A path is a sequence of actions leading from one state to another state, while a solution is a path leading from the initial state to a goal state.
- Path cost. Sum of the costs of the individual actions along a path. The path cost from the initial state to a state s is indicated by g(s).
- Search and expansion. The process of looking for solutions, and in particular for solutions with low cost, is called search. A search is carried out by "expanding" the states that can be achieved from a particular state. An expansion is the application of the available operators to a state in order to know the states that can be achieved from it. Notice that when the search terminates and the execution of the solution takes place, an execution of a physical action corresponds to each expansion.

It is helpful to think of the search process as building up a search tree whose nodes correspond to the states of the state space. The root of the tree is the initial state, and the leaf nodes correspond either to a goal state or to a state without successors. The search strategy is then the algorithm that is used to iteratively expand the search in some nodes of the search tree in order to find a solution.

Search strategies can be divided into two main categories. The uninformed or "blind-search" strategies explore the branches of the search tree without using any information about the nodes to expand. The informed or "heuristic-search" strategies use information about the

cost from a state to the goal to selectively expand the nodes. A review of some blind search strategies that were relevant for this research is presented in appendix 1 s. 13.1.1 (cf. also Korf, 1988).

2.1.1 Critical Observations

It is important to stress why problem solving can be viewed as a form of predictive planning (also cf. Russell and Norvig, 1995, pp. 338-339). During the searching process the selection of the operators employed to build the solutions to the problem is done on the basis of some "model of the environment". An "operator" (as has been defined previously) incorporates part of such a model as the state that the agents expect to observe after the operator is executed in a particular state. When a solution is found (and considered satisfactory) the agent can execute the single operators that make up the solution in the environment. In this way the capacity to predict is used to constrain the future course of action.

Another important aspect of problem solving is that the search strategies presented can be used only when the effect of an action (operator) is deterministic, or stochastic with a finite number of possible outcomes. In fact only in this case it is possible to explore in a systematic way all the possible sequences of actions. This aspect is particularly relevant for this research since the controllers designed and implemented here are required to be capable of dealing with a stochastic world (cf. s. 1.1.2). As we shall see chapter 8 and 9 address this problem by proposing exploration strategies that resemble the iterative-deepening search and the bidirectional search, but are adapted to deal with the stochastic outcomes of action execution.

2.2 Planning as a Searching Process: Heuristic-Search Strategies

Heuristic-search strategies use information about the cost of the cheapest path from a given state *to the goal* to determine which node to expand. A function that calculates such a cost estimate is called a "heuristic function" and can be indicated with h[s], where s is the current node. For most problems the cost of reaching the goal from a particular state can be estimated but cannot be determined exactly. Heuristic functions are problem specific. Some important heuristic-search strategies relevant for this research are reviewed in appendix 1, s. 13.1.2.

2.2.1 Critical Observations

The core aspect of heuristic-search strategies is that they exploit some information about the estimated "distance" that the states have from the goal. This aspect is particularly relevant because it resembles what is done by the reinforcement learning algorithms employed to design the neural planners presented later. These algorithms work by assigning "evaluations" to states. The states' evaluations form a gradient field over the states, and reflect the distance of them from the goal. As shown in s. 13.2.11, there is a precise correspondence between some forms of dynamic programming and some heuristic-search strategies.

S. 2.1.1 has already shown that search methods can be thought of as a form of planning. As planning processes, search methods suffer of a major drawback that stems from the fact that they treat states as "black boxes". When a search strategy selects an action (operator) in a given state, it has to consider all the possible available actions, independently of their pertinence to the state. This generates a combinatorial explosion after few planning steps (cf. s. 13.1.1). Heuristic search methods attempt to face this problem by assigning a cost or value to states in order to focus the search. Nevertheless, the combinatorial explosion is still a major difficulty for problem solving, even if we consider the sophisticated A*, IDA* and LRTA*

(cf. cf. s. 13.1.2). A possible solution to this problem is to exploit the advantages deriving from "opening up" the representation of states, goals and actions. The effects of this are so important that a new branch of artificial intelligence, that of "planning", has been formed to study them.

2.3 STRIPS Planning: Partial Order Planner

Within STRIPS planning, a planning problem can still be framed as it has been done in s. 2.1. The big innovation of STRIPS planning is a new way of representing the states, goals and actions, the "STRIPS representation" (STanford Research Institute Problem Solver; Fikes and Nilsson, 1971). The basic idea underlying the STRIPS representation is to "open up" the representation of states, goals and actions. The representation of the states is broken into a conjunction of logical predicates. Lists of predicates describe the initial state and the goal states. The representation of actions is more complex. An example of how an action (operator) is represented is shown in Figure 2.1.

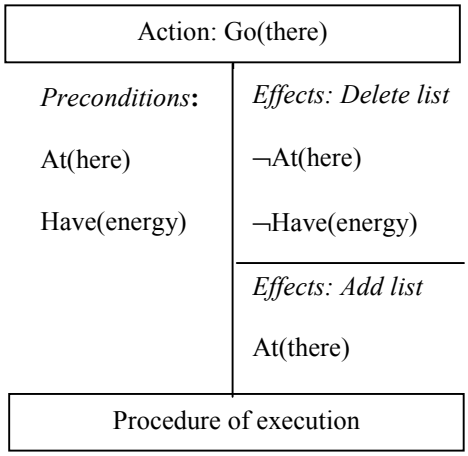


Figure 2.1: An example of action, with preconditions and effects.

In this diagram "Go(there)" is the label of the action. The "Procedure of execution" is a specification of the single steps that have to be carried out when the action is executed. An action contains some "preconditions", i.e. a list of predicates that describe the set of states where the action is applicable. It also contains some "effects", i.e. a list of predicates that describe the set of states where the action can lead if executed. The effects are divided in two parts: a "delete list", that contains all the precondition predicates that do not hold anymore as a consequence of the action execution, and an "add list", that contains all the predicates that describe the consequences of the action execution.

2.3.1 Situation Space and Plan Space

It is possible to describe a problem in STRIPS language and solve it by starting with the initial state and applying operators one at a time until a state that includes the literals of the goal is reached. A blind-search strategy could be used to this purpose. This would be a kind of planner called "situation space" planner because it searches through the space of possible situations (Russell and Norvig, 1995, p. 345). It would be also called a "progression planner" because it searches forward from the initial situation. The main problem with the situation

space progression planners is the branching factor which implies a huge size of the search space after few steps ahead.

One way to cut the search space is to search backwards. This search is called "regression" planning. This kind of search is possible within a STRIPS representation, because the actions contain enough information to regress from some preconditions of one action, to an action that has some effects that satisfy that preconditions. Regression planners are more efficient than progression planners because usually few actions can be linked to the goals, while many actions can be linked to the initial state. Unfortunately searching backward is complicated by multiple goals or, as in chess, by the excessive size of the space. In these conditions situation space planners are incomplete (they do not always arrive at the solution; Russell and Norvig, 1995, p. 345).

An alternative strategy is to search through the space of plans rather than the space of situations. This strategy implies to start with a simple incomplete plan ("partial plan"), and then to add actions to it until a complete plan is obtained. A set of operators is defined that allows passing from one plan to another plan. The operators add an action, impose an ordering to the actions of the plan, instantiate a variable, and so on. An important aspect of this strategy is that the order with which the actions are found is irrelevant.

Many planners use the principle of the least commitment, which states that one should leave things to be worked out as late as possible. The reason is that if some choices are made too early, it is more likely that they will cause a backtracking process later. The principle is also applied to the ordering of the steps. The ordering should be made as late as possible. A planner that can represent plans in which some steps are not ordered, is called a "partial order planner". The principle of least commitment also applies to variables: they should be instantiated as late as possible. Plans in which every variable is instantiated are called "fully instantiated plans".

2.3.2 Partial Order Planner

Now one of the first planners based on the principles just mentioned, the Partial Order Planner (Sacerdoti, 1977), is described here in general terms. This planner starts with an initial plan represented by the start and the goal, and tries to add actions to it. To *keep the search focused*, the planner only considers adding actions that serve to achieve a precondition that has not yet been achieved. The planner stores information about the "causal links". A causal link marks the fact that an effect of an action satisfies a precondition of another action. Causal links are "protected" in the sense that if an action can break one of them, then it cannot be inserted in the plan. The planner also stores information about "ordering constraints" between pairs of actions. An ordering constraint between two actions implies that the first action can be executed only after the execution of the second action.

At the beginning the planner selects an action that satisfies some of the preconditions that describe the goal. Unlike the search algorithms of s. 2.1, the branching factor is now quite limited. Then the planner tries to add other actions to satisfy the unlinked preconditions of the actions already inserted in the plans. If an insertion violates a causal link, the planner tries to insert an ordering constraint so that the violating action is executed before (demotion) or after (promotion) the two actions whose link would be violated. If this is not possible the planner backtracks, and other causal links / ordering constraints / new action insertions are tried out. A solution is achieved when a complete and consistent plan has been assembled. A plan is complete when all the preconditions and effects of actions of the plan are linked. A plan is consistent when the causal and ordering links have no contradictions.

2.3.3 Critical Observations

It is important to notice that in STRIPS planning, as in problem solving, the designer hardwires the model of the environment into the system by defining the actions. In fact the effects a STRIPS action represent the prediction of the effects that the planner should *expect* from the execution of the procedure of the action itself, given the preconditions in correspondence to which it is executed. This is an important simplification since the acquisition of a reliable model of the environment is a crucial step for the success of planning of autonomous agents.

We have seen that breaking the state representation into logical statements through STRIPS representations allows the planners to cut down the branching factor of the search. Unfortunately, this strategy implies two operations that are very difficult to implement with neural networks: (a) the input patterns from the sensors need to be converted into logical statements; (b) the logical statements needs to be processed in sophisticated ways to build the plan.

The first difficulty derives from the fact that we still do not have an efficient neural network model that can parse a complex input from the environment (e.g. a visual image) and build a logical or equivalent description of the "relevant aspects" of it, e.g. by describing the objects present in the image.

The second difficulty derives from the following considerations. A central aspect of STRIPS planners are the reasoning processes. These are based on sophisticated and precise mechanisms (action insertion, matching between predicates, backtracking, checks of violation of causal links, variable instantiation, etc.). Neural networks *can* implement any computational mechanism that can be implemented with a computer (McCulloch and Pitts networks are equivalent to finite automata: Rojas, 1996, p. 43). However the most interesting properties of neural networks such as generalisation, fault tolerance, and prototype extraction, can be obtained only if neural networks use distributed representations and parallel processing (cf. s. 4.4). For these reasons neural networks are not suitable to implement mechanisms that operate on local representations as those necessary for the preconditions and effects used in STRIPS reasoning.

Given these difficulties, this research has chosen to refer to a concept of state as a whole, and to use neural networks that decompose the input patterns in terms of "features" instead of logical propositions. However, the problem of how parsing the state representation to isolate "relevant" aspects so as to improve the flexibility and efficiency of intelligent systems, remains an important open problem when implementing neural planners.

2.4 STRIPS Planning: Conditional Planning, Execution Monitoring, Abstract Planning

The Partial Order Planner is based on the assumption that in a first stage a plan can be usefully generated and then it can be executed. Many environments of real problems do not allow to use this strategy because they have the following features (Russell and Norvig, 1995, p. 46):

- They are inaccessible (or "partially observable", cf. also s. 13.2.2): the sensors of an agent do not detect all the aspects of the environment that are relevant for the selection of actions.
- They are stochastic: the next state of the environment is not completely determined by the current state and action (inaccessible environments are always stochastic to the eyes of the agent).

- They are ill detected and acted upon: the information returned by the sensors and the execution of actions through the effectors are affected by noise.
- They are dynamic: the world continuously changes while the agent is deliberating and acting.
- They are continuous: percepts and commands to effectors are not limited in number, distinct, and clearly defined (in which case they would be "discrete"). Rather they are encoded with vectors of continuous values.

When the environment has some of all of these features the results of the execution of actions can be different from the expected ones. This implies a serious challenge to planners as the partial order planner. Two solutions to this problem have been proposed that imply a modification of the partial order planner: conditional planning and execution monitoring.

2.4.1 Conditional Planning

Conditional planning (Warren, 1976; Linden, 1991) deals with uncertainty by taking into account the possible situations that might arise during the execution of the plan. "Sensing actions" are used for this purpose. These actions allow the agent to gather information about the current state of the environment, so as to execute the appropriate part of the plan.

The plans are generated with mechanisms similar to the ones shown for the partial order planner. The main difference is that there are "conditional links" that can be created by inserting a sensing action to satisfy unsatisfied preconditions. When executed, a sensing action returns a "context", i.e. a statement hold to be true (or, in the more complex case of "parameterised plans", a parameter) and used to match an unsatisfied precondition. The context is inherited by the following actions until the last step.

When a sensing action is inserted, an alternative plan has to be built for the case the sensing action returns the alternative context (the negation of the first one). This is done by inserting a second finish step that is a copy of the original but differs from it for having this alternative context (this is similar to a further goal to satisfy). This second branch of the plan is built with the usual mechanisms. Notice that for each sensing action inserted an alternative branch of the plan has to be generated.

2.4.2 Execution Monitoring and Replanning

Execution monitoring offers another solution to deal with uncertainty. It is based on the idea of "re-planning": the agent formulates a plan, executes it and monitors the consequences of the actions, re-formulates and adjusts the plan when the consequences of actions are different from the expected ones. An example of this strategy is IPEM, Integrated Planning Execution and Monitoring System (Ambros-Ingerson and Steel, 1988). In IPEM planning and execution are fully integrated. IPEM has many aspects in common with the partial order planner. At a general level, the main differences are the following ones. After the execution of each action the agent carries out the following operations:

- The agent updates the internal representation of the environment. The updating of the internal representation of the environment is done on the basis of perceptions. This representation makes up the new start state for the agent.
- The parts of the old plan that are no more applicable are deleted. This is done on the basis of the new representation of the environment.
- The plan is updated through mechanisms similar to the ones showed for the partial order planner.

Notice that in its pure form, re-planning and execution monitoring are based on a rigid plan, so that any slight divergence from it triggers the re-planning process.

2.4.3 Abstract Planning

An important chapter of the planning literature is the one on *abstract planning* (Sacerdoti, 1974; Russell and Norvig, 1995, pp. 371-385). Abstract planning has been motivated by the observation that the number of "primitive operators" (the operators that can be directly executed by the agent's effectors) needed to make up a plan to solve many real problems is in the order of hundreds or thousands. In these cases the planners reviewed so far cannot find a solution in a reasonable amount of time. Abstract planning has proposed to solve this difficulty by using "hierarchical decomposition". This uses the concept of "abstract operator". An abstract operator can be decomposed into a group of steps that forms a plan that implements the operator. Eventually the decomposition can lead to have all primitive operators. This decomposition can be stored in a library of plans and retrieved when needed. To implement this strategy it has been necessary to extend the STRIPS language to allow to handle abstract operators, and to modify the planning algorithms to allow for the replacement of the abstract operators with their decomposition.

2.4.4 Critical Observations

The major drawback of conditional planning is that the number of possible conditions to take into considerations grows exponentially with the number of actions, so that for many realistic tasks it soon becomes impossible to take into account all possible situations that may arise while executing the plan. Execution monitoring has also some drawbacks, as it produces very fragile plans that require frequent replanning. Replanning can also be very expensive, for example when one gets a puncture and does not have a spare tyre. Having "spare tyres" in cars is the result of a conditional planning process rather than of a replanning process (Russell and Norvig, 1995, p. 407).

These considerations suggest that better solutions fall between the extreme cases of full conditional planning (or "universal planning") and full replanning. A desirable solution would be a planner that on one side builds up a plan that can deal with the most likely outcomes of its execution and on the other side is capable of replanning when the less likely situations are encountered. This is what the planners designed and implemented in chapter 8 and 9 do.

The issue of abstraction is a crucial issue for any planning system. Planning expresses its full power in comparison to reactive behaviour when it is applied to long-lasting tasks, and when it can operate on the basis of abstract operators. It is not easy to find a way to implement the concept of macro-operator with neural networks because it is difficult to find a neural correspondent of the process of "abstraction" in general. Chapter 11 starts to explore a possible simple solution to this problem.

2.5 STRIPS Planning: Probabilistic and Reactive Planning

The previous sections have stressed how important it is that planners are capable of dealing with incomplete and erroneous information, and with unexpected stochastic outcomes of actions by reacting appropriately to the current situation. This section reviews four planners, the BURIDAN planner algorithm, reactive planning (universal plans), decision theoretic planning, and Maes' planner, which go in this direction. The BURIDAN planning algorithm is interesting because takes into consideration probabilities, and therefore represents a bridge to

Markov decision processes analysed in s. 13.2 (cf. also the C-BURIDAN planner, Draper et al., 1994; and the MA planner, Ma and Doran, 1993). Reactive planners are interesting because they take to the extreme the idea underlying conditional planning, that of considering what to do in the case of different possible outcomes of actions. Interestingly, they have been shown to be equivalent to Markov decision processes. Decision theoretic planning is relevant because it is based on Markov decision processes, but makes use of decision trees, whose nodes are the state-variables, to represent transition functions, value functions and policies. Decision theoretic planning are based on the idea, inspired by STRIPS-like planning, that actions affect few state variables. Finally Maes' Planner is interesting because it presents a reactive planner that works on the basis of connectionist-like activation diffusion, and can be considered a bridge between the STRIPS planning systems and the connectionist "activation diffusion planners" (cf. s. 4.5.1).

2.5.1 BURIDAN Planning Algorithm

The BURIDAN planning algorithm (Kushmerick et al., 1994) assumes that the agent has incomplete information about the initial state and actions with stochastic (known) effects. The algorithm produces a plan that reaches the goal with a probability over a certain threshold, starting from a probability distribution over initial states. A state is described as a set of logic propositions. The actions are defined in terms of preconditions and effects produced with given probabilities. These probabilities depend on the value of the preconditions at the moment of execution of the actions. Given a plan, i.e. an action sequence, it is possible to compute the probability that it leads to the goal. A plan with a probability of success greater than the threshold is built by using mechanisms similar to the ones described for the partial order planner (action insertions, causal links, promotion, etc., cf. s. 2.3.2) modified to take into consideration the stochastic nature of actions.

2.5.2 Reactive Planning and Universal Plans

The idea exploited by reactive planning (Schoppers, 1987; Schoppers, 1989) is to abandon altogether any commitment to any particular sequence of actions. At execution time the current situation is classified, and the response planned for that kind of situation is performed. This can be done because the plan generates conditional advice of the kind: "IF a condition P arises AND you are trying to achieve goal G THEN the appropriate response is action A". The class of problems that the agent is capable of facing is specified only by the goal. No initial state is needed: the plan allows the agent to achieve the goal from any initial state. The plan is hence called a "universal plan". At planning time, the planner builds a universal plan by back-chaining from the goal conditions, using the effect descriptions as goal reduction operators. Back-chaining terminates when the satisfaction of the preconditions being examined or a contradiction is achieved.

2.5.3 Decision theoretic planning

Planning under uncertainty can be modelled through Markov decision processes (see s. 13.2.1). Planning problems commonly possess "structure" in the transition probabilities, value functions and policies in the sense that many states have similar or same transition probabilities / values / policy actions. If states are represented with state variables, this means that few variables are usually necessary to correctly predict / yield those transition probabilities / values / policy actions. Decision theoretic planning (Dearden, 2001; Boutilier et

al., 2000) assumes a finite number of states for each state variable (e.g. two values in the case of Boolean variables) and uses decision trees to represent states' transition probabilities / values / policy actions. This allows decision theoretic planning to exploit the structure mentioned since decision trees can abstract over state-variables.

To illustrated the key ideas of decision theoretic planning, let us consider the problem of representing the transition probabilities in a compact form. When an action is executed, few state variables ("children") might change their values each depending on the values of few other variables ("parents" of the child). The dependencies of *each state-variable* (affected by the action) from its parents can be represented by one decision tree. A node of this tree represents a parent, the edges of a node represent the values that the parent node can assume, and the leaves (assuming Boolean variables) represent the probability that the child variable to which the tree refers is true. In this data structure a branch of the tree, from the root to a leave, represents a particular set of states where the variables corresponding to the nodes of the branch have the particular values corresponding to the edges of the branch, and the variables not present in the branch can assume any value. The reward function, the value function and the action policy can be represented by three trees of the same type, but whose leaves respectively represent the probability of getting the reward, the values, and the actions, again associated to the states' regions represented by the different branches of the tree. The key idea of these type of representations is that they are particularly compact (e.g. compared to the tabular representations, see s. 13.2.8) since state representations of trees abstract over variables. This is made possible by the circumstance for which many states have the same transition probabilities / values / policy actions.

By using these tree data structures, the fundamental algorithm proposed by theoretic decision planning allows computing ("regressing") the tree that represents the expected value of performing a particular action a at the various states (the Q values, see s. 13.2.1), given a particular evaluation tree (the V values, see s. 13.2.1). To give an idea of how it works, let us consider a starting evaluation tree tree(V^0) corresponding to the simple reward tree, that says which variables are relevant to achieve the reward, and let us assume that we want to compute the Q tree tree(Q_a^1) corresponding to tree(V^0) and the execution of action a (one step regression). The idea exploited by the algorithm is that if the transition probability tree and the reward tree are available, it is possible to find out the variables which the variables important for the reward depend on. This information allows building a tree (called "probability tree") that "factors", i.e. partitions the state space into regions reached with the *same probability*, under the execution of a, and having the same value indicated by tree(V^0). Then, on the basis of the probability tree, and taking into account the discount coefficient and the reward itself, it is possible to build tree(Q_a^1). More in general, given any value tree tree(V^k) and the transition probabilities' tree, the same mechanisms allow building tree(Q_a^k). Given a value function V^k, these ideas can be exploited to produce a new value function Q_π^{k+1} that represents the value of executing the policy π for one step and receiving terminal values based on V^k. In fact for a value tree tree(V^k) and a policy tree tree(π), we can generate tree(V_π^{k+1}) that is just tree(Q_π^k). This is the key step that can be exploited in successive steps of "policy evaluation" (see s. 13.2.4). Steps of policy evaluations and policy improvements, that make the policy greedy with respect to the estimated value function, yield a "policy iteration" algorithm (see s. 13.2.4) that can be used for planning. Notice that each step of policy evaluation generates different tree(V^k) that reflect a factorisation of the state space into regions with same values. Hence, the whole algorithm both updates the structure of these trees and the values at the leaves.

2.5.4 Maes' Planner

The planner proposed by Maes (1989, 1990, 1991) represents actions with the STRIPS template. In addition each action is characterised by an "activation value", a real number used to decide if the action is triggered and executed. Three kinds of connections make up the architecture of the system:

- The signals coming from the sensors are converted into a set of predicates. Each action is connected with the sensors, and receives a bottom-up activation from them. This activation is proportional to the number of the action's preconditions that match the "sensorial" predicates.
- The system has a set of goals to satisfy expressed in form of predicates. Each action is connected with the goal system, and receives a top-down activation from it. This activation is proportional to the number of the action's effects that match the goal predicates. Eventually the activation from the goal is also proportional to the importance given to the goal at the moment, e.g. if the goal is ingest_food and the agent is very hungry. In this way the system is capable of dealing both with "cognitive goals", e.g. represented by predicates, and "motivations", e.g. represented by the intensity of "needs".
- Actions are also connected by bidirectional connections between them. The intensity of the connections between two actions is proportional to the number of their preconditions and effects that match.

The system works as follows. The sensors and goals send their activation to the actions with which they are connected. The activation of each action decays according to a certain rate, and is transferred (in a certain percent) to other actions both through the backward and forward connections. As a consequence the actions' activation accumulates while time elapses. When all the preconditions of an action are satisfied and the action's activation overcomes a certain threshold, the action is triggered and executed.

2.5.5 Critical Observations

The BURIDAN planning algorithm and other planners that try to incorporate the probabilities of different outcomes in their reasoning process have the strength of focussing planning on the most likely outcomes of the plan execution. This is an important feature that will be incorporated in the planners designed and implemented in chapter 8 and 9.

"Universal plans" were developed as a general scheme for reactive planning. However they turned out to be a rediscovery of the idea the "policy" of the Markov decision processes, presented in chapter 3 (Russell and Norvig, 1995, p. 411). Markov decision processes are at the core of the planners designed and implemented here. It is very important to mention that universal planners have an important drawback (Ginsberg, 1989). This drawback affects any kind of planner that is based on "compilation" of reactive plans before action execution ("reactive planning"), so it is also relevant for the planners proposed here, based on Dynamic Programming and Markov decision processes. The drawback is that given a goal, a universal plan has to be capable of deciding what to do in *any* situation that may arise during the plan execution. This implies that the situation→action rules to prepare increase exponentially with the size of the problem space. After all the biggest advantage of planning versus reactive behaviour is precisely that planning allows to decide what to do when it is actually needed. Reactive planning ignores this point. For example even a simple simulated robot as the one considered here has 50 binary sensors, so the number of the possible situations that it can sense is huge, 2^{50}. This causes major difficulties both to generate and to store all the possible situation→action conditions: the time and space complexity increase exponentially with the

problem space. Three solutions have been proposed to face this problem: (a) function approximation (Sutton and Barto, 1998, p. 193; cf. s. 13.2.8); (b) focusing the planning activity on the relevant areas of the problem space (Sutton and Barto, 1998, p. 246; cf. s. 3.4); (c) replanning when necessary (cf. s. 2.4). These solutions are exploited in the planners proposed in chapter 8 and 9.

Decision theoretic planning represents an important bridge between classic STRIP-like planning and planning based on Markov decision processes. It offers an important extension of dynamic programming to the cases where states are represented by state-variables ("features"). The theoretical analysis behind the algorithms proposed by decision theoretic planning gives an important contribution to explaining how different features play different roles in the prediction of values, and how this "structure" of the problems might be exploited through space abstraction. On the other side, decision theoretic planning implements abstraction over features by complex mechanisms that work on the single features so that they might not scale well to cases with many features. For example, for each policy evaluation step the evaluation tree needs to be modified on the basis of the previous evaluation tree, the reward tree and the transition function tree. This means that the data structure storing the values needs to be modified. This approach might be inefficient compared to alternative algorithms and data structures that work on features in parallel and "weight" the importance of features for values' prediction by only changing parameters. For example in the case of neural networks usually only the weights are changed, while the data structure, i.e. the network architecture, is not.

Maes' planner implements planning by spreading the activation backward from the goals and forward from the sensors through the network of actions. This activation generates chains of actions highly activated that connect the two "extremes" of the current state and the goal. These chains are then executed according to the satisfaction of their preconditions and level of activation. Maes' planner is interesting because it is quite original and because it is taskable (cf. s. 5.1). However, it is important to notice that the principle underlying its functioning is very similar to the one used by activation diffusion planning (cf. s. 4.5.1). The fact that Maes' planner considers operators instead of states as activation diffusion planning does, is not very relevant. In fact it is possible to find close correspondences between the two systems. The backward activation has a role similar to the activation diffusion of activation diffusion planning, i.e. activating the operators with a decreasing intensity on the basis of their distance from the goal. The forward activation causes an accumulation of activation of the operators that corresponds to the current state. This is equivalent to the triggering of the operators in activation diffusion planning. For these reasons Maes' planner can be considered a sort of dynamical implementation of activation diffusion planning (cf. Tyrrell, 1994, for the limitations of Maes' planner).

2.6 Navigation and Motion Planning Through Configuration Spaces

The planning methods reviewed in this chapter, but also those reviewed in chapter 3 as dynamic programming, are based on discrete representations of states. This causes difficulties when they are applied to robotic problems as navigation or motion planning. In fact these problems imply continuous state variables. Configuration spaces are general mathematical tools that can be used to implement planning with states described by continuous state variables (Yoshikawa, 1990; Russell Norvig, 1995, pp. 790-808).

The state of a robot can be described with k real values. Each value indicates the state of one of the k degrees of freedom of the robot, for example the (x, y) position of the robot in two-dimensional space, or the angles formed by the links of the robot's arm. The k values can

be considered as a point in a k-dimensional space C, called the "configuration space" of the robot. If O is the set of points in C for which any part of the robot bumps into or "is inside" an obstacles, the set difference F = C - O is called "free configuration space". Assuming an initial point s_1 and a goal point s_2, the robot can safely move between the corresponding points in physical space if and only if there is a continuous path between s_1 and s_2 that lies entirely in F. Planning through configuration spaces implies finding one path that satisfies this condition and also some efficiency criterion, such as the shortness of the path.

A planning method based on configuration spaces is called "cell decomposition". This method is based on the identification of some "cells" within F. The cells are contiguous. Planning is implemented as a discrete search problem on these cells. An alternative method is called "skeletonization". This method collapse the configuration space into a one-dimensional subset called a "skeleton". A skeleton is a graph with a finite number of vertices connected by links. Both vertices and links lie within F. A "path" is planned by using a (graph) search method applied to the skeleton. If the start and/or the goal points do not lie on the skeleton, it has to be possible to easily compute short path segments between them and the nearest point on the skeleton.

3 Markov Decision Processes and Dynamic Programming

It has been suggested (Sutton and Barto, 1998, p. 66 and p. 89), that Markov decision processes (Puterman, 1994) are the best framework with which reinforcement learning problems can be presented and investigated, and that dynamic programming (Bertsekas, 1995) furnishes sound theoretical foundations to it. Appendix 2, s. 13.2, presents the mathematical details of the parts of Markov decision processes, reinforcement learning and dynamic programming that are the starting point for the controllers presented here, or are important for the issues analysed in the following chapters. This wide appendix is justified by the fact that the issues presented in it, though available in some textbooks and articles, are quite specialist, so the reader might not be familiar with some of them. The reader that is already familiar with the material presented in appendix 2 is invited to consider only the mathematical symbolism used in it since this symbolism is used throughout the thesis.

This chapter will focus on few relatively new aspects of Markov decision processes, reinforcement learning and dynamic programming, that are quite central for this research. S. 3.1 will present the restricted class of Markov decision process considered in this research. S. 3.2 will presents some critical observations on dynamic programming and its relationship with heuristic search. S. 3.3 will present the Dyna architectures, and in particular the Dyna-PI architecture, that are the starting point for the controllers studied in the next chapters. Finally s. 3.4 will analyse some techniques that allow reinforcement learning controllers to focus planning activity on restricted areas of the state space.

3.1 The Problem Domain Considered Here: Stochastic Path-Finding Problems

The controllers presented in this thesis apply to a restricted class of Markov decision problems called "stochastic shortest-path problems" (Barto et al., 1995). Given that this research does not put a stress on the optimality of the algorithms proposed, the expression "stochastic path-finding problems", or simply " path-finding problems", will be used henceforth. What will be required for the algorithms is to yield a "satisfactory" performance. As an empirical rule, the performance will be judged as satisfactory if the "length" of the solution (path) found is much shorter than the solution found by a random walk, and about twice or less the size of the optimal solution in the absence of noise during action execution. An approximate measure of the optimal solution can be easily computed knowing the size of the simulation scenario, the size of the single moves of the simulated robot, and the start and goal states in the scenario. All these elements will be furnished in the second part of the thesis.

In stochastic path-finding problems, the reward is equal to 1 (or, in general, to a fixed value) for a *unique* goal state s_g, and equal to 0 for all other states:

$$MR[s_g] = 1 \qquad MR[s] = 0 \qquad \forall\, s \neq s_g \qquad \text{Eq. 3.1}$$

where MR[.] is the reward function.

In the typical simulation, the simulated robot aims at finding a policy that takes it from an initial state s_i to the goal state s_g with the minimum number of steps. When the simulated robot reaches s_g it is set again at s_i and a new trial begins (these are also called "episodic tasks", cf. Sutton and Barto, 1998, p. 60). In some other experiments when the simulated robot reaches the goal it is set at a different initial state. It is assumed that $\mathbf{x}_g$, the vector of signals returned by the simulated robot's sensors at the goal state, is known by the simulated robot (e.g. stored in a suitable data structure/memory), while $\mathbf{x}_i$, corresponding to the initial state, is directly perceived. Notice that when the robot stops acting to *re-plan* (cf. s. 8.3.2), the new start state $\mathbf{x}_i$ is the current one.

A second class of stochastic path-finding problems considered in the thesis implies multiple goals achieved asynchronously. "Asynchronously" here means that at each moment the simulated robot is pursuing only one goal. In the thesis two variants of a multi-goal navigation stochastic path-finding problem are considered. In the first variant the simulated robot pursues the same goal several times, and is assigned another goal only after several successes with the first one. At each success the simulated robot is set at a start state or at some other state randomly drawn. In the second variant of the problem, a new goal is assigned to the simulated robot after each success. This new goal is randomly drawn from a set of goals. In this case there is no need to reset the simulated robot in new state: the state of the goal just reached is the starting state from which the simulated robot pursues the new goal. As we shall see, these variants of the problem are considered in different part of the thesis to investigate different aspects of the controllers.

Now some aspects of the multi-task problems are formalised because they are particularly important to clarify the concept of "taskability" (cf. s. 5.1). At each moment the simulated robot pursues the goal state $s_g \in S_g \subset S$ assigned to it, where S_g is the subset of states used as goal. The simulated robot's task is to reach the goal from a start state $s_i \in S$. In the case of stochastic policies, finding a solution to the problem requires finding a suitable mapping from the current state, goal and the available actions $a \in A$, to the probabilities of such actions (cf. s. 13.2.3):

$$(S \times S_g \times A) \rightarrow [0, 1] \qquad \text{Eq. 3.2}$$

In the case of deterministic policies, finding the solution of the problem requires finding a suitable mapping from the current goal and state to suitable actions:

$$(S \times S_g) \rightarrow A \qquad \text{Eq. 3.3}$$

It is very important to notice that when the simulated robot pursues only one goal during its existence, then the mappings to learn are the ones of Eq. 13.4 or Eq. 13.6. These mappings are much simpler than the mappings of Eq. 3.2 and Eq. 3.3 thanks to the fact that they do not require considering the information about the goal pursued. In the case of stochastic policies the simulated robot has "simply" to assign a proper probability distribution to the actions, and to select one action on the basis of this distribution, on the basis of the state. In the case of deterministic policies, the simulated robot has to select a proper action on the basis of the state. This fact is very important because in some circumstances it renders planning agents much more efficient than reactive agents. In fact in order to solve multi-goal tasks reactive agents have to learn the complex mapping of the kind $S \times Sg \times A \rightarrow [0, 1]$ (or $S \times Sg \rightarrow A$) i.e. a mapping for each possible goal pursued. Instead, planning agents can once and for all learn and store knowledge in the model of the environment, that is independent of the particular goal pursued. Then they can dynamically build the necessary mapping $S \times A \rightarrow [0,$

1] (or S → A) on the basis of this knowledge. This is the core idea underlying the concept of "taskability" (cf. 5.1). Chapter 8 will show how it is possible to modify the basic Dyna-PI architecture to make it fully taskable.

Before closing this subsection, it is important to stress that the one-goal problems considered here are more restricted than the ones considered in classic artificial intelligence planning. In fact the planners of this literature usually specify the goal as a set of abstract properties that the desired state has to satisfy. This implies that the goal is actually a set of states. Given the difficulties that current neural-network models encounter in dealing with abstract representations of states, this research focuses on the simple case where a goal corresponds to a single state (with the tolerance of some noise).

3.2 Critical Observations on Dynamic Programming and Heuristic Search

This section presents some critical observation on dynamic programming and its relationship with heuristic search analysed in Appendix 2, s. 13.2.9, 13.2.10, and 13.2.11. One of the reasons dynamic programming (cf. s. 13.2.9 and 13.2.10) is an important machine learning method is that it uses policies instead of plans. This allows agents to deal with noisy environments where the effects of the actions are stochastic. In fact, whatever the effects of the previous action executed are, the selection of the new action will be optimal with regard to the current state, and will not be committed to any previous decision (cf. s. 5.3 for a summary of the advantages and disadvantages of plans and policies).

Trial-based real-time dynamic programming (cf. s. 13.2.11) represents an improvement of the idea of policy because it allows the agent to focus the backups on few relevant states, and to prepare "partial policies", i.e. policies that work well only for few states (a random action is used for the other states). However it is still not fully satisfactory because for certain environments *too many* state could be reached under the execution of the optimal policy, even if these events occur very rarely. For example, in the simulation scenario used in the following chapters the effect of one action is a movement toward a destination point, but the actual point reached has a *Gaussian* distribution around the destination point. This means that in theory all the possible points of the arena could be reached with one movement, in practice this means that the points far from the destination point are never reached during the simulations. So why should the system worry about them? The solution to this problem is to take these probabilities into consideration in some ways. This is the solution adopted in the controllers presented in the next chapters. There the system "simulates" the possible effects of action execution, i.e. the states reached given the "noise" of the policy and the noise of the effects of action, and prepares a policy only for states that are more likely to be visited during the use of the policy itself. Incidentally, notice that the noise of the actions' effects are actually not "simulated" by the system when planning because the model of the environment is deterministic. However, this is a problem caused by the particular implementation of the model presented here, not by the general functioning of the planner.

The equivalence between trial-based real-time dynamic programming and learning real-time A* (cf. s.13.2.11) is important because it builds a bridge between the two important fields of heuristic search and dynamic programming. Notice that it has been possible to show a full equivalence between the two techniques because learning real-time A* *learns* the heuristic with experience (it does not necessarily require a heuristic a-priori as the majority of the other heuristic-search methods do, cf. s. 13.1.2).

Most importantly, the equivalence shows that dynamic programming and heuristic search are based on a gradient field of heuristic values or evaluations over the states. In s. 5.2 it will be shown that "activation diffusion planning", a kind of planning mechanism often used in

neural planners, is itself based on the idea of gradient field over states. This result will be used to suggest that the idea of gradient field unifies many approaches relevant for neural-network planning. In turn this will be shown to be a positive result for neural-network planning since the idea of gradient field can be implemented in a straight forward way with neural networks.

3.3 Dyna Framework and Dyna-PI Architecture

Dynamic programming implies to carrying out *full backups* (cf. s. 13.2.9 and Figure 13.1). Dyna architectures (Sutton, 1990; "Dyna" stands for "dynamic programming") integrate the idea of dynamic programming, i.e. computing the state-evaluation function on the basis of a model of the environment, and the idea of reinforcement learning methods, i.e. computing the state-evaluation function on the basis of *sampling backups* (cf. s. 13.2.9 and Figure 13.1; Sutton and Barto, 1998, p. 243). Sample backups bring faster convergence because they explore states in depth rather that in width while updating the evaluations of states. This favours quick "propagation" of evaluations from the states with high evaluations/rewards towards other states (Sutton and Barto, 1998, p. 245-246).

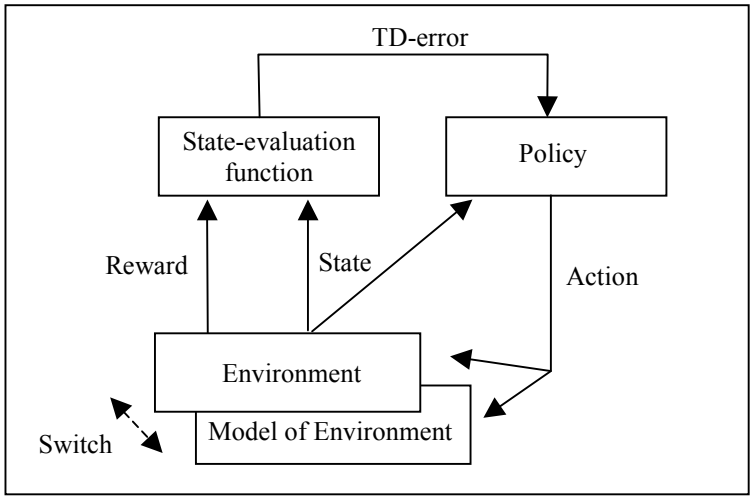


Figure 3.1: A typical Dyna architecture. The reactive part of the controller (the state evaluation-function plus the policy) can be trained either through a model of the environment, or through experience in the environment.

The architecture of an agent with a dyna architecture is represented in Figure 3.1 (cf. Sutton, 1990). When the reinforcement learning method used is the actor-critic method, the architecture is called "Dyna-PI". "PI" stands for "policy iteration", the algorithm at the core of the actor-critic methods, cf. s. 13.2.6. Another dyna architecture, based on the popular Q-learning (Watkins and Dayan, 1985), is "Dyna-Q" (Sutton, 1990).

The components of the architecture are the state evaluation function and the policy structure (as in a standard reinforcement learning agent) plus a model of the environment as defined in s. 13.2.1 (the probability transition function plus the reward function). The functioning of the architecture is simple. The architecture can function as a normal reinforcement learning architecture, in which case the updating of the evaluations and policy are based on real experience. Alternatively, it can function as a *planner*, in which case the

updating of the evaluations and policy are based on *simulated experience* generated through the model of the environment.

In general the simulated experience is generated as follows (as an example, the case of the actor-critic methods is considered, cf. s. 13.2.6). Given a state s_t, the policy selects an action a_t. This action, together with the state s_t, is sent to the model of the environment, that returns the expected new state s_{t+1} and the reward r_{t+1} as consequences of that action. The states and the selected action can be used to train the evaluation function and the policy in the same way it is done with standard reinforcement learning. This means that s_t and s_{t+1} are used by the evaluation function to return $V'^{\pi}[s_t]$ and $V'^{\pi}[s_{t+1}]$. In turn these values, together with the "predicted" reward r_{t+1}, are used to compute the error e_t. Finally this error is used to update both the evaluation function and the policy (cf. s. 13.2.4, 13.2.5, and 13.2.6).

The states as s_t used to generate "simulated experience" can be generated in different ways. For example they can be generated at random from the set of possible states. Or, given an initial state, they can be generated by using the states predicted by the model of the environment, plus the actions selected by the policy, to predict other states in sequence (cf. trajectory sampling, s. 3.4).

If the evaluation function and policy are trained for some time through simulated experience generated by using the model of the environment, they improve their competence to evaluate and act in the environment. For this reason this training, together with the use of the model of the environment, can be regarded as a form of planning.

3.3.1 Critical Observations

Dyna-PI architectures are type of "compiling planners" (cf. Mitchell, 1990) because the outcome of planning is not directly used for acting, but is store in the reactive components (memory structures) of the system. When the system passes from planning to action execution, what it does is simply to act on the basis of the reactive components. The alternative is a planning system where the outcome of planning is kept in a temporary memory structure, is used to act, and then is eventually incorporated into the reactive components to be retrieved at a later time when a similar situation is encountered.

Compiling planning has the advantage that the outcome of planning is directly stored in the "most usable" form, i.e. into reactive components, and is permanently incorporated in the skills of the system. This produces the nice property of Dyna-PI architectures for which both planning (if based on a good model of the environment) and acting, contribute to improve the evaluations and the policy, no matter when they are executed. Compiling planning has also the advantage that no extra memory structures are needed to temporarily store the outcome of planning.

Unluckily, compiling planning has also some disadvantages. In particular if the planning process generates several possibilities of low quality before arriving to a good policy, and the reactive memory structures are used as "working memory", the contents of the skills could be damaged by the planning process. This is particularly important if neural networks are being used, because these are prone to suffer of "catastrophic interference": the temporary storage of any noisy/useless information tends to damage the contents of the existing memory (s. 4.4.1). In this case it would be better to store temporary policies in buffer memory structures, and then transfer only some of them to the long-term reactive memory structures at a second time, after they have proven their quality in the environment.

3.4 Prioritised Sweeping and Trajectory Sampling

Prioritised Sweeping. We have just seen that any simulated experience generated improves the evaluations and policy, if the model of the environment is enough accurate. However the methods used to generate such simulated experience greatly influence the convergence speed of the process, because it can focus the updating of the evaluations and policy on "relevant" or "marginal" regions of the state space.

Consider problems where states are represented as whole discrete entities, i.e. not by state variables or features (cf. s. 13.2.8). In these cases "prioritised sweeping" can be used (Moore and Atkenson, 1993; Sutton and Barto, 1998, p. 239) to improve the speed with which the evaluation function and the policy are updated.

Prioritised sweeping is a method based on a relatively simple idea. For example consider the popular case of Q-learning (cf. s. 13.2.4 and 13.2.5): if the evaluation of one state-action pair changes a lot, then the evaluation of its "predecessors" (the state-action pairs that have led to the state at least one time in the past) would change a lot if updated. To exploit this idea, the system keeps a queue of the Q values to be updated in decreasing order of "priority". The priority of a state-action pair is defined on the basis of the amount of change that its Q value would undergo if updated, and on the basis of the transition probability from this state-action pair to the successor. At each cycle the state-action pair with the highest priority in the queue is updated. Then the priority of its predecessors is computed, and the state-action pairs with priority over a certain threshold are inserted in the queue. At each cycle a given number of these updates (and predecessors' insertion in the queue) is carried out.

Although the original formulation of prioritised sweeping assumed whole states, further research has proposed alternative formulations capable of working with state variables (Wiering et al., 1998; Dearden, 2001). These alternative formulations keep trace of the dependencies between the single state variables. On the basis of these dependencies, prioritised sweeping is applied to the "predecessor" variables closely linked to those variables whose value is significantly updated.

Trajectory Sampling. Another interesting way to focus search on relevant regions of the problem space is "trajectory sampling" (Barto et al., 1995; Sutton and Barto, 1998, p. 247). In stochastic path-finding problems this sampling implies that planning starts from an initial state, for example the current state, and generates a simulated sequence of states through the *current policy* and model of the environment, until the goal state is reached. At the same time it updates the evaluations and policy for the states encountered along the way. Sutton and Barto (1998, pp. 247) report about a simple abstract problem used to compare trajectory sampling with a "full sweeping" process, i.e. a systematic repeated updating of the evaluations of *all* states. The experiment used undiscounted episodic tasks generated randomly as follows. For each of the $|S|$ states, two action were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state action pair. On all the transitions there was a 0.1 probability that the episode ended. The expected reward on each transition was chosen randomly in a Gaussian distribution with mean 0. The performance was measured in terms of the evaluation of the initial state. The results of the experiment have shown that trajectory sampling has an advantage and a disadvantage versus a "full sweeping" process in terms of speed of learning. The advantage is that it ignores uninteresting parts of the space. The disadvantage is that when the evaluations and policy become accurate, the same old parts of the space are backed up over and over without improvement of evaluations and policy.

The idea of trajectory sampling is very important because it is exploited in the planning controllers designed and implemented in the following chapters. However, it should be noticed that its formulation, as presented in Sutton and Barto (1998, pp. 247), is incomplete if used for path-finding problems. For example, it does not deal properly with the danger of getting stuck in dead-ends, or with the problem of deciding when to plan and when to act. Chapter 8 and 9 will propose and implement some solutions to solve these problems.

3.4.1 Critical Observations

The strength of prioritised sweeping is largely caused by the fact that it executes backups *backward* from the states with high rewards associated towards other states. For example in the case of stochastic path-finding problems it executes backups backward from the goal state. In fact the predecessors of the states from which a high reward has been obtained receive a high priority, so they are likely to be updated. Chapter 9 proposes and implements a planning controller (neural bidirectional planner) that exploits an alternative idea: the backward "propagation" of evaluations. It process works by generating sequences of states backward from the goal and forward from the current state.

The strength of trajectory sampling derives from the fact that it focuses the search on the states that the policy visits with a high probability. This seems a powerful idea that can be implemented in a relatively easy way. The planners proposed and implemented in this thesis are all based on this idea. The forward planner implemented in chapter 8 generates trajectories of states starting from the start, so it focuses the search around it. The bidirectional planner implemented in chapter 9 generates trajectories both from the goal and from the star, so it focuses the search around them.

4 Neural-Networks

In this chapter the neural networks and algorithms used as "building blocks" to design the planners presented in the following chapters are introduced. These are the classic "feed-forward networks", trained with the "backpropagation algorithm", and the "mixture of experts networks", trained with an algorithm specifically designed for it.

4.1 What is a Neural Network?

What is an (artificial) neural network? It is difficult to give a precise definition, since a great number of models have been proposed in the literature (cf. Haykin, 1999, for a wide review). However, it is possible to identify some defining traits and some desirable traits of neural networks that are particularly significant for this research. The *defining traits* are the following ones:

- Architecture. The architecture of a neural network consists of a set of units linked with connections. The units use the connections to exchange quantitative signals in a parallel fashion.
- Processing. Each unit of a neural network is a simple device: it takes the signals from some connected units, processes these signals in a simple way, and as a result sends a signal to other units.

The *desirable traits* are the following ones.

- Local learning. The network can "learn", i.e. some units change their processing properties in time. If learning is present, it takes place on the basis of information available locally to the unit. For example the "weights" of the connections of a unit are updated only on the basis of the activation of the units directly connected with it. Local learning is desirable since it does not require complicated architectures to carry learning signals to the target units.
- Distributed representations. Information is represented in a distributed fashion on many units and weights. This rules out the possibility of having "local representations" where one weight or the activation of one unit represents a whole significant chunk of information. A consequence is that the system cannot work as a "finite automaton" (cf. Rojas, 1996, p. 43) processing information in a logical way. An example of this would be: "If unit x is active it means that the environment state is s, if unit y is active it means that the system is pursuing goal s_g. If unit x and y are active then unit z is activated, and this means that action a is selected". Distributed representation of information is desirable since is allows having the generalisation property of neural networks (see below).
- Noise. Each component of the system is capable of tolerating some amount of noise ("fault tolerance"). This property usually relies on the fact that the system uses distributed representations. Noise can even be an important ingredient of the architecture and functioning of the system. This property is desirable since it implies that the disruption of some components of the system is accompanied by a gradual degradation of performance ("graceful degradation", Rolls and Treves, 1998, p. 30).

- Local learning, distributed representations and noise tolerance are also desirable because they increase the biological plausibility of neural networks.

4.1.1 Critical Observations

The design of the controllers presented in the next chapters has attempted to build neural networks that posses all the traits illustrated in the previous sections. This has not always been possible, in particular for two aspects of the controllers.

The first aspect concerns the learning algorithms used. Given the computational needs met in designing the controllers, it has been necessary to use learning algorithms that violate the third requirement and do not have a local nature. For example, this is the case of the learning algorithms used to train the mixture of experts networks (Jacobs et al., 1991; cf. s. 13.3.2) used in chapter 7 and 10, or the use of the backpropagation algorithm (Rumelhart et al., 1986; cf. s.13.3.1) in chapter 7: these are not local learning algorithms. The adoption of these algorithms has been necessary because local learning algorithms currently known have a limited computational power (cf. Rolls and Treves, 1998, p. 23-94).

The second aspect has been the control of the flow of information among the different parts of the system. With some effort it has been possible to transform this algorithm into a neural network that possessed the first two traits (this has been done for a simplified version of the algorithm illustrated in Figure 8.3). This has not been a surprise since a constrained class of neural networks, the networks of McCulloch and Pitts, are equivalent to a "finite automaton", i.e. they can be used to execute any kind of computation that can be executed by a computer (Rojas, 1996, p. 44). The problem has been that the outcome of these efforts was a neural architecture that did not posses the fourth and fifth traits, the use of distributed representations and noise tolerance. As a consequence, the decision has been taken not to transform the algorithms controlling the flow of information between the different neural components into a neural network, but to leave them in the form of code.

This experience shows that there are some aspects of planning that require some precise control mechanisms that resist an implementation with neural networks that satisfy the "desirable traits" listed previously. Some examples of these aspects are these: "switching" between acting and planning, controlling the flow of information between neural modules, and switching different neural modules on and off. It remains an open question if there are other forms of planning that *do not* require these precise control mechanisms, or if they are necessary with any kind of planning. In the later case, the use of logical finite-automaton like neural modules would be unavoidable.

4.2 Critical Observations: Feed-Forward Networks and Mixture of Experts Networks

Feed-Forward Networks and Backpropagation. Feed-forward networks and the backpropagation algorithm usually used to train them are illustrated in appendix 2, s. 13.3.1. As mentioned, these neural networks, and the ones reviewed in s. 13.3.2, have been used as building blocks for the neural planners designed and implemented in the next chapters. One reason for which this has been done is that these neural networks have been extensively studied, so their properties are quite well known. This made it easier to design more complex controllers, as neural planners, on the basis of them. It also made it easier to understand the overall behaviour of these complex controllers. Another reason for which those neural networks have been chosen as building blocks, is that quite effective learning algorithms have

been proposed for them. As a consequence, if a system is built on the basis of them, its learning capabilities can be quite effective.

A fundamental quality of these neural networks is that they have a feed-forward activation. This means that the signals coming from the input propagate towards and generates the output in a direct fashion. In the introduction it has been mentioned that predictive planning is likely to require "looping" neural systems, i.e. neural systems where the output of some component networks is fed back into the system itself. This feedback is necessary because predictive planning requires that the prediction of the effects of actions' execution, produced by some networks, be used to change the way the system acts in some circumstances, and this "way" is itself "expressed" by other component networks of the system. As we shall see, in the planning architectures designed and implemented here, these looping processes are implemented with feed-forward networks that feedback their output into the input of other feed-forward networks. In particular the basic architecture used in this research to implement planning is showed in Figure 4.1.

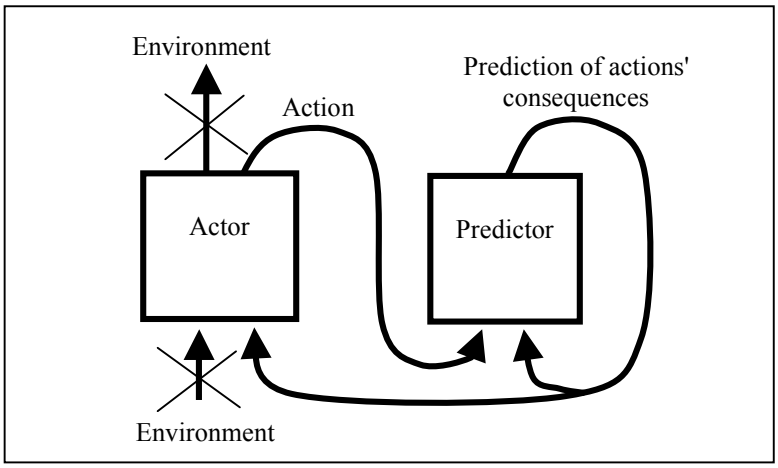


Figure 4.1: Example of "looping" neural architecture used to implement planning. The single component neural networks are feed-forward networks.

This graph shows that, while planning, the actions "programmed" by a suitable network (the "actor") are not executed in the environment, but are sent to another network that incorporates the model of the environment (the "predictor"). This network predicts the "consequences" of these actions. These consequences are fed back to the actor, which is temporarily disconnected from the environment. They are also fed back to the predictor itself to produce the successive prediction.

Closing the issue of "looping" neural networks, it is important to mention why the recurrent neural networks proposed by Elman (1990) are not suitable to implement planning. These are feed-forward neural networks where the activation of the hidden layers is fed back into the input. These networks are an effective way of implementing short-term memories, for example to use the information contained in few recent input patterns, together with the current one, to select the current action. These networks might have been used to substitute the feed-forward networks used here in order to have this type of memory. However, and this is the point stressed here, they could have not been used to substitute the overall architectures presented here where looping was necessary to learn to generate predictions and feed them back into the system's input units.

The backpropagation algorithm is a supervised learning algorithm. This implies that to function it needs a "teaching output" that has to be furnished by an external "teacher". This would seem to limit the applicability of this algorithm to reinforcement learning systems because these do not have a teaching output signal to use for training. However in chapter 7 we will see that in the case of the actor-critic reinforcement learning *it is* possible to employ the backpropagation algorithm for the evaluator. In fact, in the case of the evaluator it is possible to build an error signal of the kind shown in Eq. 13.21 (analogously it would be possible to use a formula of the kind shown in Eq. 13.17 to implement Q-learning). It is also possible to use the backpropagation algorithm for the actor. In fact it is possible to use 1 and 0 as teaching signals to respectively increase or decrease the merit (cf. s. 13.2.6), and hence the probability, of the action that has been selected. The merits of the other actions are left as they are as no information about how to change them is available. It is important to notice that, at the level of the actor, previous research (cf. Lin, 1992) has shown that when a localist representation of the actions is used (one unit for one action merit or Q value), it is better to use a different neural network (e.g. a three layer feed-forward neural network) for each unit. In fact this avoids interference problems and speeds up the learning process. Notice that the localist representation of actions has been used throughout this research and is also used in the majority of reinforcement learning studies.

Mixture of Experts Networks. As we shall see in chapter 7, one of the strengths of this architecture is its capacity to avoid interference problems in the case of multi-goal tasks. This architecture has this capacity because it uses different neural modules to deal with different regions of the input-output space. This implies that when the weights of one module are updated, the weights of other modules are not disrupted.

In the previous section we have see that, in general, it is possible to use a supervised learning algorithm, such as error backpropagation, both in the case of the evaluator and the actor. Unfortunately, while it is possible to use a mixture of experts network for the evaluator, it is not possible to use it for the actor. In fact the algorithm used to train this architecture needs the error signal of all the output units (cf. Eq. 13.50). In the case of the actor, this error signal is available only for the action selected, so the learning rules of Eq. 13.51 cannot be applied. As a consequence in chapter 7 and 10 the mixture of experts architecture is used only to implement the evaluator, while a new hierarchical modular architecture is used to implement the actor.

4.3 Neural Networks for Prediction Learning

This subsection reviews some works that have used neural networks as "predictors", i.e. as models of the environment. Nolfi and Tani (1999) present a work where a simulated robot follows the wall of a room (stereotyped behaviour) and has to learn to anticipate the future input patterns (vision of the wall). The neural architecture they use to this purpose is based on a hierarchy of three-layer feed-forward networks that learn through the backpropagation algorithm. The network of the lower level takes the current input pattern as input, and learns to produce the following input pattern as output. The activation of the hidden units is sent to a Kohonen network (Kohonen, 1982) capable of classifying the input into broad categories. The classification made by the Kohonen network is sent into a second level network that learns to predict how this categorisation changes. The authors show empirically that the second level network is capable of predicting some regularities in the input flow (for example which room is entered), that are abstract in terms of time and details, and that the first level network is not capable of predicting.

Sequences learning problems are also tackled by Schmidhuber (1992). The neural networks he proposes are based on this principle. A network is used to predict its next input on the basis of the previous ones. Since only unpredictable inputs carry new information, a second network, working on a slower and self-adjusting time scale, takes as input the inputs that are unexpected by the first network. The performance of this system is superior to the performance of other systems.

Duckett and Nehmzow (1999) present some experiments where a robot builds a graph-based map. A neural network is used to predict the probability of existence of open spaces (not yet in the map) in a given direction from a position occupied by the robot. Lee et al. (1998) present a work where neural networks are used to build a model of the activation of the sensors of a robot in an office, on the basis of the x, y co-ordinates of the position occupied.

4.3.1 Critical Observations

The works just reviewed are quite relevant for the work presented here. This may appear strange because in some cases they focus on the prediction of the behaviour of aspects of the environment *independent of the agent's actions*, or in other cases they focus on the prediction of the consequences of a stereotyped behaviour of the agent. In the later cases the agent always chooses the same action in correspondence to each state, so these cases are equivalent to the former ones.

Nevertheless, the models reviewed are relevant for prediction within reinforcement learning. In fact, as we shall see in chapter 8, reinforcement learning systems usually use a small number of primitive-actions. This makes it possible to build a prediction model *for each action.* In this case the single prediction model (neural network) is trained and used to predict the consequences of *one action only*. This creates a situation equivalent to the situations considered in the literature just reviewed where the agent is predicting the consequences of a stereotyped behaviour.

4.4 Properties of Neural Networks and Planning

What are the characteristics of neural networks that may produce interesting results when they are used for planning? Among the most appealing properties of neural networks there are the following ones: generalisation (and noise tolerance, that is closely related with it), prototype extraction, learning, parallel processing. Now these properties are considered in detail and their relevance for planning explained. Notice that the discussion that follows focuses on the particular kind of neural networks used here, presented in appendix 3, s. 13.3.1 and 13.3.2. These are feed-forward hetero-associative networks. Feed-forward hetero-associative networks are networks where the signal flow travels in one direction from the input to the output units, and that are capable of learning to associate an output pattern to an input pattern (Rumelhart and McClelland, 1986).

4.4.1 Generalisation, Noise Tolerance, and Catastrophic Interference

Neural networks' generalisation property is relevant for this research for a number of reasons. The introduction set the constraint that the controllers designed here should be capable of guiding a simulated robot interacting with a noisy environment. The interaction with the environment through sensors and effectors can cause a combinatorial explosion of possible sensory and motor configurations that cannot be dealt with one by one. Neural networks are capable of dealing with this problem because, thanks to their generalisation capacity, they

discover "common structure" between different sets of input output associations, so that they can compress information into the distributed representations based on the weights (cf. s. 13.3.3).

The problem of noise is correlated with the previous problems. The sensorial apparatus of robots returns patterns that are affected by noise. The generalisation property allows neural networks to deal with this problem because they can be trained with several input patterns and still be capable or responding appropriately to versions of them corrupted by noise.

Unluckily, generalisation has also some costs. "Interference" is an important one. Interference is caused by the same mechanism that underlies generalisation: the updating of weights executed to learn an input-output association influences other input-output associations. If the first association is dissimilar/not correlated with the second ones, this may result in a negative effect in terms of error (Hinton et al., 1986). Interference is particularly impairing when different sets of input-output associations are learned at different times, i.e. the learning of the input-output associations of the sets are not interleaved. In fact, each time a set is used to train a neural network for several times in a row, the information previously accumulated for the other sets is disrupted (cf. Sharkey and Sharkey, 1995; Blanzieri and Katenkamp, 1996).

A consequence of this is that when neural networks are used to control autonomous robots, interference is particularly impairing. In fact different sets of input-output associations generated by the interaction of the robots with the environment tend to be clustered in different periods of time. This point is even more important if planning is implemented in a way similar to what has been done in the following chapters. In this case, planning implies that the controller focuses on the same goal for a long period of time before passing to another goal (cf. s. 10.1). This focussing implies that a particular set of input-output associations is learned several times before passing to another set.

4.4.2 Prototype Extraction

There is a property of neural networks closely related to the generalisation property and noise tolerance: the capacity to generate prototype representations of the input patterns. Suppose that a three-layer feed-forward neural network with sigmoidal units is trained with noisy versions of some input-output pairs. If the original input pattern of some of the pairs is presented to the network, the network will tend to return the original output pattern of the pairs (without noise), even if it has never experienced them. The network has extracted the "prototype" of the input-output associations and tends to suppress the noise. This property mainly relies on the non-linearity of the neural network (cf. McClelland et al., 1986).

As we shall see in s. 8.4.3, this property is particularly important for the kind of neural planners implemented here. In fact these planners generate simulated "mental walks", i.e. possible future sequences of states of the environment that the controller expects to observe by following a particular course of action. As we have seen in s. 4.2 this generation of sequences of states is implemented by using a neural network (predictor) whose output is repeatedly fed back into its own input layer (together with the "programmed" action). Given the numeric representations used by neural networks, there is the danger that noise accumulates during this looping, so that the output becomes a meaningless noisy pattern not corresponding to any state of the environment. We shall see that, thanks to the prototype extraction property of neural networks, this danger is kept under control, and the coherence between the patterns generated and real states tends to be preserved. In fact the patterns generated tend to be prototypes (without noise) of the real states, i.e. the patterns that

correspond to real states tend to be "points of attractions" for the sequences of predicted states generated.

4.4.3 Learning

One of the most appealing properties of neural networks is their capacity to learn. These means that initially neural networks associate random output patterns to input patterns, if their weights are drawn randomly as it is usually done, but with suitable training (cf. s. 13.3.1 and 13.3.2) they can learn to associate appropriate patterns with them.

Notice that learning is not necessary for planning. A neural planning system, with fixed architecture and weights, could plan solely on the basis of the activation of its units. However, in this case the designer would encounter the problem of how to find the weights of the neural system. Learning, like other evolutionary/adaptive strategies, is a way to solve this problem. In fact it is based on algorithms that automatically find the weights suitable for the problems.

Learning has positive and negative implications for planning. One positive implication, connected with the point just mentioned, concerns the degree of autonomy (i.e. absence of human intervention) that can be achieved with neural planners. By definition predictive planning relies on information about the consequences that actions have on the environment, incorporated in the "model of the environment". In classic artificial intelligence the designer usually hardwires such information in the system (cf. s. 2.1.1 and 2.3.3). The capacity of learning makes it possible to design neural planners that build up their own model of the environment autonomously through experience (cf. s. 4.5 for some examples of this).

One negative implication concerns the time required by learning. The majority of the algorithms employed to train neural networks require that the neural network experience each input-output association several times (Haykin, 1999, for a review). Each time the network's performance improves of a small amount until it reaches a desired level. One remarkable feature of planning is its flexibility and its "one shot" nature. For example, if we want to move an object away from us, we can think one time about the consequences of pushing or pulling the object, *store in "one-shot"* the output of this processing in some form of memory, and select the proper action accordingly. This is not easy to implement with neural networks. In fact few algorithms developed so far to train neural networks implement *one shot learning* (these are usually based on some kind of Hebb rule, cf. Hebb, 1949, and Hopfield, 1982). The problem is that the learning capacities of one-shot learning algorithms are usually limited and imply the loss of the useful properties of generalisation, information compression, prototype extraction, and noise reduction (Rolls and Treves, 1998, p. 33). Given that these properties are very important, in this research it has been decided to use incremental learning algorithms as error backpropagation (cf. s. 13.3.1). As we shall see the negative consequence of this will be that the planners have to "think" about the same situations over and over in order for the knowledge about what to do in different circumstances to be stored suitably.

4.5 Planning with Neural Networks

In this section, some neural planners that are representative of all existing neural network planning controllers are reviewed. The objective is to highlight the principles that have been used so far to this purpose. Neural planning controllers can be grouped under three categories. The first category includes planning controllers based on dynamic programming and Dyna architectures, reviewed in s. 13.2.9 and 3.3, so they are not considered in this section. The second category includes planning controllers based on the principle of "activation diffusion".

The third category includes planning controllers that use a gradient descent algorithm to compute the proper sequence of actions or subgoals that make up the plan.

4.5.1 Activation Diffusion Planning

Activation diffusion planning (Lei, 1990; Hampson, 1998) is one of the most popular techniques used to implement neural planning controllers. Methods based on it have often been used to build and use neural "cognitive models" for spatial navigation (Mataric, 1991; Levenick, 1991; Kortenkamp and Chown, 1992; Ravel et al., 1998; Trullier and Meyer, 1998) but also to implement neural motion planning for plant and robot control (Fomin et al., 1996; Zeller et al., 1997; Fleuret and Brunet, 2000). It is a planning method that can be "easily" implemented with neural networks. The method has interesting relations with dynamic programming and heuristic search, as we shall see in s. 5.2.

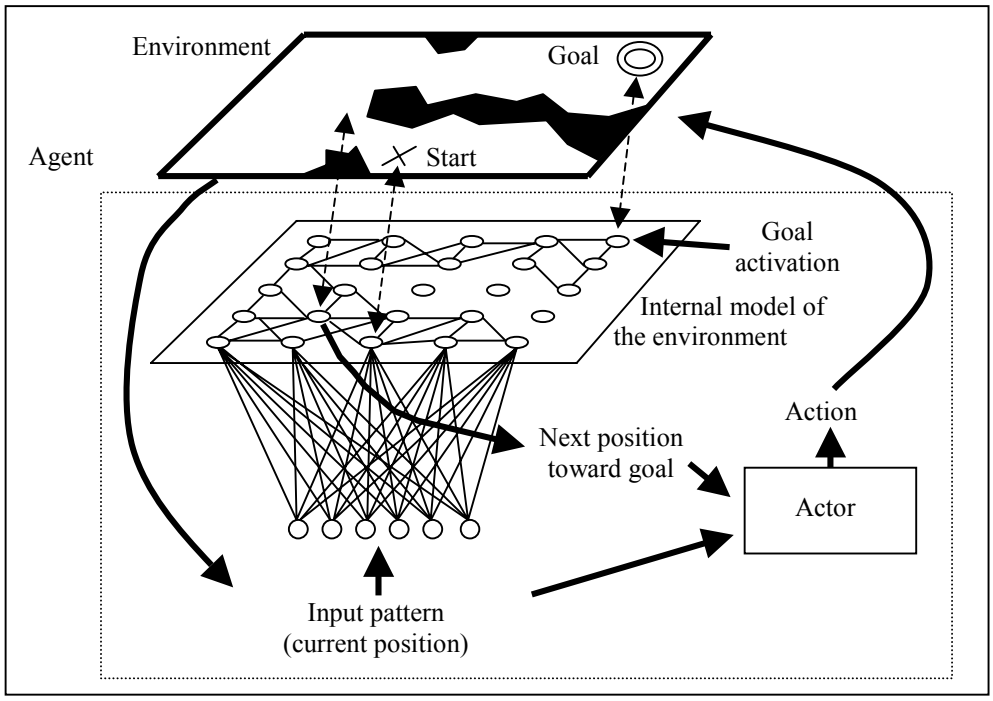


Figure 4.2: The architecture and environment of an agent capable of executing activation diffusion planning. The bold arrows represent the main flows of information. The dotted arrows show the correspondences between points in the environment and internal units representing them.

The best way to show how activation diffusion planning works is to consider a navigation task such as the one presented in Figure 4.2. This graph shows both the task and the controller's architecture implemented in neural terms. In particular the graph shows: an arena with obstacles where the agent has to solve a path-finding problem; the agent's internal model of the environment, consisting of a two-dimensional array of neural units; the input units of the agent, each of which is activated when a particular "feature" of the environment is detected; a component of the architecture that here has been called "actor". The core of the controller is the internal model of the environment. When the agents is in particular position of the environment, *one* unit of the model corresponding to it is activated. This is made possible by the connection between the input units and the model units. These connections store a "prototype" of the features of the percept generated by the environment in that

position. For example the model of the environment and these connections could consist of a Kohonen neural network (Kohonen, 1982). This network builds the prototypes by learning and activates one unit of the model at a time through a "winner-takes-all" competition.

In an initial phase the agent wanders randomly in the environment and builds up some *links* between the model's units by experiencing which transitions between these units (positions in the environment) are possible and which are impossible. Successively, when the controller is planning to go to a particular position in the environment, it activates the model's unit that corresponds to it with activation r. At this point the activation *diffuses* from the goal unit to the neighbouring units, decreasing in intensity when passing through the links. At each time step the activation can diffuse from one unit to its neighbouring units one link distant, and decrease when passing through each link. The activation level of each model's unit becomes γ^t r, where t is the minimum number of links from that unit to the goal unit. When the activation diffuses from the goal to the other units, if a unit receives activation from more than one neighbouring unit, it assumes the maximum activation possible.

When the activation reaches the unit corresponding to the current position, the "plan" is executed: at each time step the agent moves to the neighbouring state corresponding to the model unit with the greatest activation. This can be implemented in several ways, for example by using a look-ahead exploration of neighbouring units to find the one with the highest activation and an "actor" as the one shown in Figure 4.2 (cf. the literature cited at the beginning of this section). The "actor" is a servomechanism that takes the features of the current position and the features of neighbouring position with the highest activation as input, and returns the proper action. Notice that if this component is a neural network, it can be trained in the initial phase together with the model of the environment.

Critical Observations. Activation diffusion planning has interesting relations with dynamic programming and with heuristic search. We have seen that the units' activation that it generates is γ^t r. Notice that these values are also the *optimal* evaluations that are generated by discounted dynamic programming after it converges when it is applied to a deterministic environment (cf. s. 13.2.9). Notice also that when applied to deterministic environments, dynamic programming, as activation diffusion, does not need to iterate, but it generates the correct states' evaluations "one-shot" starting from the goal and moving away towards the states more distant from the goal. In these respects there is a precise equivalence between the two techniques.

Activation diffusion planning is very interesting for its simplicity and the speed with which it generates plans. However it has a major drawback: each state needs to be represented with a unit. If fact if each state were represented by many units through a distributed representation (cf. s. 4.1) the activation diffusing to the units of one state would influence the activation of other states that share the same units. This implies that the space complexity is proportional to the number of states of the environment to be stored. This problem is less impairing if, as in the example, neural networks are used to compress several states into few "prototypes". However this is still not satisfying. In fact the number of units needed still grows proportionally to the number of states since one unit can still represent only *one* prototype. The planners designed and implemented in the next chapters use *distributed* representations for the states of the environment. When distributed representations are used, patterns can be stored much more efficiently (cf. Rolls and Treves, 1998, p. 41). The drawback of this approach is that a lot of iterations are needed to updated the evaluations for each state, as the evaluations of different states depend on the same features, and hence tend to have reciprocal influence (cf. s. 13.3.3).

Notwithstanding its drawbacks, activation diffusion planning is a very interesting framework. For example the one-shot nature of the activation diffusion process that formulates a plan, and the format of the model of the environment in terms of explicit links between "contiguous" states (model's units) that can be visited by selecting suitable actions, are very appealing properties. Interesting insights can be achieved by comparing activation diffusion planning and dynamic programming, and by attempting to design planners that incorporate features and strengths of both. Chapter 9 implements a planner that diffuses the evaluations from the goal much like activation diffusion planning diffuses activation from the goal.

4.5.2 Neural Planners Based on Gradient Descent Methods

This subsection briefly reviews some neural planners that use gradient descent methods for finding the action plan or subgoals. The first neural planner (Tani, 1996) has been used to control a robot that solves a navigation problem by planning. The robot moves at a constant speed, and is endowed with an hardwired navigation system that allows it to move toward the biggest open space between the obstacles. The robot has to plan a sequence of binary ({0, 1} i.e. "left" or "right") actions to reach the goal, where an action is the decision of which open space to choose when a "branching point" is met. A branching point is a point where two open spaces are visible. The system is based on a model of the environment implemented by a feed-forward neural network trained with a back-propagation algorithm in a preliminary learning phase. This network takes the current state and the programmed action as input, and returns the predicted future state as output. The plan is built by a gradient descent method that minimises a cost function with respect to the actions. To this purpose the binary actions at each step of the plan are considered as the extremes of a [0, 1] segment. This cost function depends on the mismatch between the predicted state and the goal state, the length of the path, and the distance of the actions from their required {0, 1} values. A function based on chaos theory is employed to overcome the problem of escaping the several local minima of the rugged cost function.

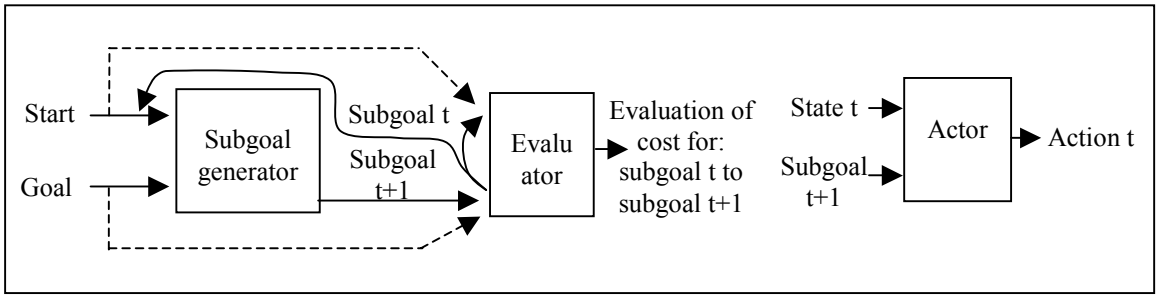


Figure 4.3: The architecture of Schmidhuber and Wahnsiedler's planner.

The architecture of a second neural planner (Schmidhuber and Wahnsiedler, 1992) is shown in Figure 4.3. This system has been tested with a simulated path-finding problem where the cost of the path is differentiable with respect to the states, and each state is encoded with (x, y) co-ordinates. In particular it is a navigation task where traversing some swamps in the way to the goal has costs depending on the swamps' depth and the positions (x, y) occupied by the agent.

The system is made up of three components: an actor that is capable of reaching a close subgoal from a given state (it could be trained or hardwired feed-forward network); an evaluator that is capable of estimating the cost of going from one state to another state in a straight path (it could be a trained or a hardwired feed-forward network); and a subgoal generator that is capable of generating the next subgoal (e.g. the next x, y to reach, on the basis of a start state and the goal state). The subgoal generator is the focus of the work. It is trained with a gradient descent algorithm so that it generates a sequence of subgoals such that the *sum* of the costs of the whole path is minimised.

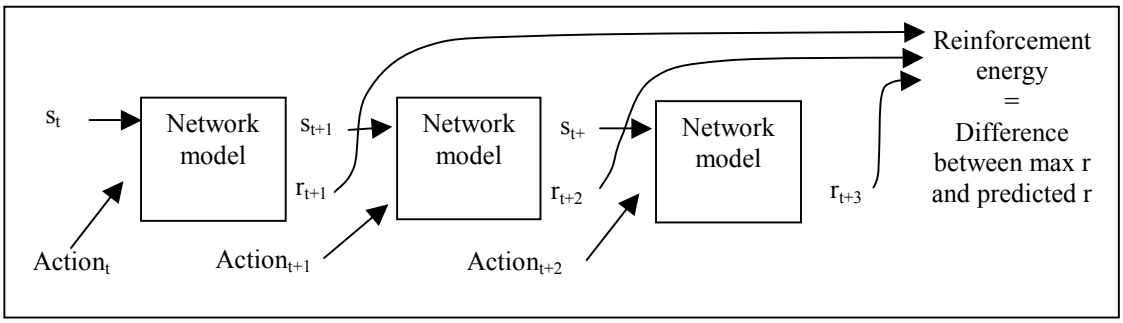


Figure 4.4: The architecture of Thrun's planner.

Thrun et al. (1991) have proposed a neural planner whose architecture is shown in Figure 4.4. The planner is mainly composed of a model of the environment capable of returning the immediate reward and next state on the basis of the current state and the programmed action. This model is trained with the error backpropagation algorithm. The planner has been tested on a robot arm (simulated and real) that has to reach a rolling ball. The position of the ball and hand, and the actions are all coded with continuous variables so that the rewards are differentiable with respect to them. The actions of the plan are computed with a gradient descent algorithm that maximises the reward.

Critical Observations. Though interesting, these systems are based on the assumption that the models of the world or the subgoal generator are differentiable with regard to actions or subgoals. This limits their applicability to tasks where the states and actions are represented by *few continuous* variables.

Moreover, the planners reviewed try to find the actions or the subgoals through an iterative process that minimises a cost function. This cost function is based on the mismatch between the predicted state and the goal state, or the cost of the path, or the reward received. In these regards, these systems are similar to the solution strategy adopted by dyna architectures, which tries to find the actions that maximise the reward. The major differences are that in the systems reviewed here actions are modified by a gradient descent method, that gives the *direction* of change, while in dyna architectures a random search is used.

5 Unifying Concepts

This section unifies and generalises some concepts explored in the previous chapters. In particular s. 5.1 investigates the relationship between learning, planning, prediction and taskability, s. 5.2 investigates the relationship between some of the most important methods of heuristic search, dynamic programming and activation diffusion planning, and finally s. 5.3 compares the concept of plan and policy.

5.1 Learning, Planning, Prediction and Taskability

This section generalises some concepts encountered in the previous chapters. In particular it offers a formal presentation of "learning of behaviour" and "planning" for *asynchronous multi-goal tasks* (cf. s. 3.1). This serves two purposes:

- To give a precise definition of the notions of " learning of behaviour " (s. 5.1.1) and "taskable planning" (s. 5.1.2). The later is a form of planning more restricted than the one defined in s. 1.1.
- To show (s. 5.1.4) that the original Dyna-PI architecture is a planner not taskable "in a strong sense".

The first point is important because the controllers designed and implemented in the following chapters are capable both of planning and learning, so a clear definition of these concepts will help to investigate their properties.

The second point is important because, even if the concept of taskability is clear and "natural" within the problem solving and planning frameworks, it is much more subtle and easily confusable within Markov decision processes and systems that are both capable of learning and planning such as Dyna architectures. Within this literature (e.g. cf. Sutton and Barto, 1998, pp. 56-57; Sutton, 1990) the use of the reward is said to be *always* more general than the use of an "explicit" goal, i.e. a goal defined as a *state* to achieve. This section and chapter 8 show that this is not the case.

The importance given to the problem of taskability by this research is justified by the choice of reinforcement learning as a framework to implement planning with neural networks.

Before continuing, it is important to notice that the definition of goals through rewards yields advantages in tasks with multiple *synchronous* goals. These are tasks where the controller has to pursue several goals *at the same time*. In these cases the controller needs to "weight" goals on the basis of their importance to decide how to distribute its efforts and time between them. These problems are not dealt with within this research. For an extended explanation of these problems and a review of some of the algorithms proposed to solve these tasks see Tyrrell (1993), Humphrys (1996) and Hampson (1998).

In the following sections learning and planning algorithms are left unspecified to include most of the algorithms considered in this thesis. Moreover, for simplicity:

- It is assumed that the environment is deterministic.
- It is assumed that the "goal" pursued is defined in terms of one particular state.
- It is assumed that the Markov property holds.

- In the case of planning, the analysis considers only the case of policies (and not the case of plans, cf. s. 5.3).
- In the case of learning, the analysis considers only the simple case of reward 1 associated with goal states, and reward 0 associated with all other states.

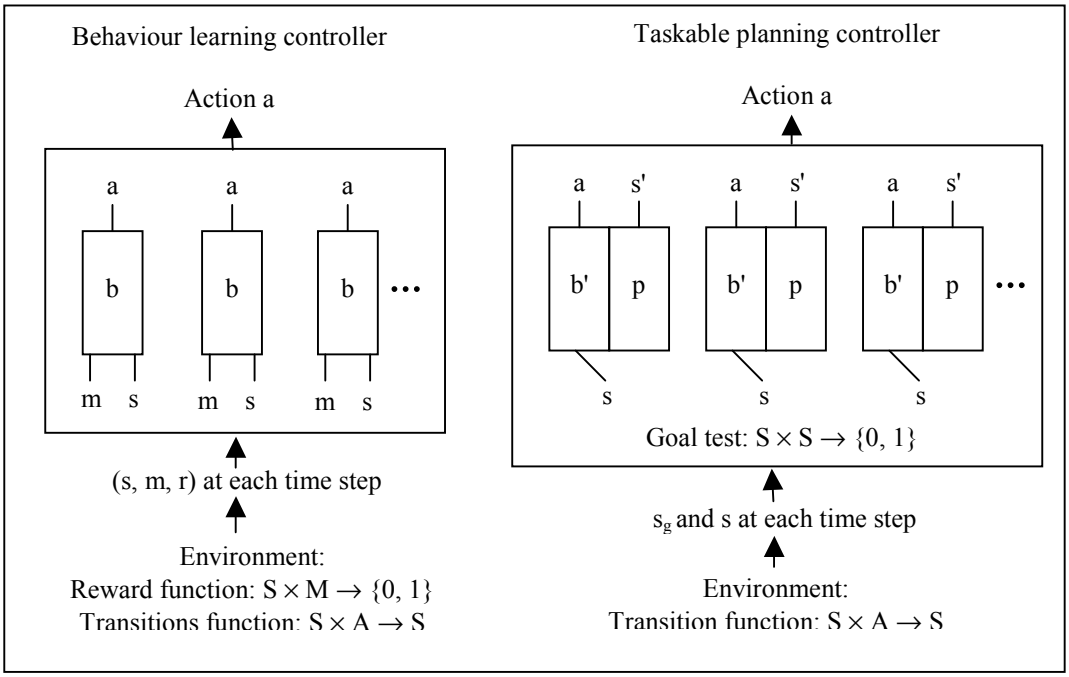


Figure 5.1: Two controllers capable of performing "learning of behaviour" and "taskable planning". m: motivation. s: state. a: action. b, b': behavioural rules. r: reward. p: prediction.

5.1.1 Learning of Behaviour

"Learning of behaviour" is the process that allows the controller to find "good" associations of the kind "state, motivation → action" according to a "reward" signal r. The task is divided into finite periods of time (trials). At each time step the input to the controller is as follows:

- The current state of the environment $s \in S$.
- A signal $m \in M$, called "motivation", on the basis of which the controller has to learn to pursue a particular goal state $s_m \in S$ within a trial. A trial ends either when s_m is achieved or when some time elapses.
- The reward $r \in \{0, 1\}$.

The controller is made up of the following components:

- An adjustable set B, called "behaviour", made up by tuples, called "behavioural rules", of the kind $b = (s, m, a) \in (S \times M \times A)$, with at most one tuple for each (s, m).
- Some other data structures and algorithms used to modify the behaviour (see below).

The environment is made up of the following components:

- Reward function: $S \times M \rightarrow \{0, 1\}$. Given a particular m, the reward is 1 if $s = s_m$, and 0 otherwise.
- State transition function: $S \times A \rightarrow S$.
- The end of the trials, the motivations and goals are exogenous. m and the corresponding goal state s_m are the same during one trial, and change at the end of trials.

The interaction with the environment is as follows:

- At each time step the environment sends (m, s, r) to the controller.
- At each time step the controller executes an action a in the environment.

The controller's task is to find a suitable behaviour in order to maximise a given function of the rewards obtained over time (e.g. the sum of them, the discounted sum, etc.). To accomplish this task the controller follows this procedure (*learning of behaviour*):

- At the beginning the controller creates the behaviour at random.
- At each time step the controller selects a behavioural rule (m, s, a) from the current behaviour on the basis of the input (m, s), and executes the corresponding action a in the environment.
- At each time step the controller evaluates the "quality" of the behaviour in terms of the rewards r obtained and modifies its behavioural rules accordingly.

5.1.2 Taskable Planning

Planning is the process that allows a controller to find "good" associations of the kind "state $\rightarrow$ action" according to a "goal test" (see below) and eventually other tests (e.g. test about costs). The task is divided in finite periods of time (trials). At each time step the input to the controller is as follows:

- The current state of the environment $s \in S$.
- The goal state $s_g \in S$ that represents a state that the controller has to achieve before the trial ends. A trial ends either when the controller achieves the goal, or when some time elapses.

The controller is made up of the following components:

- A fixed set of "behavioural rules" $b' = (s, a) \in B' = S \times A$.
- A fixed set of "predictions" $p \in P$, each associated to a particular b'. A prediction is a fixed couple of the kind (b', s'), that says which is the state s' that is reached if the action a of the rule b' is executed in the state s of the same rule. P is the controller's "model of the environment".
- A "behaviour" (or policy), i.e. an adjustable subset $B'_c \subset B'$ of behavioural rules (s, a), with at most one rule for each possible $s \in S$.
- A goal test, i.e. a function of the kind $S \times S \rightarrow \{0, 1\}$. The goal test is applied to each state s visited to check if it corresponds to the goal s_g (the outcome of the test is positive or negative respectively).
- (Eventually) other tests, based on the "costs", "length", etc., to manipulate the behaviour.
- Some other data structures and algorithms used to modify the behaviour.

The environment is made up of the following components:

- Transition function: $S \times A \rightarrow S$, where the current state and action generate the following state.
- The goal is the same during a trial, and changes at the end of the trials. The goal is set exogenously.

The interaction with the environment is as follows:

- At each time step of a trial the environment sends a goal state s_g and a state s to the controller.
- At each time step the controller executes an action a in the environment.

The controller's task is to reach the goal states s_g in each trial. To accomplish this task the controller follows this procedure (*taskable planning*):

- Lookahead. At each time step the controller searches for the behavioural rules making up the behaviour B'_c by using the model of the environment P ,the goal test, and the other tests.
- Action execution. At each time step, on the basis of the state of the environment s, the controller selects a behavioural rule from the behaviour B'_c and executes the corresponding action a in the environment.

Observations. According to the previous formalisation, "learning of behaviour" is a procedure that changes the behaviour on the basis of the reward signal coming from the environment, i.e. on the basis of *experience in the world*. Usually, as it is done in reinforcement learning, this procedure works by trial-and-error, i.e. it "guesses" a behaviour, tests it by the world and then tries to improve it on the basis of the consequences in terms of reward. An important consequence of this is that learning of behaviour needs to experience the goal many times in order to become capable of achieving it efficiently. On the contrary (in principle) planning does not need experience in the world before acting appropriately. In fact the core of planning is a *search procedure* carried out on the basis of the model of the environment. This procedure tries to find a suitable *linked combination of behavioural rules* that satisfies the test goal and eventually other tests. An important consequence of this is that planning can potentially prepare a plan and then act and reach the goal efficiently in "one shot". The search procedure of planning can work in many ways. An example are the searching and planning methods reviewed in chapter 2, based on systematic explorations of the possible combinations of the behavioural rules. Another example could be a simulated annealing procedure (Russell and Norvig, 1998, pp. 113-114) that searches in the space of the behaviours and uses the goal test and the other tests to find a "good" one. Other examples are the neural network planners inspired by the Dyna architectures presented in the following chapters.

5.1.3 Taskability: Reactive and Planning Controllers

We are now in the position of clarifying in which sense reactive controllers and planning controllers can be taskable (cf. also Sutton, 1991; Russell and Norvig, 1995, p. 790, on the concept of taskability). In the case of the "behaviour learning controller" the signal of the motivation plus the state of the environment are the input to the system. If the controller can receive |M| different motivations, than a user can train the controller to achieve |M| different goal states by associating a proper reward function to them. Notice that m can be a part of s, for example it can be a particular vocal command pronounced by the user. This implies that there are no practical limits for the number of possible motivations. Notice also that the reward r itself can be a part of s, for example it can be a particular "rewarding" word pronounced by the user in some circumstances. In the previous presentation m and r have been distinguished from s for the particular role they assume in affecting learning and behaviour.

After training is accomplished for some motivations and goals, the user can direct the controller to pursue the various goal states by giving the corresponding motivation signal to it. Notice that in this way the controller can be directed to achieve *only* the goals for which it has been trained. If the user wants that the controller pursues *a new different goal*, it has to:

- Furnish a new motivational signal.
- Train the controller with a new reward function. In practical terms this means that the user, or some other mechanism in the environment, has to be there while the controller learns, and has to furnish a reward when the controller achieves the goal.

In these cases, where "a particular motivational signal *and* a particular reward function are used for each goal", we say that the controller is "taskable in a weak sense".

The case of the "taskable planning controller" is different. If a user wants to assign *any new goal* to the controller, it is only necessary to assign the desired goal s_g to the controller.

This means that there is no need of a new *reward function* for each new goal. In fact the test goal, *internal* to the controller, allows the controller itself to build and select the proper plan or policy.

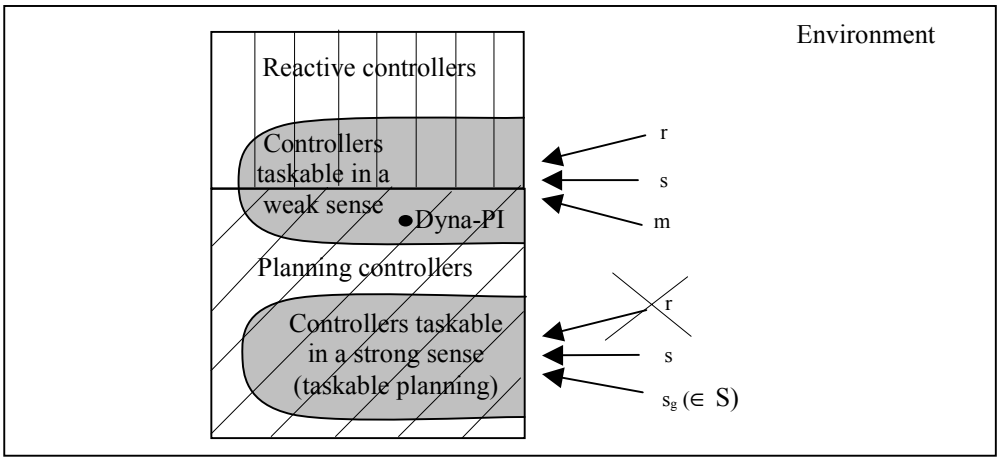


Figure 5.2: The different nature of the reactive and planning controllers with regard to taskability, graphically represented with sets. Notice that taskability in a strong sense, in comparison to taskability in a weak sense, does not require a reward signal from the environment. However, it does require a signal s_g that usually contains more information than the motivation signal required by weak taskability. Dyna-PI (the dot) is an instance of planning controllers but it is not taskable in a strong sense.

Notice that in the case of the "behaviour learning controller", the motivational signal can have any nature: it is enough that it can assume a number of different configurations equal or greater than the number of different goals relevant for the user. Instead, in the case of the "taskable planning controller" the goals signal that directs the behaviour of the controller has to be a state $s \in S$. This usually means that it carries a greater amount of information, again measured in terms of number of different configurations that it can assume.

In cases where "any goal $s_g \in S$ can be assigned to the controller with no need for a reward function", we say that the controller is taskable in strong sense, or simply that it is "taskable". Figure 5.2 presents a graphical summary of the possible cases.

The controller proposed and implemented in chapter 8 is a planning controller *and* a reactive controller at the same time. In fact when it plans to achieve a goal, it also stores knowledge about how to reach it reactively in permanent memory structures at the same time. After achieving the same goal several times by planning, this knowledge becomes ready available to pursue the same goal in a reactive way. As we shall see, in this circumstance the signal of the goal assumes the role of "motivation", in the sense that is used by the system to retrieve the reactive skills suitable to pursue the goal.

Detecting Taskability. It was important to draw a clear distinction between learning of behaviour and taskable planning because when one implements planning with neural networks it is easy to confuse the two, especially with learning neural systems that pursue multiple goals. Given the previous analysis, how is it possible to determine if a controller is a

taskable planner? There are two tests that the controller should pass to be classified as "taskable (in a strong sense)":

- The controller receives the goal state "from outside". Eventually the state transition function part of the model of the environment is given to the controller. The controller does not need any other information to achieve the goal. In particular it needs no information about the reward function or the reward.
- When the goal is assigned to the controller for the first time, the controller is capable of achieving the goal with an enhanced efficiency if compared to its reactive components (in the case there are some) or better than the random solution (in the case there are no reactive components). This criterion derives from the "one-shot nature" of planning.

Notice that here only planning controllers that work by "compiling" the outcome of planning into the reactive components are considered (cf. Mitchell, 1990; Dyna-PI architectures are compiling planners, cf. s. 3.3). In these systems the performance of the planning components plus the reactive components is at least comparable with the performance of the reactive components alone. These tests will be used to check if the controllers proposed and implemented in the later chapters are taskable.

5.1.4 Taskability and Dyna-PI

Now it is possible to clarify why Dyna-PI is not a taskable planner. As mentioned reinforcement learning algorithms need a different reward function, and a motivational signal, *for each goal* pursued. Dyna-PI architectures are based on a model of the environment that represent *both* the transition function and the reward function. The consequence of this is that Dyna-PI architectures are not taskable in a strong sense. In fact if a *new* goal is assigned to the controller, the controller does not have a model of the reward function for it. The only ways the controller can pursue the new goal are: (a) the controller is trained with the new reward function so that it can *learn* the part of the *model* of the environment related to it; (b) the part of the model of the environment related to the new reward function and goal is directly furnished to the controller. In both cases the controller does not satisfy the definition of taskability in a strong sense given previously. The consequence of this is that the only thing that an agent can do if it is assigned (or it selects) a goal is pursuing it on the basis of a random walk (see Lin, 1992).

5.2 A Unified View of Heuristic Search, Dynamic Programming, and Activation Diffusion

This section attempts to build a unified view of some important searching and planning methods analysed in previous chapters. These methods are: LRTA* (cf. s. 13.1.2) and trial-based real-time asynchronous dynamic programming with deterministic environment (cf. s. 13.2.11); Uniform cost search from the goal (cf. s. 13.1.1) and "cost" dynamic programming with deterministic environment (cf. s. 13.2.9 and 13.2.11); Dyna architectures (cf. s. 3.3) and "discounted" dynamic programming with stochastic environment (cf. s. 13.2.9); Activation diffusion planning (cf. s. 4.5.1) and "discounted" dynamic programming with deterministic environment (cf. again s. 4.5.1). The adjective "discounted" is used to refer to problems defined through goal states with positive rewards and non-goal states with 0 rewards (cf. s. 13.2.1). The adjective "cost" is used to refer to problems defined on the basis of goal states conceived as absorbing state and costs caused by the execution of actions (cf. s. 13.1.2 and s. 13.2.11).

All the methods just listed, abstracting from the details, are based on two steps:

- Building a gradient field of "evaluations" (or "heuristic") over the states, increasing (discounted methods) or decreasing (cost methods) toward the goal (in some cases an initial approximate heuristic or set of evaluations are available to the system).
- Generating a plan (on-line or off-line) by "looking ahead" one step through a model of the environment, and by selecting the "closest" state to the goal according to the gradient field.

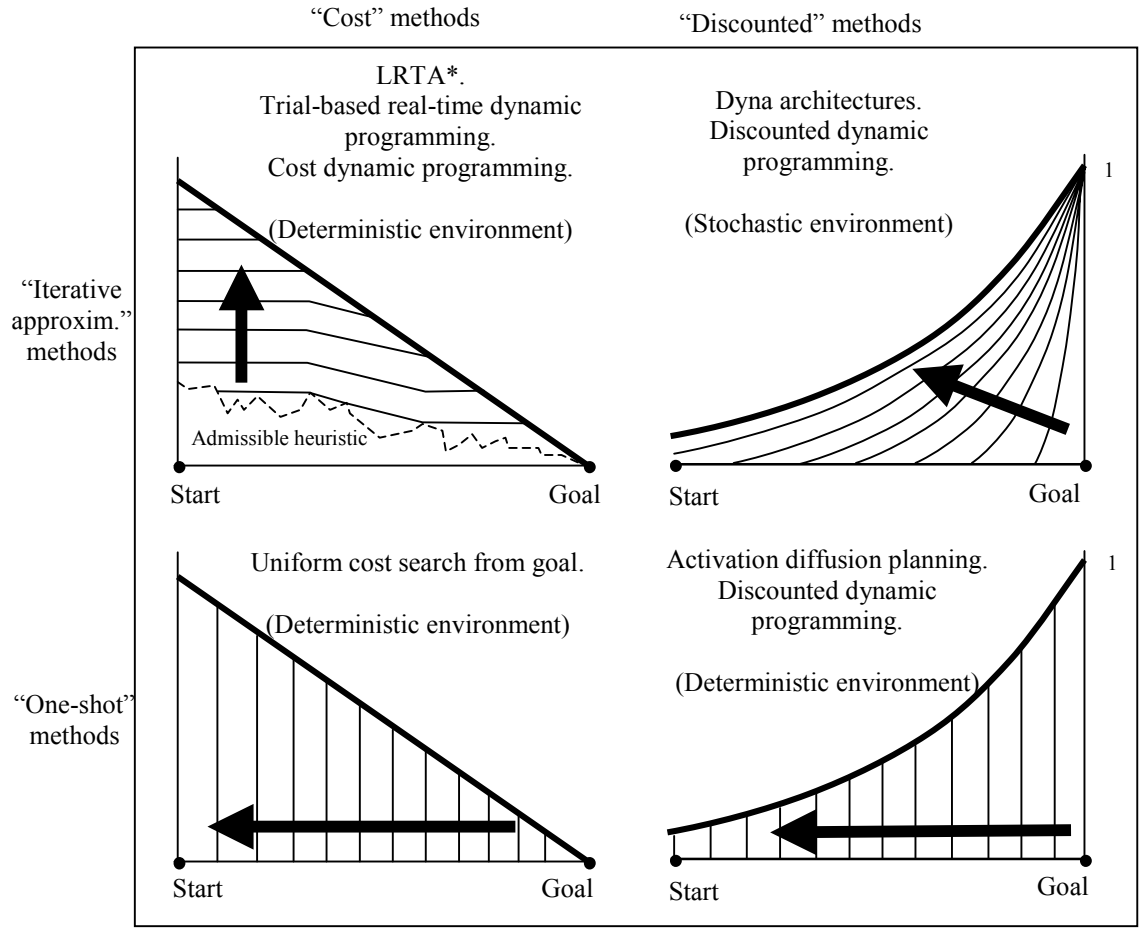


Figure 5.3: A unified view of the evaluations updating performed by some of the most important methods reviewed in this chapter. X-axis: the states between the start and the goal (for simplicity only the start and goal states are marked). Y-axis: the level of the evaluations. Dotted line: the admissible heuristic of LRTA*. Bold curves: the optimal evaluations (in the case of the cost methods, the costs are assumed to be proportional to the distance of the states from the goal). Thin lines: the "wave-front" of the approximate evaluations. For simplicity these evaluations are represented by regular lines that ignore local noisy details (especially in the case of dyna architectures that deal with a stochastic environment). Bold arrows: the expansion of the wave front of the evaluations under the updating of the algorithm. Top part: methods that build the correct evaluation function (heuristic) by iterative approximations. Bottom part: methods that build the correct evaluation function one-shot. Left part: methods based on goals as absorbing states and costs. Right part: methods based on positive reward associated to the goal and "discounted" evaluations. Comments: LRTA* tends to "fill in holes" of low values of the heuristic and to created values flat or descending toward the goal.

It is interesting to notice that it is possible to classify the listed methods and highlight their relationship on the basis of: (a) the nature of the evaluations employed; (b) the way they

update the evaluations. Figure 5.3 proposes a unified graphic summary of the four possible combinations of these two dimensions. For graphical reasons, this figure shows a one-dimensional space of states and represents graphically how the listed methods behave along the two dimensions.
On the basis of the evaluations' nature, we can distinguish between:

- Left part of Figure 5.3: cost methods where the gradient field decreases linearly for states progressively closer to the goal (LRTA*, cost trial-based real-time dynamic programming with deterministic environment, cost dynamic programming with deterministic environment, and uniform cost search from goal).
- Right part of Figure 5.3: discounted methods where the gradient field decreases exponentially for states progressively more distant from the goal (Dyna architectures, discounted dynamic programming with stochastic and deterministic environment, and activation diffusion planning).

On the basis of the way the methods update the evaluations we can distinguish between:

- Top part of Figure 5.3: methods that build the correct evaluation function, or heuristic, by iterative approximations (LRTA*, trial-based real-time dynamic programming, cost dynamic programming with deterministic environment, Dyna architectures, and discounted dynamic programming with stochastic environments).
- Bottom part of Figure 5.3: methods that build the correct evaluation function one-shot (uniform cost search from goal, activation diffusion planning, and discounted dynamic programming with stochastic environments).

Notice that only Dyna architectures and dynamic programming are capable of dealing with stochastic environments (top right part of Figure 5.3) while all other methods assume a deterministic environment.

Summarising, in cases of absence of initial heuristic and repeated trials, the majority of methods considered in the previous chapters are based on the strategy of building a gradient field of evaluations and, in different measure, they are related to some form of dynamic programming. This means that the principles of *evaluations* and *lookahead search* on the basis of these evaluations are very general and powerful. This circumstance is very important for neural network planning. In fact it is relatively easy to conceive of ways to implement neural networks that learn to produce an evaluation gradient field over the states. The following chapters will show some ways to do this.

5.3 Policies and Plans

It is important to draw some conclusions on the differences between plans and policies.

Plans. In its *pure form* a plan is a sequence of actions of the kind "a_1, a_2, a_3,…" to be executed one after the other (cf. left of Figure 5.4). What is important is that there is an *ordering between the actions*: the execution of one action is conditional to the execution of the previous action in the plan, and is triggered by it. Notice that the plan execution is "blind" i.e. there is no monitoring of the actions' outcome.
Policies. In its *pure form* a policy is a set of associations of state-action pairs of the kind "(s_1, a_1), (s_2, a_2), (s_3, a_3),…" for *all* the states of the state space (cf. right of Figure 5.4). A policy is executed as follows. At each time step the current state is detected, the list is scanned with the state being used as retrieval-key, and the action of the pair with the state corresponding to the current state is executed. Notice that there is *no ordering* between the actions i.e. the current state is sufficient to decide which action to trigger (this requires that the Markov property holds for the policy to be successful, cf. s. 13.2.2).

Plans vs. Policies. The advantages of plans vs. policies are as follows:

- Plans take less time to be prepared and less space to be stored in comparison to (complete) policies that require a huge amount of both. In fact plans focus on relatively few states and actions.
- The information contained in the ordering of the action furnishes some kind of memory that can be useful with partially observable Markov decision problems (Wiering and Schmidhuber, 1998; cf. s. 13.2.2).

The advantages of policies vs. plans are:

- They are capable of coping with stochastic outcomes of action execution and non-perfect models of the environment because the execution of each action is based on the current state, and because the policy specifies what to do for each possible state visited. Plans have more problems in dealing with these situations because they are committed to a particular ordering of actions, and do not "know" what to do in case of unexpected outcomes of actions' execution.

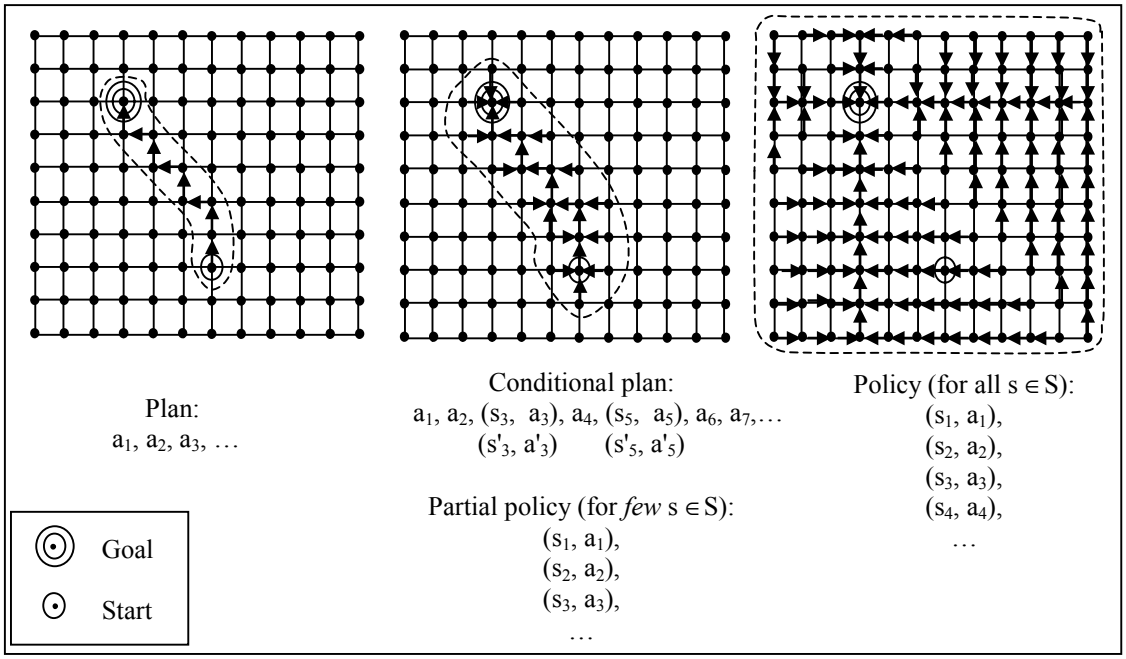


Figure 5.4: Example of plan (left), conditional plan and partial policy (centre), and policy (right). Cf. text for details. The dotted curves are drawn around the area of the state space on which the different methods are competent.

Conditional Planning and Partial Policies. We have seen (cf. s. 2.3 and 2.4) that many planners that have been implemented are different from the pure plan just illustrated. In fact they build plans that are partially ordered or use monitoring of the outcome of action execution. In the case of "conditional planning" they use "perceptive actions" that returns some aspects of the current state that condition the plan's execution. As mentioned (cf. centre of Figure 5.4 and s. 2.4.1) conditional plans are close to the concept of policy. On the other side we have seen that the idea of an satisfactory policy defined for each state is not viable in practice, so that the policies used in practice tend to focus on limited regions of the state space (cf. s. 13.2.9, 13.2.10, 13.2.11). A partial policy is a policy that is satisfactory and well

defined only for few states among all possible ones. For all other states the policy uses some default criteria to select the actions (centre of Figure 5.4).

On the basis of these considerations it can be concluded that the best strategies are probably to be found between the two extreme strategies of pure plans and pure policies. In particular this is what a good planner should do (cf. also s. 2.4.4):

- Use partial policies (or conditional plans, cf. centre of Figure 5.4). This means that when planning the controller should prepare to act in some states that deviate from the states that the controller expects to visit with high probability.
- The possible extra states to consider in the partial policy should be only the ones that have a probability of being visited over a certain threshold, given the noise caused by the (eventually) stochastic policy and the stochastic outcome of actions.
- Replan (cf. s. 2.4.2) when some states different from the ones considered in the partial policy, are encountered during the actions' execution.
- Monitor the states visited during actions' execution to guide the selection of the partial policy's actions and the decision of replanning.

PART 2

DESIGNING AND TESTING NEURAL PLANNERS

This second part of the thesis presents the empirical investigations of the research. Each chapter is organised around some problems and has a fixed structure. Each chapter starts with an introduction that presents the set of problems which the chapter deals with, gives an overview of the neural controller used in the experiments, and highlights the novel traits of this controller by comparing it with other controllers existing in the literature. Then the chapter presents the details of the neural controllers and the simulation scenario with which the problems illustrated in the introduction are investigated. Next, the chapter presents the results of the simulations run with these controllers and their possible interpretations. Finally it presents the drawbacks of the controllers and draws the conclusions.

The neural controller presented in each chapter is usually created by adding some extra components to the controller presented in the preceding chapters, so that the complexity of the controllers presented increases across the chapters.

During the research, in order to guarantee the possibility of comparing the results of different simulations, an attempt has been made to keep the conditions under which they have been run consistent, to use the same measures of performance and behaviour, and to present the data in the same format. However, this has not always been possible, since experiments have been run over a long period of time during which new ideas, problems and developments have arisen. As a consequence, the results *across* the chapters are sometimes not fully comparable in quantitative detail, and are sometimes presented with slightly different formats (scale of graphs, details of moving averages, etc.). The results *within* each chapter have been produced with the same conditions and presented in similar ways so that they should be fully comparable.

6 Neural Actor-Critic Reinforcement Learning

6.1 Introduction: Basic Neural Actor-Critic Controller and Simulations' Scenarios

Problems Tackled. This chapter presents a neural implementation of the actor-critic controller. This controller is at the basis of the reactive and planning controllers presented and investigated in the following chapters. It also presents the simulated landmark-navigation scenarios used to test these controllers. Two kind of landmark navigation scenarios will be presented that have different levels of complexity. One scenario has landmarks only outside the arena where the simulated robot moves, and another scenario has landmarks inside the arena. In the later case the landmarks also have the role of obstacles. The components of the basic neural actor-critic controller are tested with these scenarios to collect data to be used to interpret the more sophisticated controllers presented in the following chapters.

The chapter also investigates how the generalisation and noise tolerance properties of the neural controller presented accelerate learning, but also how they exacerbate the "aliasing problem" (cf. s. 13.2.2). Some simulations show the effects of the variation of some parameters of the model and some parameters that control noise, and furnish a justification for the choice of some aspects of the controller's architecture.

A last problem that the chapter tackles is the capacity of discounted reinforcement learning to deal with long periods of time. In particular some simulations show that it has problems to update the evaluations and the actions' probabilities for states far from the goal. This problem is crucial if reinforcement learning methods are used to implement planning, as planning is most useful when it is applied to long periods of time, as happens with abstract planning (cf. s. 11.4.4).

Overview of the Controller. The general functioning of the controller can be described as follows. The actor, a feed-forward neural network, yields a stochastic action-selection policy, and the evaluator, a second feed-forward neural network at the core of the critic, evaluates the states of the environment in terms of expected future rewards achievable with the current actor's policy. The evaluator improves the quality of the evaluations, by experiencing the rewards, through a supervised learning algorithm, while the actor improves the action-selection policy, by increasing the probabilities of actions that bring the controller to ascend the gradient field of evaluations, through a trial-and-error process.

What is New and Related Work. The actor-critic controller implemented in this chapter differs in some aspects from the actor-critic models proposed in Sutton and Barto (1998, p. 151). For a neural implementation of this model see Lin (1992). See Sutton and Barto (1998, pp. 197-200) for the general principles of implementing reinforcement learning controllers through "gradient descend methods" such as the Widrow-Off rule (Widrow and Hoff, 1960). The major differences between the controller presented here and the controllers presented by these authors are the following ones:

- Actions' probability distribution of the "actor". The actor presented here uses a function used to build the actions' probabilities that is simpler than the popular soft-max function shown in Eq. 13.19. Though popular, the soft-max function yields a distribution that has not been demonstrated to be better than other distributions. For example Thrun (1992) shows some cases where it is actually worse than the ε-greedy policy (for which cf. s. 13.2.5). Some simulations with the scenarios presented later have shown that the soft-max function leads the actor to converge too fast, and this worsens the negative effects of the aliasing problem (cf. s. 6.4.3). For these reasons a function that is simpler than the soft-max function and converges more slowly has been used to build the probability distribution of actions (cf. Eq. 6.1).
- The "matcher". As we shall see the controller presented here uses a hand designed neural network called "matcher" to internally produce the reward signal (cf. s. 6.3). After the simulated robot is assigned a particular goal (the state that the simulated robot has to pursue) the matcher returns a reward 1 if the goal has a similarity with the current input pattern above a certain threshold, otherwise it returns a reward 0. This is different from what happens within the standard reinforcement learning controllers, where the reward is thought of as being given from outside the controller when the controller reaches the desired state (cf. Sutton and Barto, 1998, pp. 56-57). This difference is of little importance in the context of simple reinforcement learning controllers, but it is quite important in the context of planning because it allows the controller to be taskable in a strong sense (cf. s. 5.1 and chapter 8).

See also Nehmzow et al. (1989) for an interesting actor-critic architecture that uses "instincts" (hardwired behaviours and task-related criteria for the critic) to enhance the performance and learning of a robot.

The problem presented in s. 6.5, related to the effects of the errors caused by function approximation when *discounted* reinforcement learning is used, has already been investigated by McDonald and Hingston (1994). This work has been pointed out during the PhD's viva, so the problem has actually been "rediscovered" during the PhD research. McDonald and Hingston present a theoretical analysis of the problem, and some empirical results to identify the problem domains that are more sensitive to it. Here, s. 6.5 presents an empirical investigation of the particular effects that the problem causes on the state values computed to solve a landmark navigation task.

Chapter's Outline. S. 6.2 presents the scenarios and the simulated robot used in the simulations throughout the research. S 6.3 presents the architecture of the neural actor-critic controller used in this chapter. S. 6.4.1 presents the functioning of the matcher, while s. 6.4.2 presents the functioning of the evaluator and actor. S. 6.4.3 illustrates the effects of the aliasing problem, while s. 0 presents the effects of the variation of some important parameters of the controller and the scenarios. S. 6.4.5 justifies the particular choice of the pre-processing component of the model. S. 6.5 presents some simulations that suggest that discounted reinforcement learning have some limitations in dealing with long periods of time. Finally s. 6.6 draws the conclusions of the chapter.

6.2 Scenarios of Simulations and the Simulated Robot

The simulations considered in this thesis mainly use two scenarios. The first scenario is shown in Figure 6.1. It is a square arena with sides measuring 1 unit, outside of which there are 4 circular landmarks of different sizes. The second scenario used in the simulations is shown in Figure 6.2. It is again a square arena with sides measuring 1 unit, but this time it has

5 circular landmarks inside. These landmarks are also obstacles for the simulated robot. Other scenarios used in the simulations will be described later.

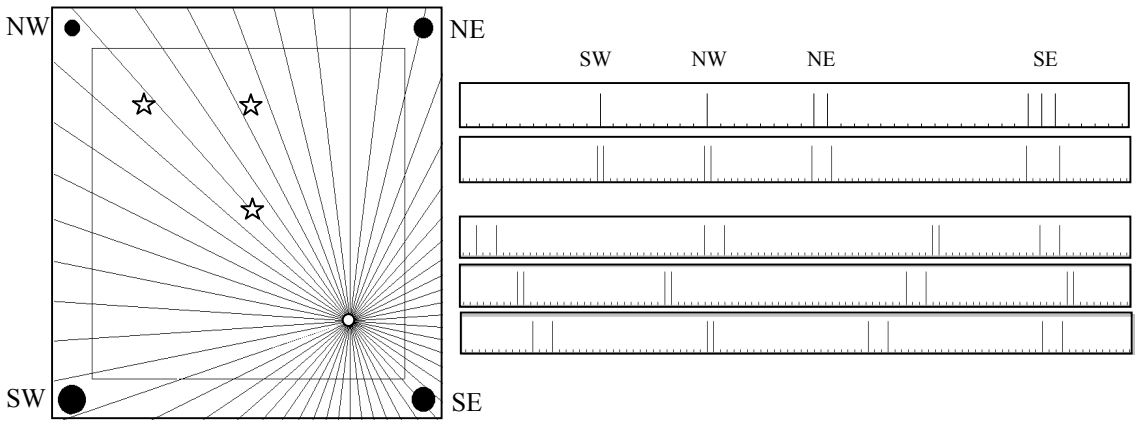


Figure 6.1: Left: one of the scenarios used in the simulations. It contains three possible goals (stars), four landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), and the simulated robot (white circle at origin of rays). The position occupied by the simulated robot is considered as "start position" in some simulations. Right, from top to bottom: the pattern of the visual sensors' activation, its re-mapping into contrasts, the three goal positions north-east, north, centre (as contrasts relative to the images viewed from the goal positions). The letters near the landmarks help to identify their respective positions on the "image" of the retina's activation shown on the right.

The simulated robot can see the landmarks with a one-dimension horizontal retina covering 360 degrees (throughout the thesis, the graphs about the retina show the activation of the sensors clockwise starting from those oriented toward the south). Notice that the simulated robot cannot see a landmark that is behind another landmark, and perceives just a "big" landmark if there are two or more landmarks that are contiguous in sight. The retina is made up by 50 units (vector **x**). Each unit x_i activates with 1 if a landmark is in its scope, with 0 otherwise. This activation is affected by noise (0.01 probability of flipping for each sensor) in the majority of the simulations of this research. It will be explicitly indicated when the entity of this or other sources of simulated noise are different from the ones indicated in this section (cf. s. 6.4.3 for an analysis of some effects of the variation of this and other sources of noise). The signals coming from the retina are always aligned with the magnetic north through a "compass". The reading of the compass is affected by Gaussian noise (0 mean, 1 degree variance) in the majority of the simulations of the research.

Before being sent to the controller, these signals are re-mapped into a vector **y** of 100 binary units representing the image "contrasts" (contrasts between the landmarks, perceived as "black", and the "background", perceived as "white"). Two contiguous retinal units give unit activation to one contrast unit y_j if they are respectively on and off, to another contrast unit if they are respectively off and on, and to no contrast units if they are both on or both off. This simple re-mapping performs *edge detection* and implements an expansion of the input space that allows the controller to work properly by using simple two-layer networks for the controller, in the scenarios considered here (cf. s. 6.4.5 for the justification of this choice). Notice that the simulated robot has a limited perception of the environment's current state (cf. s. 13.2.2 on partially observable Markov decision problems, and Wyatt et al., 1998, for the importance of the "Markov assumption" for mobile robots).

At each cycle of the simulation the controller has to select one of eight actions, each consisting of a 0.05 step in one of eight directions aligned with magnetic north (north, northeast, etc.). The outcome of these actions is affected by Gaussian noise. In the majority of the simulations of this research this noise has a 0 mean and 0.01 variance. If the simulated robot moves against the arena's boundaries or the obstacles it "bounces back", i.e. it is set at the old position.

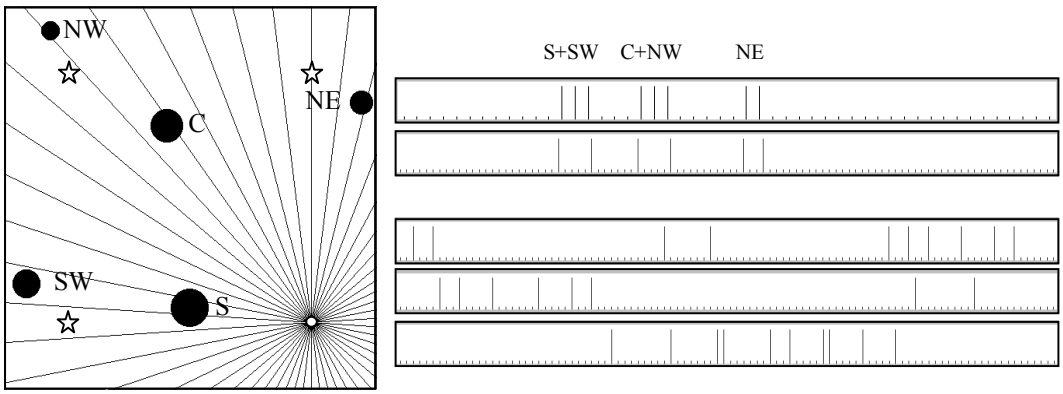


Figure 6.2: Left: a second scenario used in the simulations. It contains three goals (stars), five landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), and the simulated robot (white circle at origin of rays). The position occupied by the simulated robot is considered as "start position" in some simulations. Top right, in the order: the sensors' activation, the corresponding contrasts, and the contrast patterns corresponding to the three goal positions, northwest, northeast, and southwest respectively. The letters near the landmarks help to identify their respective positions on the "image" of the retina's activation shown on the right.

6.3 Architectures and Algorithms

This section illustrates the details of the neural actor-critic controller. The components of this controller are shown in Figure 6.3.

Now the single components are analysed, starting from the matcher. An autonomous robot learning by reinforcement learning is endowed with structures that take input from the environment and yield a "reward" or "punishment" internal signal when some states of the world are achieved that are relevant for the robot itself (e.g. some important resources). When the autonomous robot is planning, it still needs to generate reward and punishment signals to train the actor and critic. However, unlike a reactive robot, the planning robot has to be capable of generating these signals in correspondence to any goal that is assigned to it. In the simulated robot considered here, this is done by the "matcher", a neural network that generates a reward signal by comparing the goal with the "simulated input patterns" generated by the predictor (the details of this will be clarified in chapter 8). To make the simulations with the reactive and the planning simulated robots comparable, this research has used the same matcher for both cases (i.e. for all the experiments reported in the thesis). See also Baldassarre and Parisi (2000) on these issues.

Now the details of the matcher are explained. The architecture of the matcher is showed in Figure 6.4. The matcher is a hand-designed neural network that yields 1 as output when the first part of its input (100 units encoding the goal, i.e. the image of the landmarks from the goal position) is very similar to the second part (100 units corresponding to the current input

or mental image). Otherwise it yields 0. The matcher is composed of 100 sub-networks each taking as input the two bits with same position of the two input parts, and implementing an "if and only if" logic function (00→1, 01→0 10→0 11→1). The output of these sub-networks is then summed, normalised to 1 (dividing it by 100), and compared with a threshold ("recognition threshold") to produce the matcher output (0 or 1). The threshold is usually set at 0.94 throughout the research. This implies that the matcher recognises an input as the goal if they share at least 94% of "bits".

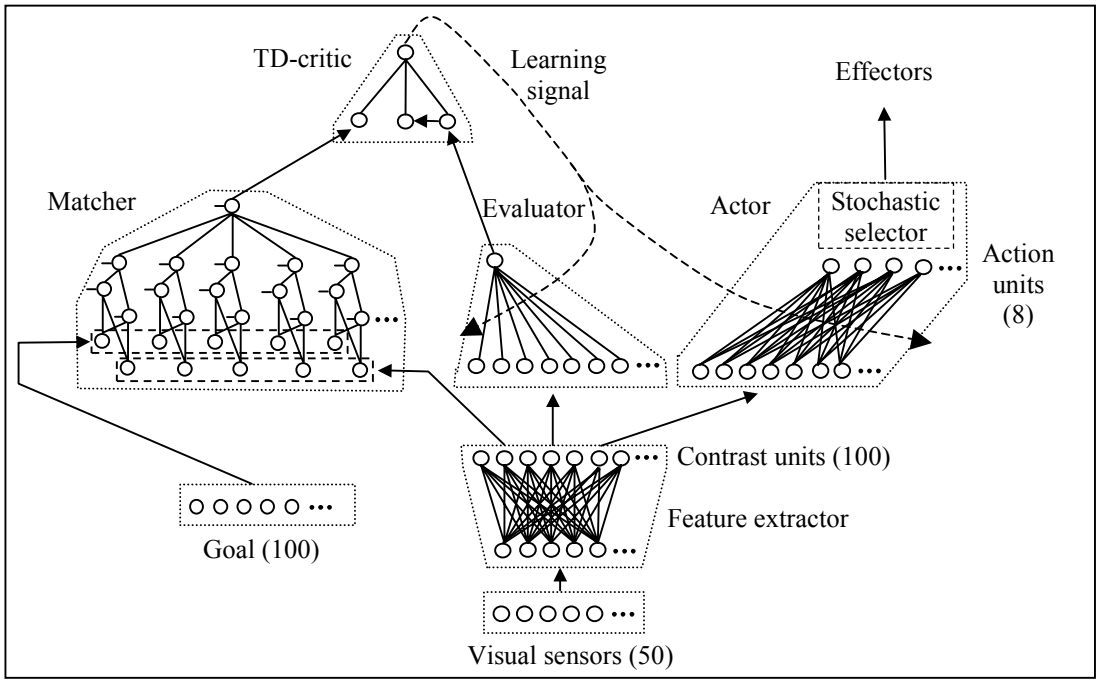


Figure 6.3: Components of the basic neural actor-critic controller. Arrows indicate that a pattern is "copied" from one unit/layer to another unit/layer. Dotted arrows indicate the learning signal. For each layer, only few units have been drawn. The missing units are marked by three aligned black dots. The total number of units of each layer is indicated in round brackets. The horizontal connection within the TD-critic copies the unit's signal with a one step delay (see text). The details of the matcher are expanded in Figure 6.4.

The actor is a two-layer feed-forward neural network that receives the contrast pattern as input and has eight sigmoidal output units that locally encode the actions (cf. Baldassarre and Parisi, 2000, for an actor with a distributed representation of actions). To select an action, the activation m_q (interpretable as "action merit") of the output units is sent to a stochastic selector that implements a stochastic "winner-take-all competition". The probability Pr[.] that a given action a_q becomes the winning action a_{win} (to execute) is:

$$P[a_q = a_{win}] = m_q / \Sigma_f[m_f] \qquad \text{Eq. 6.1}$$

The evaluator is a two-layer feed-forward neural network that receives the contrasts as input and with its linear output unit yields the estimate $V'\pi[yt]$ of the evaluation $V\pi[yt]$ of the contrast pattern yt. $V\pi[yt]$ is defined as the expected discounted sum of all future reinforcements r, given the current actor's action-selection policy π (cf. Eq. 13.9):

$$V^{\pi}[s_t] = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots] \qquad 0 < \gamma < 1 \qquad \text{Eq. 6.2}$$

where γ is the discount factor (set to 0.95 in the simulations of this chapter), and E[.] is the mean operator.

The TD-Critic is an implementation in neural terms of the computation of the Temporal-Difference error e defined as (cf. Eq. 13.15):

$$e_t = (r_{t+1} + \gamma V'^{\pi}[\mathbf{y}_{t+1}]) - V'^{\pi}[\mathbf{y}_t] \qquad \text{Eq. 6.3}$$

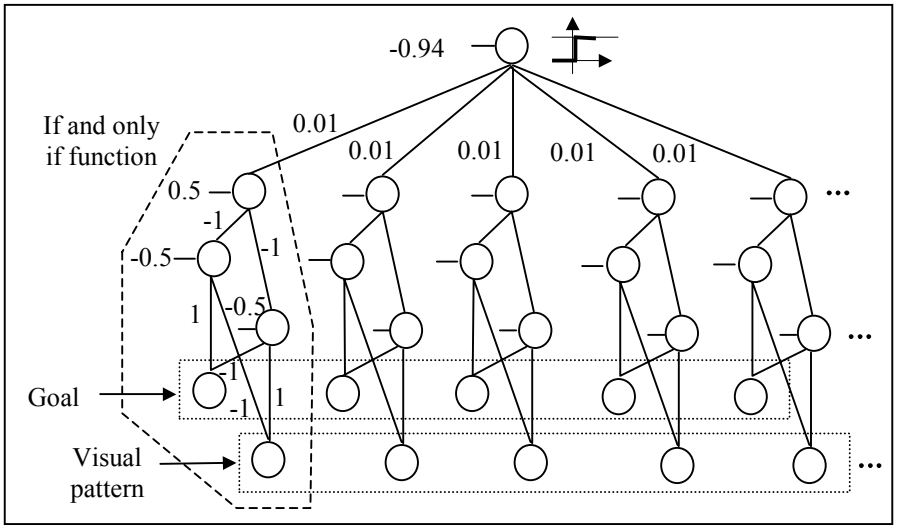


Figure 6.4: Details of the architecture of the matcher. The input of the matcher is made up of two parts, one for the goal pattern and one for the pattern corresponding to the current input (dotted lines). Each couple of bits of the two patterns with the same position are compared by a network implementing an "if and only if" function (dashed line). Only five of these networks are shown in the graph. The output of these networks is summed and compared with a threshold. Each unit implements a step function (shown for the output unit of the whole matcher). Each unit has a bias (little horizontal lines) whose value is indicated in the graph. The hardwired weights of the first of these networks are shown in the graph.

The evaluator is trained with a Widrow-Hoff algorithm (Widrow and Hoff, 1960; cf. s.13.3.1) that uses the error signal coming from the TD-Critic. The weights w_j are updated as follows:

$$\Delta w_j = \eta \, e_t \, y_j \qquad \text{Eq. 6.4}$$

where η is a learning rate (set to 0.1 in the simulations of this chapter) and y_j is the activation of the contrast unit j *at time t*. This implies that the evaluator's estimate is made closer to the target value ($r_{t+1} + \gamma V'^{\pi}[\mathbf{y}_{t+1}]$). This target is a more precise evaluation of $\mathbf{y}_t$ than $V'^{\pi}[\mathbf{y}_t]$ because it is expressed at time t+1 on the basis of the *observed* r_{t+1} and the new estimate $V'^{\pi}[\mathbf{y}_{t+1}]$ (cf. Barto, 1994).

The actor is also trained with a Widrow-Hoff algorithm according to the error signal e_t. Recall that this signal measures the actor's capacity to select actions that bring the simulated

robot to new states with an evaluation higher than the average evaluation experienced previously by moving from that same state. The change of the merits is achieved by updating the weights of the neural unit corresponding to a_{win}, and only this, as follows:

$$\Delta w_{win\,j} = \zeta\, e_t\, (4\, m_{win}\, (1 - m_{win}))\, y_j \qquad \text{Eq. 6.5}$$

where ζ is a learning rate (set to 0.1 in the simulations of this chapter) and $(4\, m_{win}\, (1 - m_{win}))$ is the derivative of the (sigmoidal) transfer function multiplied by 4 to homogenise the size of the learning rates of the actor and the (linear) evaluator (in fact the maximum of that derivative of the transfer function is 0.25 for the actor, and 1 for the evaluator).

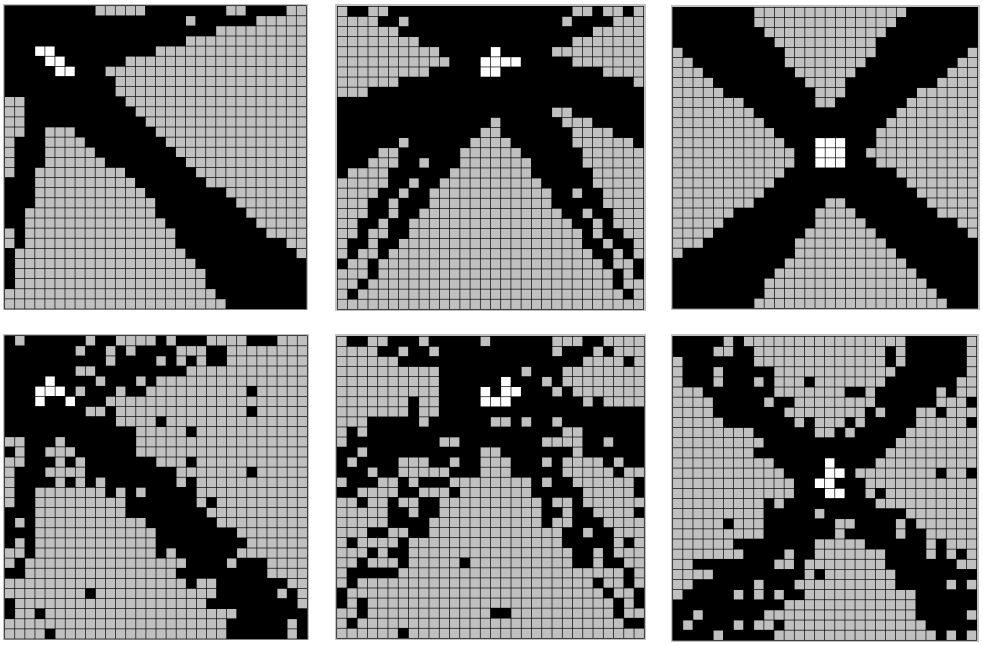


Figure 6.5: Case of the scenario with 4 landmarks outside the arena. The graphs show the activation of the matcher for the three goals (first, second and third columns of graphs), without (top row) and with (bottom row) noise of the sensors and the compass. Each single graph shows the output of the matcher in 30×30 different positions of the simulated robot on the arena (grid). White cells mark positions that the matcher recognises as goal (here the mather's output is 1). Black cells mark positions similar to the goal where the activation potential of the matcher's output unit (before being compared with the threshold) is over 0.84. Grey cells mark positions where such activation is below 0.84.

Particular attention has to be paid to the training of the evaluator when the tests are broken into "*trials*", for example when the simulated robot is set at a new position chosen randomly after it reaches the goal. In this case problems may arise. This is true in general, but using generalisation methods as neural networks exacerbates the problems. In particular Eq. 6.3, Eq. 6.4, and Eq. 6.5 need to be modified in two ways at the end and beginning of each trial:

- When the reward is 1 at the end of a trial (the goal has been reached) $V'^{\pi}[\mathbf{y}_{t+1}]$ needs to be set at 0. In fact the evaluator will tend to return a $V'^{\pi}[\mathbf{y}_{t+1}] > 0$ because $\mathbf{y}_{t+1}$ tends to be similar to $\mathbf{y}_t$. This causes the target $(r_{t+1} + \gamma\, V'^{\pi}[\mathbf{y}_{t+1}])$ used to adjust $V'^{\pi}[\mathbf{y}_t]$ when the goal is reached at t+1, to grow above 1. In turn $V'^{\pi}[\mathbf{y}_{t+1}]$ grows even more following $V'^{\pi}[\mathbf{y}_t]$ (again because $\mathbf{y}_{t+1}$ is similar to $\mathbf{y}_t$) generating a dangerous positive feedback that in some

conditions leads the evaluations to grow indefinitely. Setting $V'^{\pi}[\mathbf{y}_{t+1}]$ to 0 produces a stable gradient field with evaluations equal to 1 for the states around the goal, and evaluations that decrease smoothly for states progressively more distant from the goal.

- When a new trial starts, i.e. when the simulated robot passes from the last state of the previous trial to the first state of the new trial, both the learning coefficients of the actor and the evaluator have to be set at 0. This has to be done because there is not an "old evaluation" to be updated on the basis of the current evaluation, and because there is not an "old state" with regard to which the action's probabilities need to be updated.

At the beginning of the simulations, the weights of the evaluator and actor are randomised within the interval [-0.001, +0.001], so the evaluations expressed by the evaluator's linear output unit are around 0, and the merits (and probabilities) expressed by the actor (and stochastic selector) are around 0.5 (and 0.125). This implies that initially the simulated robot explores the environment randomly, and then it starts to shape the evaluations on the basis of the rewards and the probabilities on the basis of the evaluations.

6.4 Results and Interpretations

6.4.1 Functioning of the Matcher

This section illustrates the functioning of the matcher with and without the noise affecting the sensors and the compass. Figure 6.5 shows the output of the matcher in different conditions for the three goals illustrated in Figure 6.1. Figure 6.6 shows the output of the matcher in different conditions for the three goals illustrated in Figure 6.2. Three relevant facts become apparent from these graphs.

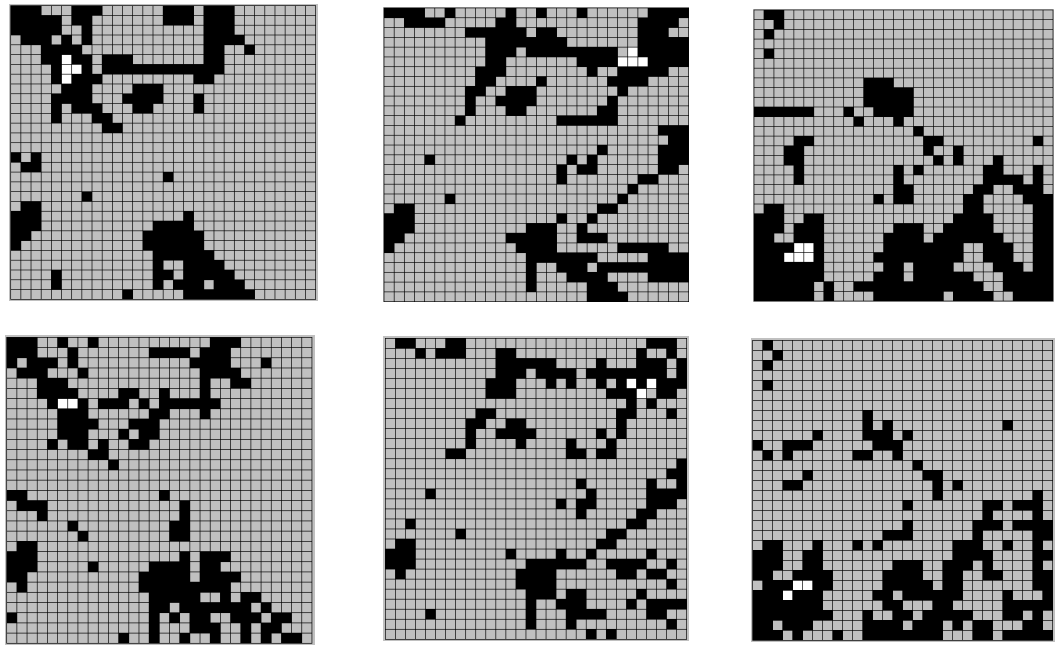


Figure 6.6: Case of the scenario with 5 landmarks inside the arena. The graphs show the activation of the matcher for the three goals (three columns) without and with noise (two rows). The interpretation of the graphs is as in Figure 6.5.

The first is that the Matcher is capable of distinguishing between input patterns very similar to the goal and input patterns different from it. The recognition threshold of the matcher can be regulated so that the sensitivity of the matcher is the desired one.

The second relevant fact is that the matcher shows an acceptable robustness for the noise affecting the input and the compass. Unluckily, the simulations have also shown that the noise affecting the pattern of the *goal* is quite disruptive, especially for planning (cf. later chapters). This problem has been successfully solved throughout the research by taking an "average input pattern" as goal, so to eliminate noise. In particular to build this average pattern, the simulated robot has been set at the goal position, 30 noisy input patterns have been recorded, and then averaged bit by bit. Then to build the pattern used as goal the average for each bit has been compared with 0.5. Each bit of the goal pattern has been set at 1 or 0 if this average was respectively above or below 0.5.

The third interesting fact is that the graphs give a good idea of the potential generalisation properties of the neural controller based on the use of the (contrast) features. The case of the scenario with 4 landmarks outside the arena (Figure 6.5) clearly shows that the controller can strongly generalises along the lines that take from one particular position to the landmarks. For example (cf. graphs) several positions between the goal and a particular landmark are considered similar to the goal position by the controller because from them the same landmark is visible in the same direction. Figure 6.6 shows that with the more complex scenario (landmarks inside the arena), the possible generalisation is more fragmented. As we shall see below, in this case generalisation still produces positive effects in terms of learning speed (cf. s. 6.4.2) but it also causes some confusion between different but similar states (cf. s. 6.4.3).

The illustration of a drawback of the matcher concludes this section. The drawback is the need to regulate the parameter of the recognition sensitivity. At least with the simple simulated robot's sensory apparatus adopted here, this parameter has proved to be very sensitive. In some cases of the simulations illustrated in this thesis it has been necessary to set it at a value different from the one generally used (0.94) because otherwise the area of recognition was too wide or too small. For example this has been the case of the northeast goal shown in Figure 6.2, for which the value of the parameter has been set at 0.96.

6.4.2 Performance of the Controller: The Critic and the Actor

Now the attention is directed to analysing both the behaviour of the evaluator, the major component of the critic, and the performance of the actor. Two types of simulations have been run to this purpose. In the first type of simulation, divided into trials, the simulated robot had to reach one goal position. When this happened, the simulated robot was set at another position chosen randomly, and another trial started. This was repeated several times. The second type of simulation was similar to the previous one, with the difference that the simulated robot was always set at the same start position at the end of each trial. Both simulations have been run with three different goals and by using the two scenarios illustrated in Figure 6.1 and Figure 6.2. The two Figures the scenarios also show the position of the start position and the three goals.

The first data collected are relative to the functioning of the evaluator in the case of random restart. Figure 6.7 shows the gradient field of the evaluations for the three goals in the case of the simple scenario after the goal has been reached the first time, after 300 steps from this event, and when the performance has reached a steady state (see below).

Some facts become apparent from these graphs. The most important is that the generalisation capacity of the controller implies that *reaching* of the goal once is sufficient to

update the *evaluations of several states*, not only the one directly preceding the goal state (cf. top row of graphs). This confirms what anticipated through the analysis of Figure 6.5. This generalisation capacity of the controller is very important because it speeds-up learning (The importance of generalisation for speeding up learning processes is considered very important in the literature, cf. s. 13.2.8).

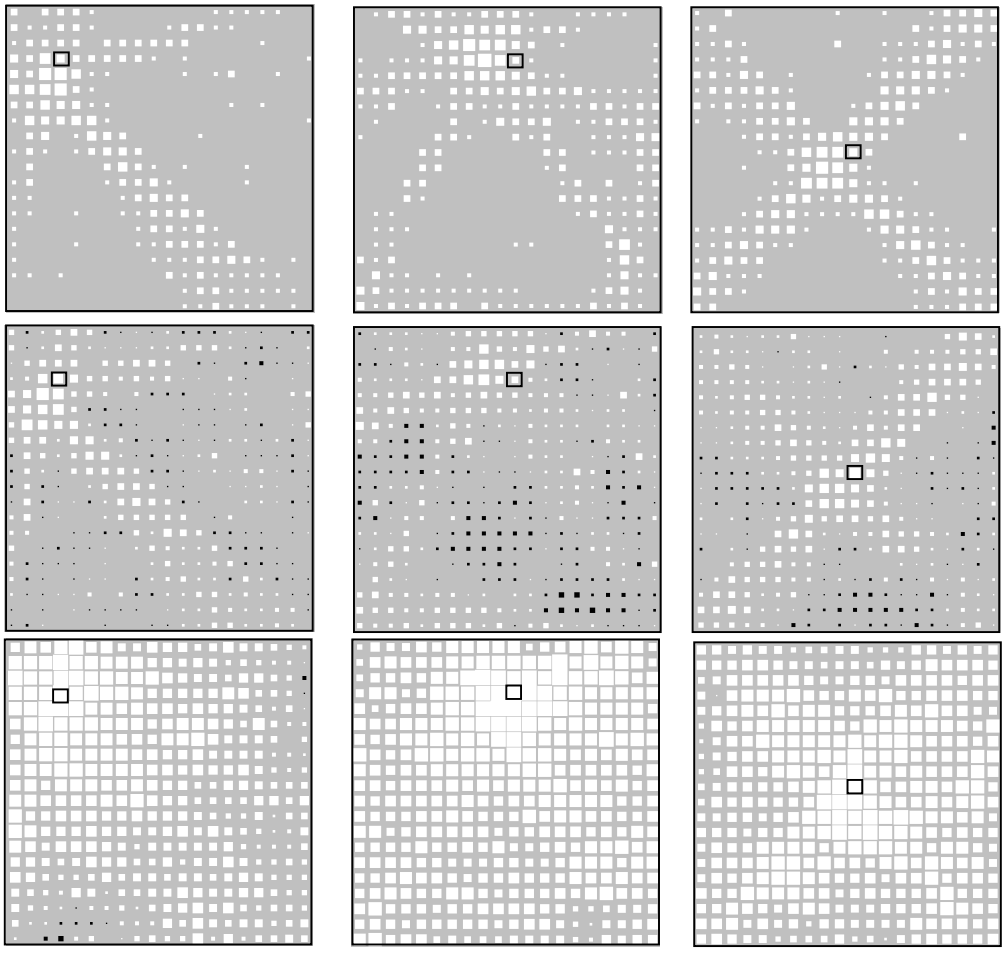


Figure 6.7: The graphs shows the gradient field of evaluations $V'^{\pi}[\mathbf{y}_t]$ for the three goals (three columns) of the four landmarks scenario. For each graph, the cell with a bold border indicates the position of the goal pursued. The first row of graphs indicates the gradient field right after the goal has been reached for the first time. The second row of graphs indicates the gradient field 300 steps after this event. The third row of graphs indicates the gradient field at the end of training. To build each graph the simulated robot has been set at 20×20 different positions on a grid overlapped to the arena, and the evaluation yielded by the evaluator has been measured. The size of the white (or black) squares is proportional to the positive (negative) evaluation given in that position. The big white cells scattered irregularly in the graphs are caused by temporary noise of the sensors. In the first and second row of graphs the gradient field is not precisely centred on the goal because the simulated robot has built it on the basis of the state from which the goal state has been encountered, and because the area recognised as goal is often bigger than a single cell (cf. Figure 6.5).

The generalisation property of the controller has also some costs. One is that some of the evaluations assigned to some states through generalisation may be wrong. This can be seen by considering the middle row of graphs of Figure 6.7. These graphs show the gradient field of the evaluations 300 steps after the goal has been reached. Some negative evaluations can be

seen (black cells), while the evaluations over the entire arena should be positive, since only positive rewards are involved in the task. This is caused by the fact that some of the evaluations assigned to some states through generalisation after the goal has been reached, are too high. As the simulated robot visits these states and *directly* (i.e. not on the basis of generalisation) updates their evaluations, these evaluations are lowered (this can be clearly observed while the simulations are running). Generalisation implies that the evaluations of other similar states are lowered too. If these evaluations were near 0, as they are for some states (see first row of graphs), the net effect is that they are pushed towards negative values. Summarising, the negative evaluations visible in the graphs reveal evaluation errors caused by the generalisation property of the controller. Luckily, the controller is capable to correct these errors as the training goes on, as shown by the bottom row of graphs of Figure 6.7.

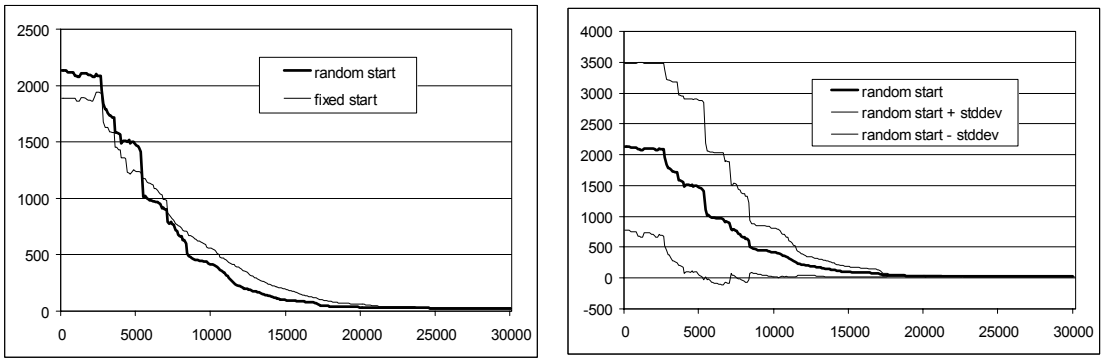


Figure 6.8: Learning curves for the first scenario with four landmarks outside the arena. Y-axes: number of steps per goal reached, averaged over the last 100 successes (a simple average is considered before 100 successes are accumulated), and averaged over 10 simulations run with different random seeds. X-axis: steps. Left: performance of the simulations run by setting the simulated robot at a randomly chosen start or to the same start at the end of each trial. Right: graph that shows the standard deviation of the simulations run by setting the simulated robot at a random start.

Figure 6.8 shows the learning curves for the two types of simulations described previously (random and fixed restart) for the first goal. The graph shows the performance of the simulated robot measured in terms of steps taken to reach the goal. For each run this measure is averaged over the last 100 goals reached (at the beginning, with less than 100 goals reached, a simple average has been used) and then averaged over 10 simulations run with different random seeds. From the graph it is apparent that the performance of the simulated robot improves from about 2000 steps per success, to about 25 steps per success. Since the robot's step is 0.05 long and the arena is 1 by 1, the optimal path to the goal is on the average about 10-15 steps long, both for the random and the fixed restart conditions (but consider that action noise and obstacles make the task more difficult). The right end of the figure gives an idea of the variance of this kind of simulations when they are run with different random seeds. This is useful to interpret the results of the following chapters when different controllers are compared.

Figure 6.9 shows the behaviour of the evaluator in the case of the second more complex scenario with landmarks inside the arena. It can be seen that also in this case the generalisation property of the controller allows the system to attribute an evaluation different from 0 to many states after the goal has been reached one time only (top row of graphs). This evaluations make up a gradient field that has a certain tendency to decrease for states progressively more distant from the goal, but it is also much more irregular than the case with

the simple scenario. The middle row of graphs shows that the controller has to do several adjustments to correct the evaluations (there are many negative evaluations). When the training goes on for several trials, the controller adjusts most of the "wrong" evaluations, but is not capable of fully correcting them. In fact the bottom row of graphs shows that there are some residual negative evaluations, and that the positive evaluations are still quite irregular. This is caused by both the simplicity of the contrast re-mapping and the neural architecture of the evaluator that is based on a simple two layer neural network with few degrees of freedom (cf. discussion in s. 6.4.5). An attempt to further analyse the reasons for the shape of the final gradient field in the case of complex scenarios is done in s. 6.4.2.

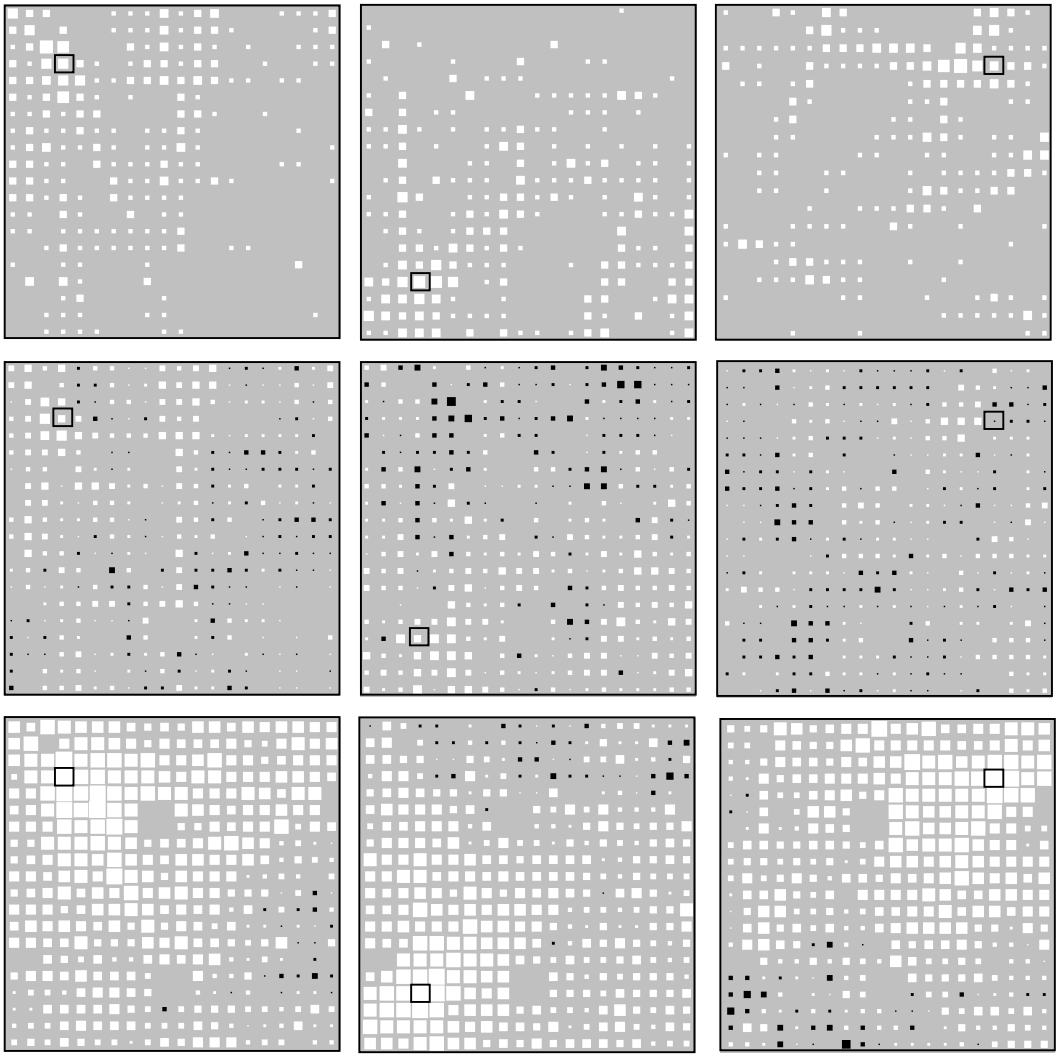


Figure 6.9: The graphs shows the gradient field of the evaluations $V'^{\pi}[\mathbf{y}_t]$ for the three goals (three columns) of the complex scenario with landmarks inside the arena. Cf. the label of Figure 6.7 for the interpretation of the graphs.

Figure 6.10 shows the learning curves of the controller in the case of the more complex scenario, for the first goal (northeast). The first interesting fact is that the controller is capable of coping with the irregularity of the evaluation gradient field just considered and to solve the task. In fact, the performance of the simulated robot improves from about 1400 - 1600 to about 25-40. The second fact is the difference between the random start and the fixed start

simulations. Why with the complex scenario is there this difference while there was no difference for the simple scenario (cf. Figure 6.8)? A tentative explanation is that the random restart favours the correct updating of all the evaluations, and this in turn favours a correct training of the actor. In contrast, the fixed-restart simulations predominantly visit a few particular states, causing problems when other states are visited (cf. Barto et al., 1995). This difference is more important for the complex scenario because, as we have seen, generalisation tends to produce less correct evaluations.

The difference of the initial performance between the simple and complex scenario simulations is not significant given the large variance of the performance itself at the beginning of the simulations.

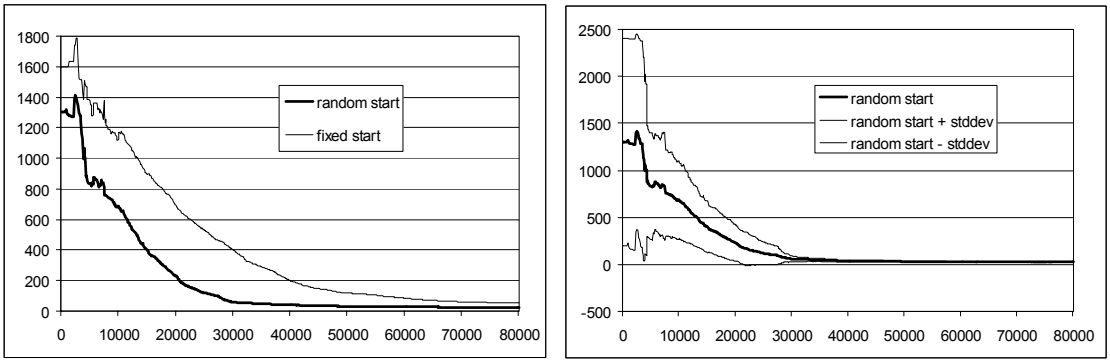


Figure 6.10: Left: learning curves with the complex scenario with landmarks inside the arena, for both the random and the fixed start conditions. The y-axis indicates the number of steps per goal reached, averaged over the last 100 successes (a simple average is considered before 100 successes are accumulated), and averaged over 10 simulations run with different random seeds. The x-axis indicates the steps. Right: graph that shows the standard deviation of the simulation run with the random start condition.

This section is concluded with an observation on the issue of the "TD(λ)" algorithm (Sutton and Barto, 1998, p. 163-191) that parallels the generalisation property of controllers that use function approximation methods (cf. s. 13.2.8). The TD(λ) algorithm is a reinforcement learning algorithm that allows the controller to extend the effects of one backup to many states visited before the current state. As we have seen, the generalisation property of the neural controller used here was already capable of extending the effect of one backup to several other similar states. These effects are similar to the effect that the TD(λ) algorithm would have produced. Indeed, a preliminary exploration of the TD(λ) algorithm within the scenarios considered here has shown that the algorithm did not improve the performance significantly, because generalisation already extended the effects of learning to several states. For these reason the TD(0) algorithm has been adopted in the rest of the research.

6.4.3 Aliasing Problem and Parameters' Exploration

This section concentrates on the "aliasing problem" (cf. s. 13.2.2) and on the exploration of some parameters of the controller.

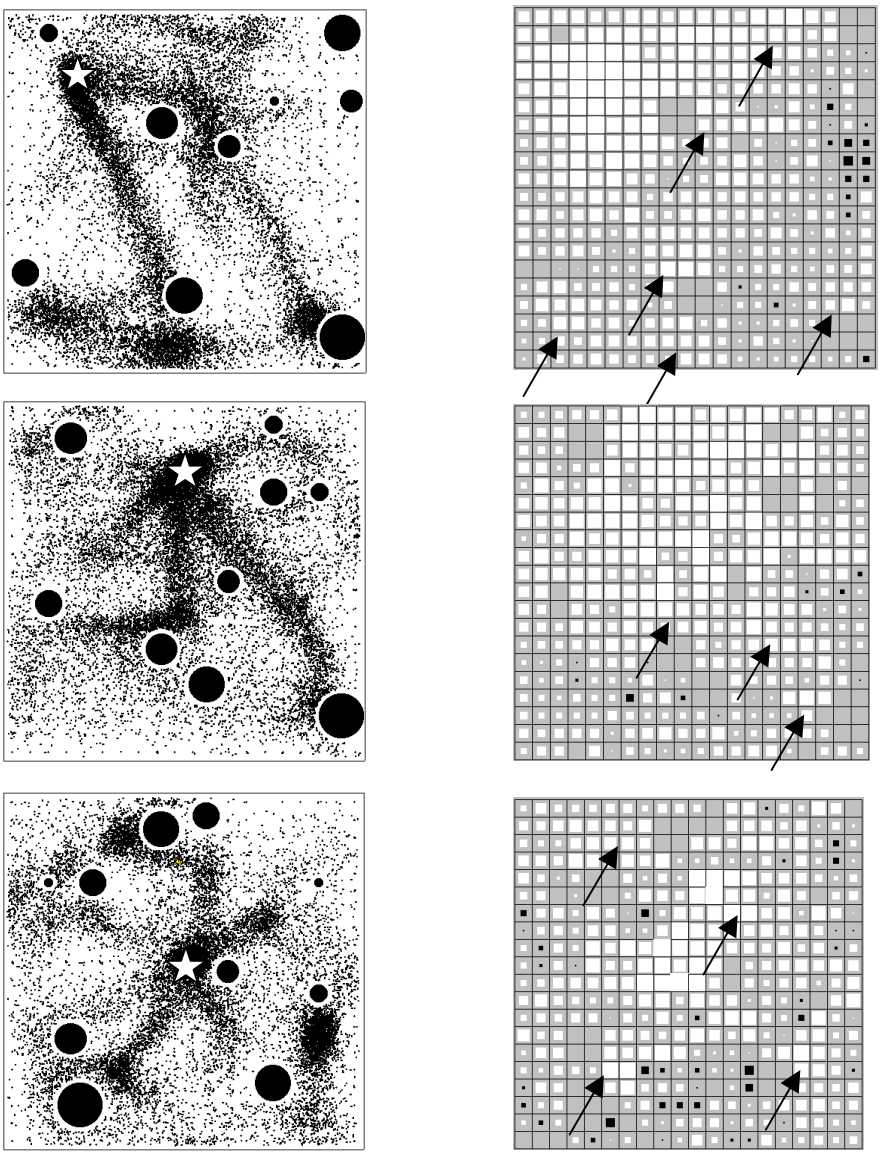


Figure 6.11: Results of three simulations run with the three different goals (indicated by stars) and various scenarios. The three rows of graphs refer to the three goals. Left column of graphs: scatter of the positions occupied by the simulated robot, where one dot indicates one position occupied by the simulated robot. The dots have been collected running the simulations for several cycles after the simulated robot achieves a steady good performance. The higher the relative density of the dots on an area of the arena, the more likely that the simulated robot occupies that area of the arena while moving. Right columns of graphs: evaluation gradient fields corresponding to the scenarios/tasks showed on the left column. The arrows highlight the areas where the evaluations appear high in comparison to their surroundings. Compare these areas with the scatter graphs on the left: they tend to correspond to areas with a high density of dots.

Aliasing Problem. The aliasing problem originates from the simplicity of the sensory apparatus of the simulated robot. This returns only partial information about the current state of the environment, for example the position currently occupied by the simulated robot. This can generate confusion between states that are different but appear to be similar through the simulated robot's sensory apparatus. Figure 6.11 shows that the aliasing problem is one of the major causes of the irregularity of the gradient field observed in Figure 6.9. They show that the problem is particularly impairing when some positions are very similar to the goal

position. In fact in these cases the evaluator tends to assign high evaluations to them, as it does for the goal position. This gives a wrong signal to the actor that learns to drive the simulated robot toward positions with high evaluations.

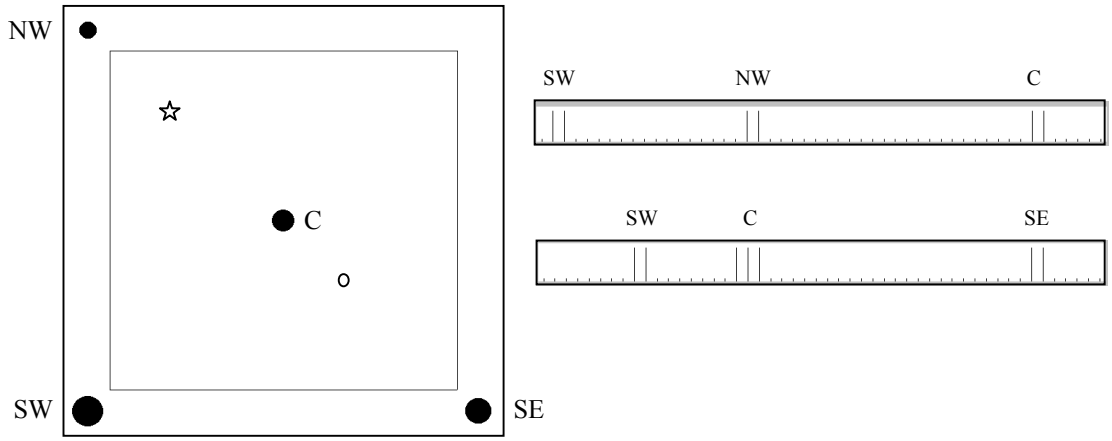


Figure 6.12: Left: Scenario with one goal (star) that creates an aliasing state (indicated by the white circle). Right: The activation of the sensors at the goal position (top) and at the "aliasing" position (bottom). Notice that the two images differ only for the position of one landmark, the one at the southwest corner, while the other two landmarks visible in the two situations appear quite similar even if they are different. The letters near the landmarks help to identify their respective positions on the "image" of the retina's activation shown on the right.

To confirm this interpretation of the previous results, a simulation has been run by using the particular scenario shown in Figure 6.12. In this scenario there is an area that generates a view very similar to the view from the goal position, hence it should be strongly affected by the aliasing problem. The simulation consists of a sequence of many trials with random start.

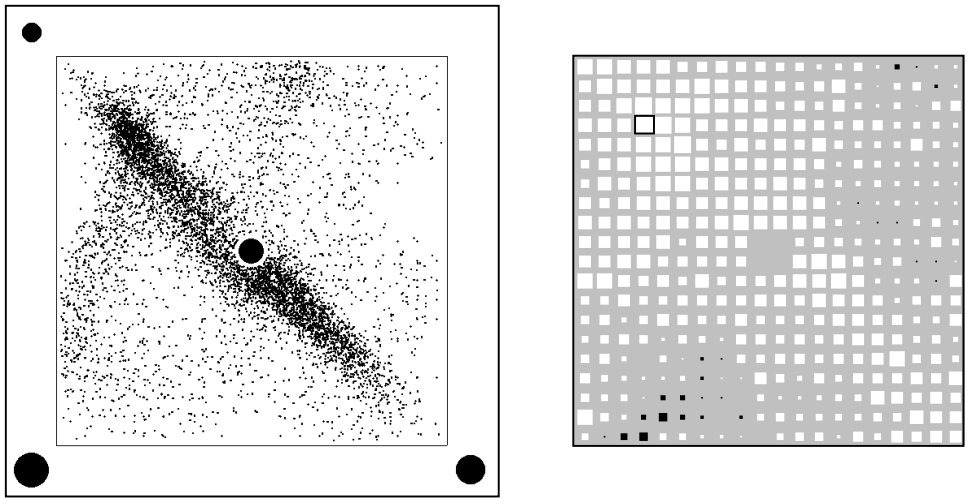


Figure 6.13: Left: Scatter graph of the positions occupied by the simulated robot in the "aliasing" simulation when the performance has stabilised. Right: Evaluation gradient field.

Figure 6.13 shows the scatter graph of the positions occupied by the simulated robot when the performance has stabilised, and the corresponding evaluation gradient field. While

the performance is still good (about 25) the scatter graph shows that the simulated robot is "attracted" by the "aliasing area" that has an image similar to the image of the goal area. The right part of the figure shows that the reason of this is that the evaluations are high in this area.

Probably there are several sources of errors that explain the irregularity of the gradient fields shown in Figure 6.9 and Figure 6.11, for example the small number of degrees of freedom available to the controller and the high level of the learning rates used (cf. s. 6.4.5 and last part of this section). However, the aliasing problem seems to be among the most important ones.

6.4.4 Parameter Exploration

Some other simulations have been run to explore the effects of using parameters with values different from the ones used in the previous simulations.

If the noise of the compass and the sensors is augmented, the aliasing problem gets worse: doubling their size (variance and probability respectively) prevents learning from converging because the simulated robot gets stuck in areas with high erroneous evaluations. Compass noise is particularly disruptive since the system is not capable of adjusting image rotations. The controller is very robust with respect to the noise of the effectors. Doubling its variance does not prevent the critic and actor from converging.

Augmenting the number of sensors improves the quality of the gradient field, but only if there is no noise in the sensors and the compass. In this respect, the noise in the compass is very disruptive since the advantages of having many sensors, each with a small scope, are eliminated if the sensors have a varying alignment with magnetic north.

The learning rates are quite important. High levels of the learning rates, around 0.1, such as the ones used in this section, allow a quick convergence of the evaluations, as shown by the first row of graphs of Figure 6.7 and Figure 6.9, and hence of performance. However, they also have three drawbacks, observed in several simulations. One is that they make the gradient field quite unstable and changeable in the course of the simulations. The second is that if they are too high (more than 0.1) they cause the evaluations to "explode" towards high positive and negative values. The third is that they augment the negative effects of the aliasing problem: the simulated robot gets stuck in areas where the evaluation gradient field has local maxima more easily. In the course of the research the learning rates have usually been chosen as high as possible but low enough to keep these drawbacks under acceptable limits.

6.4.5 Why the Contrasts? Why no more than the Contrasts?

Now there are enough elements to justify the use of the "contrast pre-processing" illustrated in s. 6.2. We have seen that the aliasing problem is quite impairing when complex scenarios with landmarks inside the arena are used. It has been found that directly using the sensors' activation pattern as input for the evaluator and actor worsens the situation. The reason is that the local maxima of the evaluation gradient field illustrated previously are caused not only by the aliasing problem, which depends on the limitations of the simulated robot's sensors, but also by the small amount of degrees of freedom of the resulting evaluator and actor. In particular with 50 sensors and no contrast re-mapping, the evaluator has only 50 weights. This has the effect that the same weights are used for situations that are similar, and this in turn worsen the aliasing problem.

With the contrasts re-mapping, the evaluator and actor has 100 weights. This means that the expansion of the input space into a bigger space before feeding the networks augments the

degrees of freedom of the two networks (cf. also Haykin, 1999, p. 257-258). Moreover, contrary to the previous situation where all the sensors corresponding to a landmark are active at each time step, few features are now active, namely only the ones that correspond to the two edges of each landmark. This allows the features to be more selective, and so alleviates the aliasing problem.

In the previous section we have seen that, nothwithstanding this improvement, the aliasing problem is still present when the contrast pre-processing is used. Why was it decided not to use more sophisticated forms of pre-processing, such as the ones listed in s. 12.3? This has not been done for several reasons:

- The focus of this research was not the solution of the aliasing problem.
- The solution of the contrasts is very simple, so it does not complicate the interpretation of the results about planning, presented in the following chapters.
- The solution using contrasts is computationally very fast, so it has made it possible to run the numerous experiments illustrated in the following chapters in an acceptable amount of time.
- With few precautions, it has been possible to keep the aliasing problem under control. In particular it has been decided to use the scenarios presented in Figure 6.1 and Figure 6.2. These are scenarios where the aliasing problem is not so impairing. This is shown in Figure 6.14 that shows the scatter graph and the evaluation gradient field for one goal of the second of these scenarios, the more complex one. The situation is similar for the other goals used throughout the research within this scenario.
-

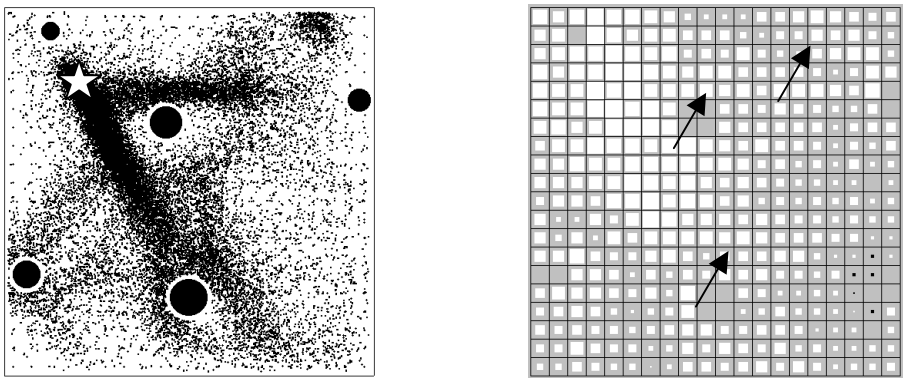


Figure 6.14: A scenario that is only slightly affected by the aliasing problem, as the scatter graph and the evaluation gradient field on the right show. Cf. the label of Figure 6.11 for the interpretation of the graphs.

6.5 Temporal Limitations of Discounted Reinforcement Learning

This section focuses on the capacity of discounted reinforcement learning to deal with problems whose solutions last for long periods of time. “Discounted” reinforcement learning is the most popular form of reinforcement learning (cf. Sutton and Barto, 1998), used throughout this research and illustrated in chapter 3. In this kind of reinforcement learning method the *optimal* evaluations decrease exponentially for states progressively more distant from the goal (cf. Figure 5.3). This section shows some experiments that suggest that this form of reinforcement learning has a limited capacity to deal with problems whose solution

last for long periods of time. In fact the states far from the goal have evaluations that are close to 0. As a consequence the learning signal built on the basis of two of such evaluations relative to two contiguous states (cf. Eq. 6.3) is also close to 0. This has two negative consequences: (a) learning based on this signal is slow; (b) the signal can be easily disrupted by noise.

A simulation has been run to support these ideas. The scenario used in this simulation is shown in Figure 6.15. Unlike the previous sections, the simulated robot is now endowed with 100 visual sensors (200 contrast units). These sensors and the compass are *not* affected by noise. The simulated robot moves in a one-dimensional space, say a corridor. At the left end of the corridor there is the start state, while at the right end there is the goal state.

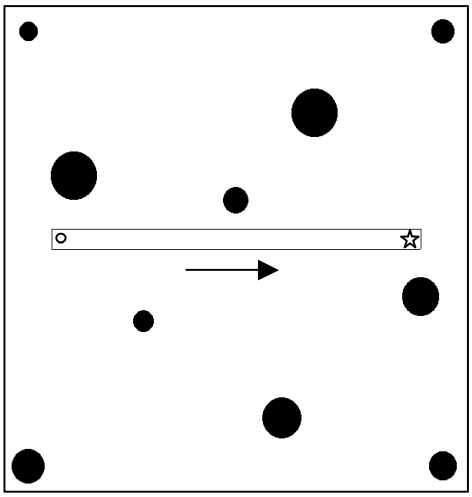


Figure 6.15: The scenario used to test the capacity of discounted reinforcement learning to deal with long periods of time. The black circles are the landmarks, the white circle is the start, the star is the goal, and the empty rectangle containing the start and the goal is the corridor along which the simulated robot is transported by the trolley. The trolley moves along the direction shown by the arrow.

The simulated robot has only two actions: move_east and move_west. The simulated robot does not move autonomously. During each trial the simulated robot is "transported on a trolley" from the start to the goal following a straight line, for the reasons explained below. When the trolley with the simulated robot reaches the goal a new trial starts. Within a trial, the trolley reaches the goal in 100 steps.

During these trials, the evaluator of the simulated robot is trained as usual, so after some trials the evaluations should converge to the optimal values and their gradient field should assume a typical exponentially decaying shape. At the same time the actor is trained in a special way. At each state the merit of the two actions is updated as if the simulated robot had selected both actions. This means that the go_left action is updated with an error built on the basis of the evaluations of the current and the previous state, while the go_right action is updated with an error built on the basis of the evaluations of the current and the following state. The use of the trolley and this special updating of the actions' merit are used to eliminate the effects of different frequencies of visit of the states, and the effects of different frequencies of the selection of actions, that would been produced by the simulated robot acting autonomously. This would make the interpretation of the results less clear.

Figure 6.16 shows the simulated robot's evaluation gradient field over the 100 states occupied by the trolley when it goes from the start to the goal. This figure shows the gradient field after 5, 50, and 500 trials. The top graph of the figure shows that the evaluations for the states near the goal have started to assume an exponentially decaying shape.

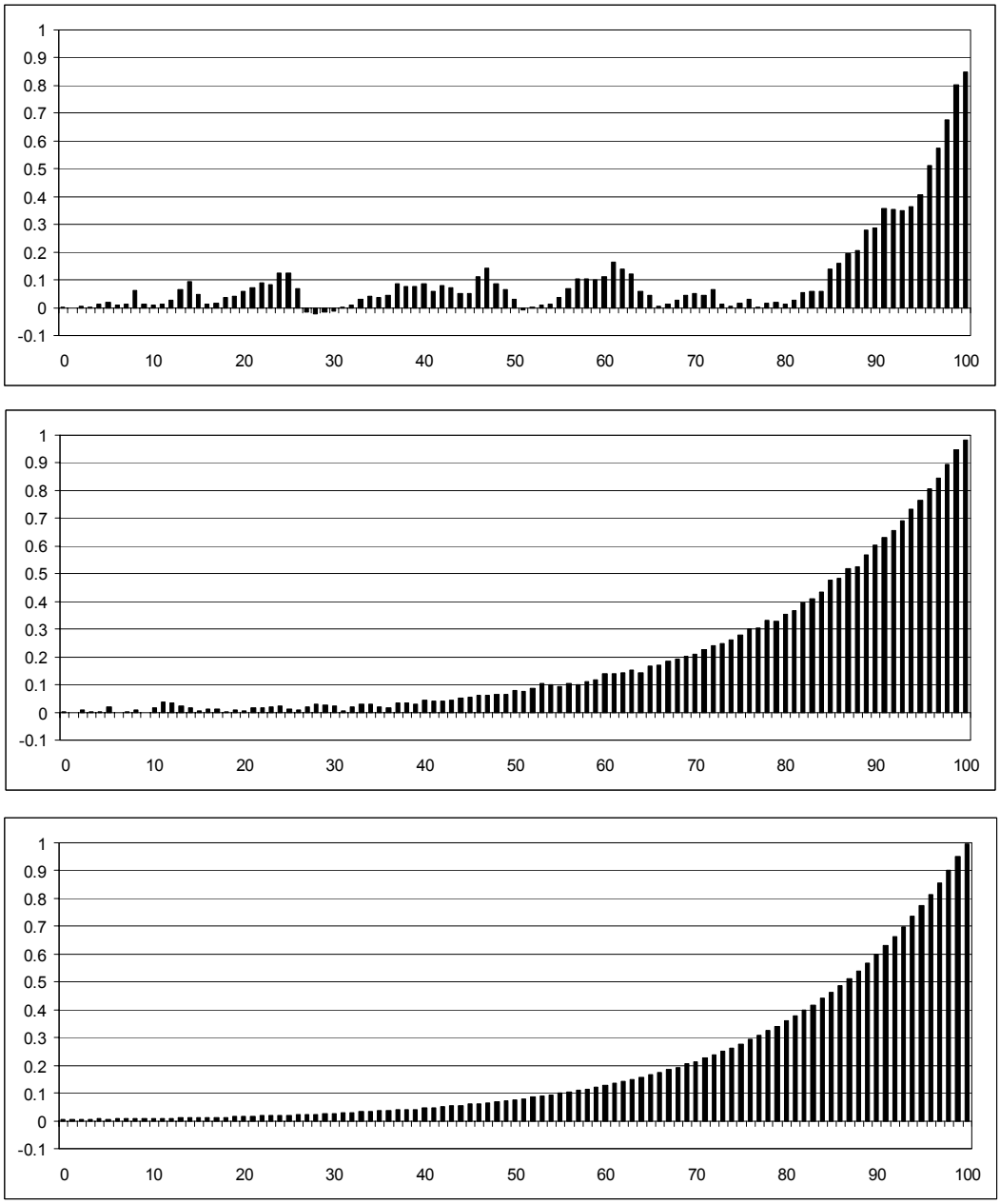


Figure 6.16: Evaluations (y-axis) for the 100 states (x-axis) visited by the simulated robot on the trolley in each trial, after 5 (top), 50 (centre), 500 (bottom) trials. State 0 on the x-axis is the start while state 100 is the closest to the goal.

Figure 6.16 also shows that the generalisation property of the evaluator allows a rapid diffusion of the values backward: states more than 5 steps far from the goal, that without generalisation would have evaluations equal to 0, have positive evaluations. However the graph also shows that generalisation has some drawbacks. For example the three “waves” of

high evaluations before the final wave close to the goal, are caused by the fact that the simulated robot encounters some landmarks that are in the same direction/position of other landmarks viewed from the goal position. The evaluation of these positions is high because the image that they produce is similar to the image at the goal position (aliasing problem).

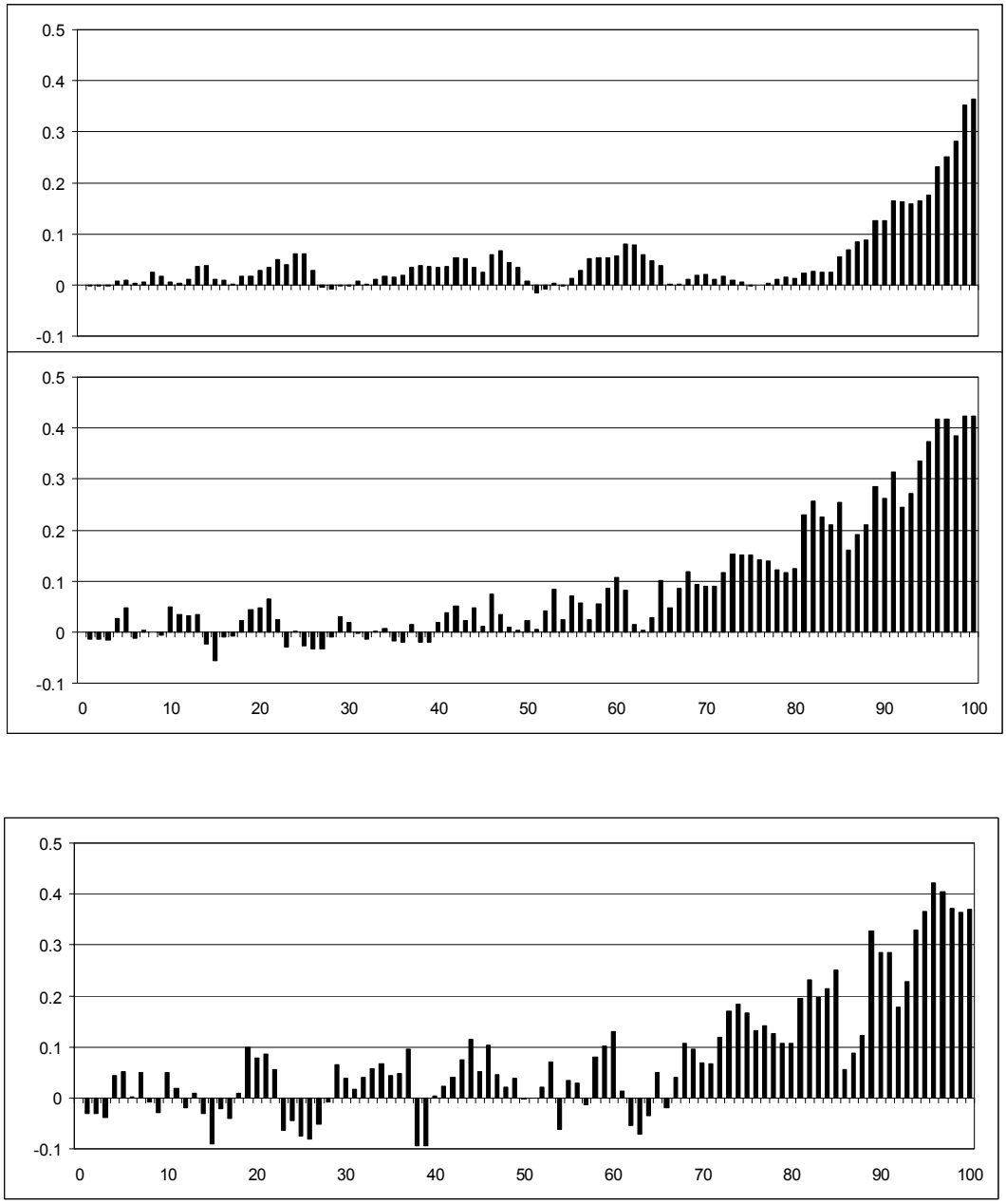


Figure 6.17: Difference between the probabilities of the go_east and go_west actions (y-axis), in correspondence to the 100 states visited by the trolley and the simulated robot (x-axis), after 5 (top), 50 (centre), 500 (bottom) trials.

The central graph of the figure shows that after just 50 trials the evaluation gradient field has almost assumed its final shape, even if its quality is still better for states closer to the goal. The final accurate shape assumed by the gradient field is shown in the bottom graph, plotted after 500 trials.

While the evaluations change towards their optimal values, the merits and probabilities of actions are updated according to the learning signal of Eq. 6.3. Figure 6.17 shows the difference between the probabilities of the go_west and go_east actions corresponding to the 100 states visited, again after 5, 50, 500 trials. This difference is about 0 at the beginning of the simulations, since the two actions have the same merits and probabilities of being selected, and should approach 1 at the end of training, as go_east is the optimal action for all the states visited.

The top and central graphs clearly show that the difference of the probabilities is much higher, hence more correct, for the states closer to the goal, while it is low and even negative for states far from the goal. This is what the experiment was intended to show. The low evaluations for the states far from the goal have two effects:

- They cause slow learning.
- They are easily affected by noise caused by wrong generalisation, unreliable sensors, etc., so that they generate a wrong learning signal for the actor.

The probabilities shown in the bottom graph are even worse than the ones of the central graph. This is an artefact of the experiment, and is caused by the fact that the evaluations are updated according to the optimal policy and not according to the actual action probabilities expressed by the actor, as required by the actor-critic algorithm. In fact when the evaluations approach their optimal values, the learning signal of Eq. 6.3 becomes 0 and the training of the actor stops or continues in the wrong direction.

What happens if the discount coefficient is increased, e.g. to 0.99, to avoid that states far from the goal are close to 0? This has been done in a preliminary experiment, and the result has been that the quality of the actions' probabilities associated the last 100 states before the goal, improves. However, this does not solve the problem, it only moves it to states more distant from the goal. In fact if the discount coefficient is smaller than 1, the evaluations will decay exponentially and will get close to 0 for some states distant enough from the goal. On the other hand, the discount coefficient cannot be increased too much (near 1) to face the problem. In fact the gradient field becomes unstable if this is done: noise plus generalisation can cause some evaluations to go over 1, and this in turn can trigger a feedback process that causes the evaluations to explode towards big values. This has been observed in several experiments (e.g. cf. experiments in s. 11.4.4).

Further experiments are necessary to support the results presented in this section. In particular the results shown suggest that it would be necessary:

- To eliminate the problem of the evaluations converging to the optimal values. This could be done by updating the evaluations on the basis of the probabilities expressed by the actor, or by running experiments with Q learning instead of actor-critic methods.
- To eliminate the advantage of the action probabilities of states close to the goal, caused by the fact that the evaluations progressively diffuse from the goal towards the start. This could be done by using a trolley that moves from the goal toward the start.

These experiments with tighter controls will be done in the future. However, the results shown here already suggest that a special attention should be paid when discounted reinforcement learning is used for problems that involve long time periods (cf. also the results of s. 11.4.4).

6.6 Conclusion

The chapter has presented a neural implementation of the actor-critic controller. This is at the core of all the controllers presented in the next chapters. The novelty of the controller with regards to previous reinforcement-learning based neural controllers is the use of a special

neural network, the "matcher", to generate the reward signal when the goal state is encountered. In the next chapters we shall see how this network, though simple from a computational perspective, is rather important from a theoretical point of view since it allows the planning controllers to be "taskable" (cf. section 5.1.2 and 8.4.1).

The chapter has also investigated the functioning of the evaluator and the actor, collecting important data useful for interpreting the results presented in the following chapters. The most important aspect investigated has been the generalisation property of the controller that originates from its neural structure. Some simulations have shown how this property allows the controller to learn quickly and to be robust in the presence of noise. The simulations have also shown that in the context of reinforcement learning the well-known aliasing problem is particularly impairing when it involves states that are similar to the goal states, because this generates local maxima in the evaluation gradient field. A simple pre-processor, that maps the input into contrasts, has been adopted that both allows the controller to limit the negative effects of the aliasing problem, and is computationally very simple.

Finally, the chapter has shown some experiments and presented some arguments that suggest that discounted reinforcement learning may have some intrinsic limits when dealing with long periods of time, at least in the way it is usually implemented (cf. Baldassarre, 2001f, and Linaker, 2001, for an investigation possible solutions to this problem). This issue is very important for planning because planning expresses its full power, especially when compared to reactive behaviour, for tasks that last for many steps. These arguments will be further analysed and supported in chapter 11, within the study of abstract planning.

7 Reinforcement Learning, Multiple Goals, Modularity

7.1 Introduction

Problems Tackled. S. 4.4.1 has illustrated two characteristics of neural networks: the utility of their generalisation property and their capacity to isolate structure common to different problems. The same section has also suggested that these properties have a cost, namely "interference": when the controller is trained on a single task for a long time, the information gathered concerning it disrupts the information previously gathered about other tasks.

An ideal neural controller should be capable of handling multi-goals tasks, of exploiting "structure" common to them (s. 4.4.1), and of avoiding interference at the same time. This chapter investigates if and how modularity can be used to this purpose. "Modularity" means that the architecture of the controllers studied is made up of clusters of neural units that have many connections within them, and relatively fewer connections with other units (cf. Calabretta et al., 1998). In particular the chapter investigates whether it is possible to design modular neural controllers that are capable of using the same modules for areas of the input-output space that share common structure, and are capable of using different modules in other cases, in order to avoid interference.

The issue of interference is very important for planning. In fact one of the strengths of planning is its capacity to use behaviours as building blocks to pursue different goals (cf. s. 5.1.2). In this research the "building-block" behaviours considered are the "primitive" actions (as "go north", "go east" etc.). However the capacity of the controllers to learn several different behaviours (directed to achieve different goals) is the starting point to scale up to forms of planning that use more complex behaviours as building blocks (chapter 11 investigates a first simple case of neural planning where the building-block behaviours are more complex than the primitive-actions).

Overview: A New Task and a New Controller. The previous chapter used a landmark-navigation task with a single goal. This chapter introduces an instance of "asynchronous multi-goal tasks" (cf. also s. 3.1, 5.1 and 6.2). A task of this type requires that the controller pursue different goals at different times. This kind of task is introduced because it exacerbates the problems of interference, and so it can be used to test if and how much a controller is affected by such problem.

After this task has been defined, a modular actor-critic controller is proposed. This architecture uses different kinds of modular networks for the evaluator and the actor. The evaluator uses a "mixture of experts network". The mixture of experts network could not be used in a straightforward way to implement the actor, so a novel two-level hierarchical architecture has been used for it. The networks of these two levels are trained with the same algorithm used to train the actor illustrated in the previous chapter.

What is New and Related Work. Asynchronous multi-goal tasks differ from synchronous multi-goal tasks. These are tasks where a controller has to pursue several tasks in parallel by assigning proper weights to them. Synchronous multi-goal tasks have been studied under the

name of “action selection problems”. See Humphrys (1996) and Tyrrel (1993), for a brief review of these problems and some methods proposed to solve them.

The functioning of the controller presented here, as the controller of the previous chapter, is based on the actor-critic model (Barto et al. 1983; cf. s. 13.2.6). As mentioned, the modular evaluator implemented here is an application of the “mixture of experts network” (Jacobs at al., 1991; cf. s. 13.3.2). The author is not aware of previous applications of this network to reinforcement learning problems (Baldassarre, 2001b). The capacity of the mixture of experts network to reduce interference was one of the reasons for its introduction (Jacobs et al., 1991). The modular hierarchical architecture and functioning of the actor used here is novel (Baldassarre, 2001e). See also Baldassarre (2000) on the problem of interference that arises when multiple goals are pursued with monolithic neural networks.

The idea that “global” function approximators, such as the feed-forward networks trained with the error backpropagation algorithm used here, are badly affected by interference and are not suitable for reinforcement learning, has already been investigated elsewhere (e.g. Sutton and Whitehead, 1993; Samejima and Omori, 1999). These and other works also suggest that “local” function approximators, such as the mixture of experts network used here, are less affected by interference.

Caruana (1995) has presented research that uses feed-forward networks trained with the error back-propagation algorithm to learn many different tasks. He has shown that such neural networks are capable of transferring skills between tasks sharing common structure.

Calabretta et al. (1998) have presented an interesting modular neural architecture that controls a robot that solves a complex compound task. This work uses genetic algorithms to train the weights, so it is not directly comparable with our results. Notwithstanding this, the work is relevant for this research since it shows that a modular controller can solve some tasks that a monolithic controller cannot.

Chapter's Outline. S. 7.2 introduces an asynchronous multi-goal task based on the navigation scenarios introduced in chapter 6. S. 7.3 presents one controller with a “monolithic” neural-network architecture, and a second controller with a modular neural-network architecture. The simulations presented in s. 7.4 show that the monolithic controller is very slow in learning because it is affected by interference, while the modular controller has a relatively good performance. This section also presents some data about how the modular controller's performance is based on emergent functional modularity. Finally s. 7.5 illustrates the limits of the controllers, and s. 7.6 draws the conclusions.

7.2 Scenario of Simulations: An Asynchronous Multi-Goal Task

The scenario used in this chapter is shown in Figure 7.1. This is the same “complex” scenario shown in the previous chapter (cf. s. 6.2 and Figure 6.2). The only difference is the position of the goals. The robot used in the simulations has the same properties of the one used in the previous chapter (cf. s. 6.2).

The simulated robot's task is to reach three different goal positions in the arena. At the beginning of the simulation the simulated robot is set at the start position (cf. Figure 7.1) and has to reach the east goal. Then each time the simulated robot reaches a goal, another goal randomly drawn from the three goals is assigned to it until the simulation stops. Since the simulated robot’s step size is 0.05, the arena’s size is 1 by 1, and the distance between each goal is about 15 steps, the average optimal straight path between two goals is about 10 steps long (i.e. (15+15+0)/3) ignoring action noise and problems with the obstacles.

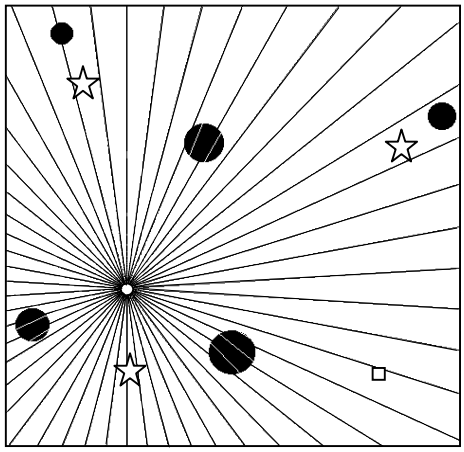


Figure 7.1: The scenario used in the simulations. The scenario contains three goals (north-west, east, and south-west, marked with a star), the start (white square), five landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), and the robot (white circle at origin of rays).

7.3 Architectures and Algorithms: Monolithic and Modular Neural-Networks

This section illustrates the details of the architecture of the two controllers used in this chapter. The first controller has an evaluator and an actor based on monolithic neural networks, while the second has an evaluator and an actor based on modular neural networks.

Monolithic Neural-Network Controller. The architecture of the first controller is shown in Figure 7.2. The general structure of the architecture is the same as the architecture of the controller presented in chapter 6 (cf. Figure 6.3). The only difference with that architecture is that now the evaluator is a *three-layer* feed-forward network instead of a two-layer feed-forward network, and the actor is composed of eight (one per action) *three-layer* feed-forward networks instead of eight two-layer feed-forward networks.

The functioning of the whole controller is the same as the one of chapter 6, with the evaluator that learns to evaluate the states and the actor that learns the "merits" (pseudo-probabilities) of the actions. The only difference is that the error backpropagation algorithm is used to train the three-layer networks used here instead of the Widrow-Hoff rule, that is applicable to two-layer networks only (cf. s. 13.3.1).

Modular Neural-Network Controller. The architecture of the modular controller is presented in Figure 7.3. This graph, together with other similar graphs used in the following chapters, uses boxes to indicate networks without depicting the internal details of them. This has been done because the compound architectures presented from now on have several components so that they would have produced overly complicated graphs if all the details had been reported.

The main architecture and functioning of the modular controller is based on the controller presented in chapter 6. The differences are in the architectures of the evaluator and actor that are now modular. The actor is a modular network composed of 6 "expert networks" and 1 "gating network". Each expert is a *two-layer* feed-forward neural network that gets the goal

and the visual contrasts as input, and has 8 sigmoidal output units that locally encode the actions. As in the actor of chapter 6 (cf. Eq. 6.1), the activation m_q ("action merit") of the output units is sent to a stochastic selector where a stochastic "winner-take-all competition" takes place to select one action. The probability P[.] that a given action a_q becomes the winning action a_{win} is given by:

$$P[a_q = a_{win}] = m_q \; / \; \sum_f m_f \qquad \text{Eq. 7.1}$$

The role of the gating network is to select an expert that, in its turn, selects the actions to be executed in the way just shown. The gating network has the same input and architecture as the experts, but it has only six sigmoidal output units, each corresponding to one expert, instead of eight. The gating network functions in the same way as the experts do, with the only difference that its output units and their activation (merits) refer to the six experts instead of the eight actions. Similarly to the experts, the gating network uses a stochastic "winner-take-all competition" to select the "winning" expert.

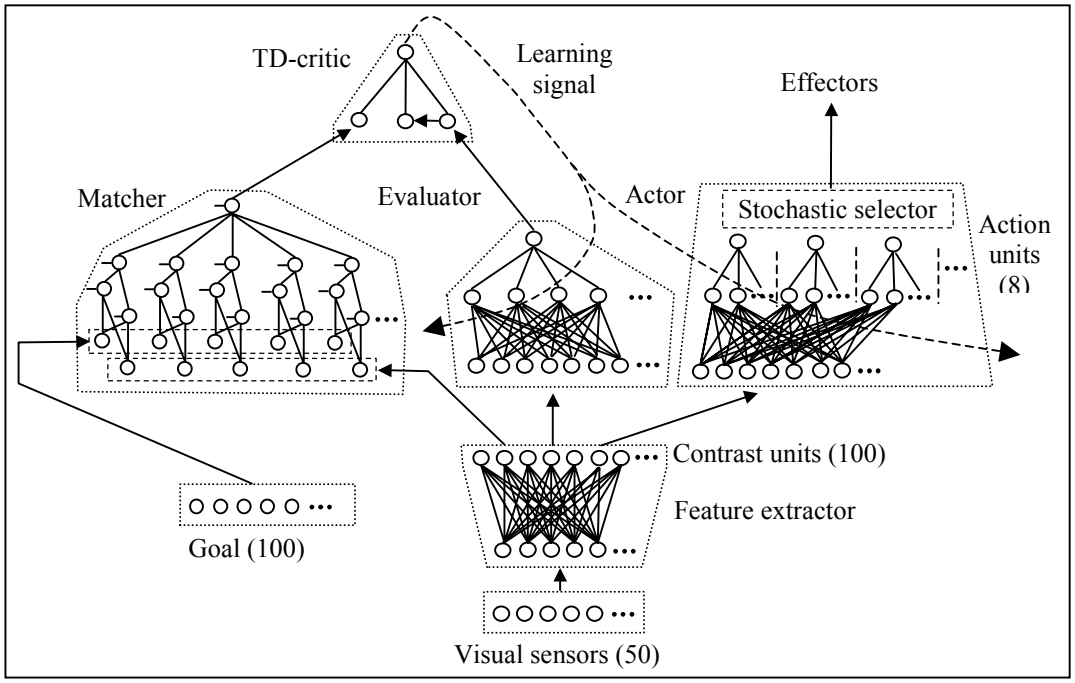


Figure 7.2: Components of the monolithic controller. Arrows indicate that a pattern is "copied" from one unit/layer to another unit/layer. Dotted arrows indicate the learning signal. For each layer only few units have been drawn. In the case of the actor only the networks relative to three actions out of eight have been drawn. The total number of units of each layer is indicated in round brackets, except for the hidden units whose number varied in different simulations.

The evaluator is a "mixture of experts neural network" composed of 6 experts and 1 gating network. S. 13.3.2 discusses further details of this architecture, and presents the mathematical justification of the training algorithm described below in intuitive terms. Each expert is a two-layer feed-forward neural network that gets the goal and the visual contrasts as input. With its linear output unit, the evaluator yields the estimate $V'^{\pi}[\mathbf{y}_t]$ of the evaluation $V^{\pi}[\mathbf{y}_t]$ of the current contrast pattern $\mathbf{y}_t$, defined in the usual way on the basis of the future rewards r (cf. s. 13.2.4):

$$V^{\pi}[\mathbf{y}_t] = E[\gamma^0 r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \ldots] \quad \text{Eq. 7.2}$$

where $\gamma \in (0, 1)$ is the discount factor, set at 0.95 in the simulations, and E[.] is the mean operator. The output of the experts is weighted and summed in order to compute $V'^{\pi}[\mathbf{y}_t]$:

$$V'^{\pi}[\mathbf{y}_t] = \Sigma_k[v_k\, g_k] \quad \text{Eq. 7.3}$$

where v_k is the output of the expert k, and the weight g_k is computed as the "softmax activation function" of the output units' activation o_k of the gating network:

$$g_k = \exp[o_k]/\Sigma_f[\exp[o_f]] \quad \text{Eq. 7.4}$$

Notice that $\Sigma_k\, g_k = 1$.

As usual the TD-critic is a neural implementation of the computation of the "temporal-difference error" e_t ("learning signal" in Figure 7.3) defined as:

$$e_t = (r_{t+1} + \gamma\, V'^{\pi}[\mathbf{y}_{t+1}]) - V'^{\pi}[\mathbf{y}_t] \quad \text{Eq. 7.5}$$

Now it is also possible to compute the *specific* temporal-difference error e_{kt} for each expert:

$$e_{kt} = (r_{t+1} + \gamma\, V'^{\pi}[\mathbf{y}_{t+1}]) - v_k[\mathbf{y}_t] \quad \text{Eq. 7.6}$$

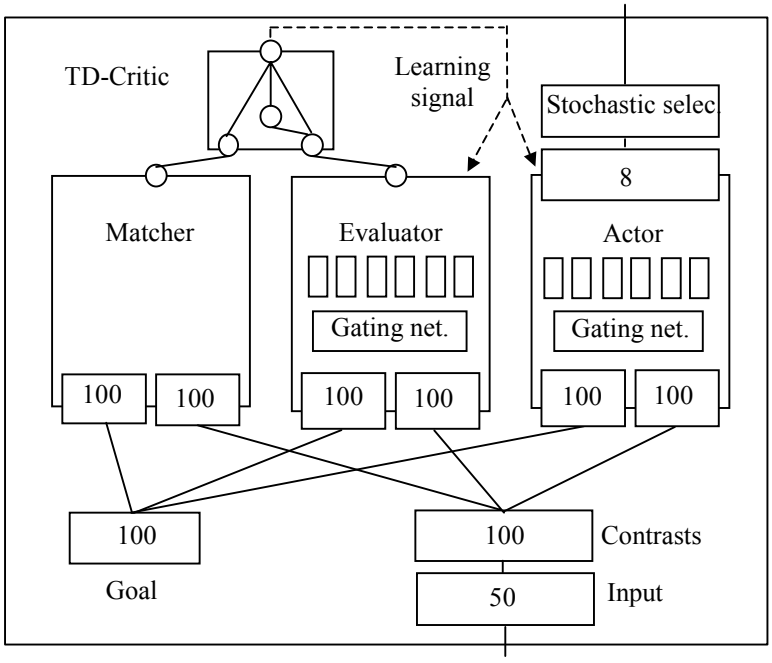


Figure 7.3: The modular controller. Arcs indicate forward connections that "copy" a pattern from one layer to another. Dashed arrays indicate the learning signal used to update the weights of the evaluator and the actor.

Each evaluator's expert is trained on the basis of its temporal-difference error. This assumes the role of error in a supervised learning algorithm. In particular the weights of the experts are updated so that the estimate $v_k[\mathbf{y}_t]$ yielded by each of them tends to be closer to the

target value ($r_{t+1}+ \gamma V'^{\pi}[\mathbf{y}_{t+1}]$). This target is a more precise estimate of $V^{\pi}[\mathbf{y}_t]$ because it is expressed at time t+1 on the basis of the observed r_{t+1} and the new estimate $V'^{\pi}[\mathbf{y}_{t+1}]$. The formula (a modified Widrow-Hoff rule, cf. Widrow and Hoff, 1960, and s. 13.3.1) used to update the weights of each expert is:

$$\Delta w_{kj} = \eta \, e_{kt} \, y_j \, h_k \qquad \text{Eq. 7.7}$$

where η is a learning rate, w_{kj} is a weight of the expert k, and y_j is the activation of the evaluator's input units at time t. h_k (absent in the Widrow-Hoff rule) is the "updated" contribution of the expert to the global answer $V'^{\pi}[\mathbf{y}_t]$, and is defined as (cf. s. 13.3.2):

$$h_k = g_k \, c_k \, / \, \Sigma_f[g_f \, c_f] \qquad \text{Eq. 7.8}$$

where c_k is defined as:

$$c_k = \exp[-0.5 \, e_{kt}^2] \qquad \text{Eq. 7.9}$$

and intuitively can be interpreted as a measure of the "correctness" of the expert k. Cf. s. 13.3.2 for a more rigorous (but less intuitive) interpretation of this measure. Notice that $\Sigma_k \, h_k = 1$.

The weights z_{kj} of the evaluator's gating network are updated to increase the contributions g_k (to the production of the evaluation) of the experts that has produced a low error:

$$\Delta z_{kj} = \xi \, (h_k - g_k) \, y_j \qquad \text{Eq. 7.10}$$

where ξ is a learning rate.

The actor's experts are trained according to the TD-critic's learning signal e_t. The updating of the actions' merit of the selected expert (and only this) is done by updating the weights of the neural unit corresponding to the selected action a_{win} (and only this) as follows:

$$\Delta w_{win\,j} = \zeta \, e_t \, (4 \, m_{win} \, (1 - m_{win})) \, y_j \qquad \text{Eq. 7.11}$$

where ζ is a learning rate, and $(4 \, m_{win} \, (1 - m_{win}))$ is the derivative of the (sigmoidal) transfer function multiplied by 4 to homogenise the size of the learning rates of the actor and the (linear) evaluator (in fact the maximum of that derivative of the transfer function is 0.25 for the actor, and 1 for the evaluator).

The weights of the actor's gating network corresponding to the winning expert are updated through Eq. 7.11, where the winning expert is considered instead of the winning action. At the beginning of the simulation the weights of the evaluator and actor are randomised in the interval [-0.001, +0.001].

7.4 Results and Interpretation

As mentioned, the task of the simulated robot was to reach one of the three goal positions shown in Figure 7.1. When a goal was reached a new one (randomly chosen between the three goals) was assigned to the simulated robot and the simulated robot had to reach it from its current position. Two groups of simulations have been run by using this scenario. The first group of simulations has used the controller with the monolithic architecture, and the second group has used the controller with the modular architecture. The performance has been

measured in terms of number of steps taken to achieve a goal, averaged over the last 100 goals reached (at the beginning, with less than 100 goals reached, a simple average has been used).

A parameter search has been done in order to optimise the number of hidden units of the monolithic controller. The results have shown that with 3 hidden units the controller is not capable of solving the task. With 5 and 10 hidden units the controller is capable of solving the task and shows an equivalent performance in the two cases. The results shown below refer to the case with 10 hidden units.

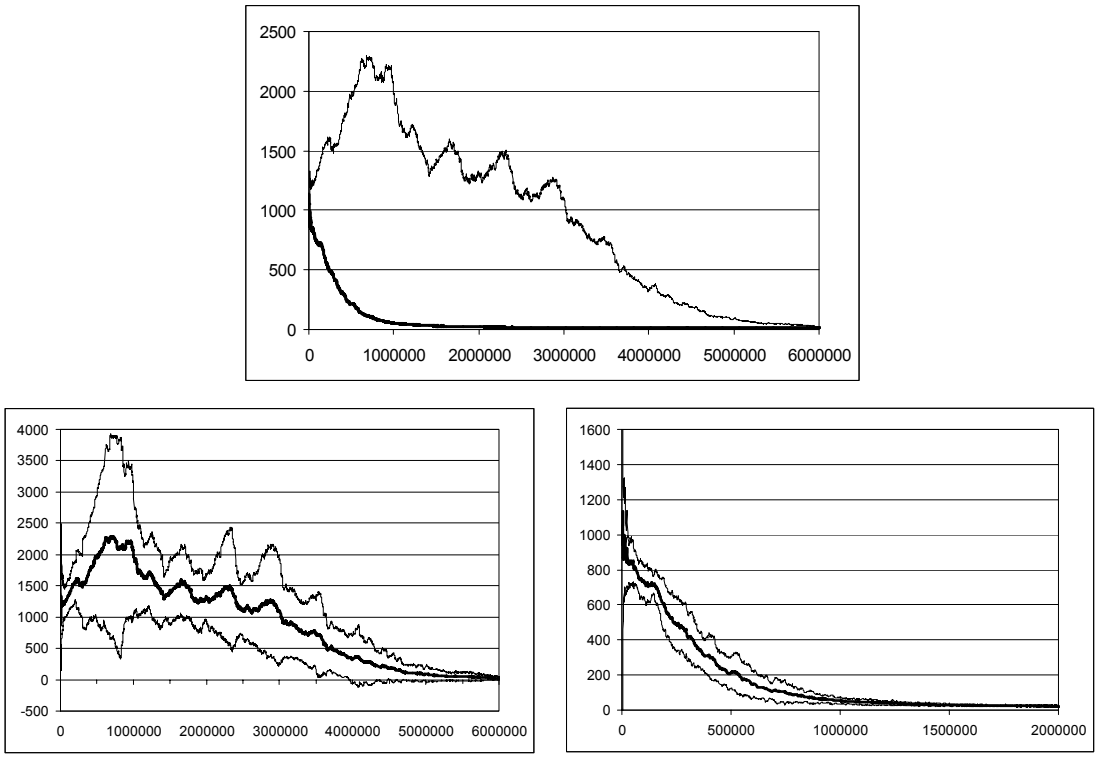


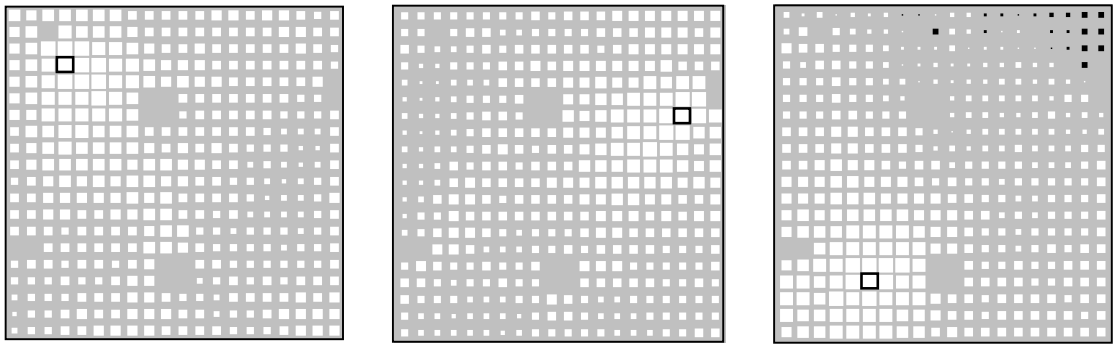
Figure 7.4: Top: the performance of the two controllers (y-axis) measured as number of steps per goal, averaged over the last 100 successes, and averaged over 10 runs of the two simulations repeated with different random seeds. The performance has been plotted against the cycles (x-axis). The thin curve refers to the "monolithic" controller, while the bold curve refers to the modular controller. Bottom: graphs showing again the learning curve of the monolithic controller and the modular controller, but also these curves plus and minus their respective standard deviations for the ten random seeds. In the case of the monolithic controller 10 hidden units and a learning rate of 0.1 have been used for the learning networks of both for evaluator and actor. In the case of the modular controller a learning rate of 0.02 has been used for all the learning networks of the controller, with the exception of the gating network for which a learning rate of 0.2 has been used.

A parameter search has also been done in order to optimise the learning rates of the evaluator and actor of the controllers. With one exception (see below) the same learning rate value has been used both for the evaluator and actor. In the case of the monolithic controller, learning rates set at 0.5 cause the evaluations to explode towards positive values. Learning rates set at 0.2 lead the simulated robot to get stuck in areas of the gradient field with local maxima for long times before the performance (measured as number of steps per goal, see below), converges to about 20 steps per goal. The data reported below refer to simulations run

with learning rates set at 0.1, for which the performance converges quite smoothly (but not yet completely, see below) to good levels.

In the case of the modular controller, the learning rates set at 0.05 lead to quick learning, with a performance of about 20 steps per goal after only 200,000 cycles of the simulation. However, the evaluations are very unstable, i.e. they continue to change at each time step, and with some random seeds the simulated robot gets stuck in local minima of the gradient field. For this reason the data reported below refer to simulations run with learning rates set at 0.02, for which the performance converges smoothly and without problems to good levels. Unfortunately, and this is a drawback of the modular controller, to obtain this result the learning rate of the gating network of the evaluator has been set at a value different from the other learning rates, namely 0.2. In fact, without this learning rate the specialisation of the evaluator (see below) failed for some random seeds, in the sense that the evaluator used one expert for two goals and the performance of the controller was disrupted.

Figure 7.4 reports the performance of the two controllers with the settings just described. The performance is measured in terms of number of steps taken to achieve a goal, averaged over the last 100 successes, and plotted against the cumulated cycles of the simulation. The graph shows the average for 10 repetitions, with different random seeds, of the two simulations. It can be seen that in the case of the monolithic controller the performance improves from about 1,000 to about 20 after 5,000,000 steps. In the case of the modular controller the performance improves from about 1,000 to about 20 after 1,000,000 steps. Recall that the optimal performance, ignoring noise and obstacles, would be about 10 steps, so the performance can be considered satisfactory. Moreover, the variance of the monolithic controller is quite large since the controller sometimes still got stuck in areas with a local maximum of the evaluation gradient field with the learning rates used, 0.1. With even lower learning rates, 0.05, the variance was smaller and the curve was smoother, but the performance converged after about 9,000,000 cycles.



Evaluator's expert: 2 Evaluator's expert: 5 Evaluator's expert: 6

Figure 7.5: Data about the emergent functional modularity of the evaluator of the modular controller after the performance converged (1 random seed out of 10). Each graph, one per goal, reports the gradient field of the evaluations yielded by the evaluator in 400 different positions of the arena. Each position corresponds to one of the 20×20 cells of the graph. For each graph, the cell with a bold border indicates the position of the goal pursued. Under each graph the evaluator's expert that is responsible for the evaluations corresponding to the particular goal is indicated.

What causes this difference in the performance of the two controllers? The cause is likely to be the interference of learning to pursue the different goals. This affects the monolithic controller much more than it affects the modular controller. In fact, after the experts of the

modular controller start to specialise in different areas of the input-goal-output space, pursuing one goal has little disruptive effects on the skill (weights) learned for the other goals.

Figure 7.5 presents some data about the "emergent functional modularity"(cf. Calabretta et al., 1998), i.e. the specialisation of the experts of the evaluator, of one of the 10 runs with the modular controller. The other random seeds have produced results with analogous quality. The graph shows the evaluation gradient field for the three goals when the performance has converged. The evaluator deals with each goal by using a different expert. In particular for each goal and in each possible position of the arena, the "weight" of one particular expert in determining the evaluation (cf. Eq. 7.4) is over 0.99. This probably means that different positions in the arena need to be evaluated in a different way for the three goals, so that the algorithm uses a different expert for each goal to avoid interference. This also means that the connections from the (contrast) input pattern to the evaluator's gating network are redundant: the information about the goal to pursue is sufficient to select an expert. Notice that the controller is capable of *not* using some of the resources available (experts 1, 3, 4). These resources could be used for other goals.

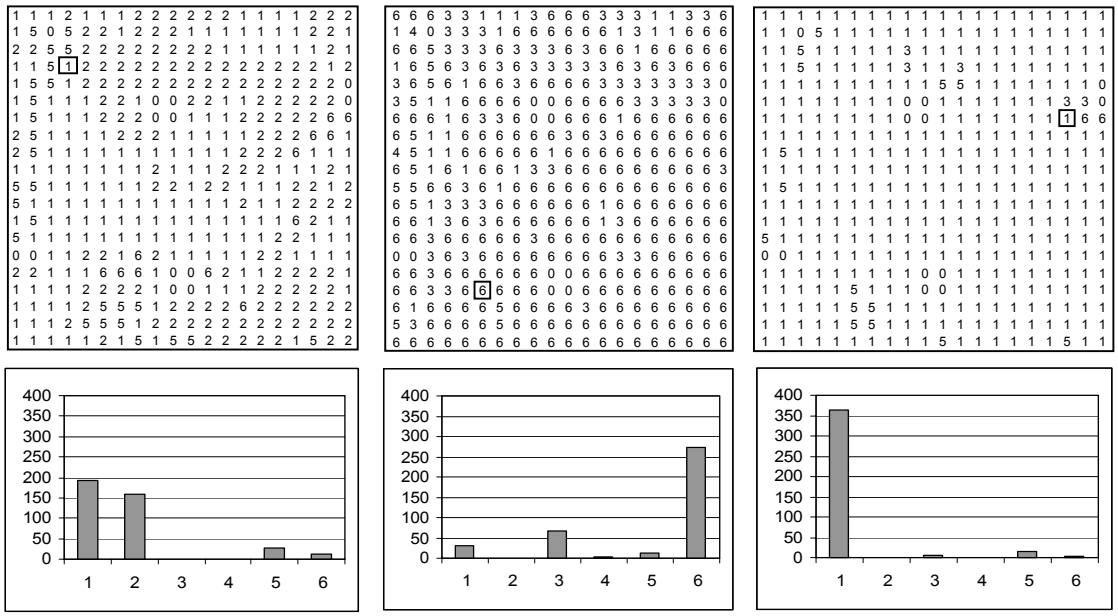


Figure 7.6: Data about the emergent functional modularity of the actor of the modular controller (1 random seed out of 10). The first row of graphs reports the ordering number of the actor's expert with the highest probability of being selected, for each of 20×20 positions on the arena. The grid of numbers can be overlapped to the arena to find the corresponding positions. A number 0 indicates a position occupied by a landmark. The second row of graphs reports the histograms that summarise the frequencies of use of the experts illustrated in the first row of graphs.

With regards to the actor, Figure 7.6 shows that the specialisation of the experts is much less pronounced. In particular the graphs in the first row show that while pursuing a particular goal the actor uses different experts in different position of the arena. The histograms of the second row summarise the frequency of use of the different experts for the different goals. Clearly the actor tends to use different experts when dealing with different goals. However now, unlike what happened for the evaluator, the visual input plays an important role and the actor uses different experts in different positions of the arena. Moreover (e.g. see expert 1 for goal 1 and 3) the same experts are used for different goals. Notice that in the actor, as in the

case of the evaluator, there is a partial use of the resources available (marginal use of experts 3, 4, and 5).

Further investigation should verify if the different use of the experts of the evaluator and actor are caused by the differences in the role they play (learning of evaluations and learning of the policy) or if it is caused by the difference between the architectures and algorithms employed for them.

7.5 Limitations of the Controllers

The chapter has shown that the monolithic controller learns very slowly when dealing with an asynchronous multi-goal task. A modular controller has shown a better performance. Unluckily, the modular controller has limitations, too. As mentioned it needs a fine-tuning of the leaning rates of the evaluator's gating network, otherwise the specialisation of the experts can fail, in the sense that the same expert is used for more than one goal, and this can reduce the performance of the controller. These problems are probably caused by the interference between the experts, in turn caused by the gating network of the mixture of experts network used here. This is based on a soft-max function. See Ramamurti and Ghosh (1997) on this problem, and on the proposal of a gating network, based on local function approximation, that is not affected by these problems.

Another limitation is the “strong” functional modularity of the evaluator that does not allow the controller to easily scale up to many goals. In fact the number of experts available to the evaluator fixes the maximum number of goals that can be pursued. If a further goal is added over this limit, the performance abruptly deteriorates, while a “graceful degradation” would be desirable.

7.6 Conclusion

This chapter has introduced a task where a simulated robot has to pursue several goals at different times. These kinds of tasks are relevant if one wants to test if a neural controller is scalable to more complex scenarios and is capable of exploiting the full potentiality of neural networks in terms of their capacity to discover common structure underlying different goals/problems and to avoid interference. Some experiments have shown how monolithic networks with hidden units, even if potentially capable of discovering common structure, are affected by problems of interference that slow learning. A controller based on a modular architecture has been designed and implemented to overcome this problem. The simulations have shown that this controller is capable of limiting the effects of interference by exploiting emergent functional modularity.

This chapter concludes with a comment on the performance of the mixture of experts network of the evaluator. The simulations suggest that this architecture may be quite rigid in its capacity to discover underlying structure and avoid interference. In fact it either uses different experts for the different goals, or it uses the same expert for more than one goal. Perhaps, in the task considered using different experts for different goals is the correct thing to do. Alternatively it is possible that the mixture of experts network is actually rigid, and not capable of discovering underlying structure between different goals as it should do. For example, the behaviour of the architecture and algorithms of the actor that generate a fuzzy specialisation of the experts might be considered an indication of higher flexibility. These issues need to be further investigated.

8 The Neural Forward Planner

8.1 Introduction: Taskability, Planning and Acting, Focussing

Problems Tackled. This chapter deals with the problems of taskability of the Dyna-PI architecture, with the problem of focussing planning around relevant states, and with the problem of interleaving acting, planning and re-planning.

We have seen in s. 5.1.4 that the Dyna-PI architecture is not taskable in a strong sense. In s. 5.1.3 a double test has been given to decide operationally if a system is taskable. A system is taskable if: (a) it works on the basis of the *goal* information only; (b) the *first time* that it reaches the goal it can reach it with an efficiency superior to the efficiency of the corresponding reactive system (if there is one) or the random solution. Here a new planning controller will be presented: the "neural forward planner" (or simply "forward planner"). This controller is inspired by the Dyna-PI architecture, but contrary to this, it passes the two tests of taskability.

S. 5.3 has explained why it is better to use "partial policies" instead of "full policies" defined for all states. S. 3.4 has argued that "trajectory sampling" is a way to focus planning on relevant states. By proposing the forward planner, this chapter aims at specifying and implementing these ideas.

S. 2.4.4 has suggested that the best strategy between the two extremes of universal planning (full policy) and pure re-planning is to have a partial policy that contemplates what to do in the situations most likely to occur, and to do re-planning when things are too different from expectations. The neural planner proposed here specifies and implements this strategy. In fact on one hand it prepares a partial policy focussed on the states around the current state, the goal state, and the states between these two, and on the other hand it triggers re-planning when the simulated robot encounters "unexpected" states during action execution.

Overview of the Controller. The planning controller presented in this chapter is built by adding some new components and algorithms to the controller presented in chapter 6, that is based on the actor-critic methods. The idea exploited here is the one at the basis of the Dyna-PI architecture (cf. s. 3.3): the evaluator and actor are trained through experience generated through the model of the environment instead of real experience. The components added to the basic actor-critic model to obtain the planning model are the following ones:

- Predictor: a neural network that implements the model of the environment.
- Matcher: a neural network that is capable of deciding if the goal has been achieved or not (either in the environment or in the simulated experience) and to generate a reward signal accordingly.
- Action-planning controller: an algorithm that determines when to act and when to plan, administers the flow of information between the different neural components that make up the whole system, and directs the generation of simulated experience during planning.

Incidentally, notice that a fourth important component, that is not present in the model, would be needed to have a fully autonomous robot. This component would have the function of memorising the goals and of recalling them at the appropriate moment. For example if the

robot were engaged in a navigation task this component could allow the robot to memorise "snapshot images" of relevant locations in space, e.g. locations where the robot has found some resources important for its activity. At a later stage, when these resources are needed, the robot should be capable of retrieving the image of the location where the resources are, so that this position would become the goal position that it would try to reach. Notice that this is a quite sophisticated function, difficult to implement.

When planning, the controller simulates experiences based on "simulated walks" of the kind: "sensory input → action → prediction of new sensory input → action…". The planning process is a form of "forward planning". In fact each simulated walk starts with the sensory input corresponding to the current state and continues with the predicted input patterns generated in a sequence. Each simulated walk terminates either when the goal is achieved (within the simulated experience), or when the length of the simulated walk becomes longer than a certain maximum length. This prevents the planning process from getting stuck in unproductive loops and dead ends, and also focuses the search around the current state. As we shall see, this maximum length is set at 1 when the planning process starts, and then is increased by 1 unit each time that the simulated walk fails to reach the goal. This guarantees that the search is progressively extended to states more distant from the current state.

It is important to briefly discuss the fact that the predictor's training has been accomplished before the tests (incidentally, this is also done by the majority of works that have implemented neural planning on the basis of the activation diffusion principle, cf. s. 4.5.1, and the planning systems based on gradient descent methods, cf. s. 4.5.2). This choice has been made for two reasons. The first is that in this way it has been possible to show that the forward planner is capable of implementing taskable planning. In fact it can reach any goal assigned to it on the basis of the information contained in the predictor, without the need to train again the model of the environment. The second reason is that model updating carried out while acting would introduce other complex problems out of the scope of this research.

What is New and Related Work. The previous section has already highlighted some aspects of the controller that are new (Baldassarre, 2001c). As mentioned the reactive components of the controller presented in this chapter, are largely based on the actor-critic model implemented with neural networks (Barto and Sutton, 1998) and analysed in chapter 6. The idea of implementing planning as a form of learning within a model of the environment is from Sutton (1990, "Dyna-PI" models, cf. s. 3.3; cf. also Barto et al., 1995, on trial-based real-time asynchronous dynamic programming applied to path finding problems, cf. s. 13.2.11). It is important to stress that previous work using Dyna-PI architectures (e.g. Sutton 1990; Lin, 1992) has used it as a way *to speed up learning*, not to implement genuine taskable planning. The reason was that a device like the matcher was needed to implement planning. The idea of the matcher is new, but it has been inspired by the idea of "goal test" used in problem solving and planning (cf. s. 2.1 and 2.3). The idea of generating simulated experiences on the basis of the current policy, called "trajectory sampling", was investigated by Barto et al. (1995) and Sutton and Barto (1998, p. 247; cf. s. 3.4 for a review). The idea of increasing the depth of the path generated during planning resembles an iterative deepening search (Korf, 1985a, cf. s. 13.1.1), but it is new because it has been developed for the application to problems with stochastic actions' effects. The planning algorithm controlling the flow of information between the components of the model, and the use of the predictor to generate "simulated walks", are new. The idea of implementing the predictor (model of the environment) with a feed-forward neural network trained with experience has already been applied by Lin (1992) and Nolfi and Tani (1999) (see other examples in s. 4.5.2). Notice that all these works use deterministic neural networks to implement a model of an environment

that is actually stochastic. This simplification is also used here (see below). An alternative approach would have been to use stochastic networks, such as the feed-forward stochastic networks proposed by Neal (1995; 1996). This idea has not been tested here. As we shall see, the predictor is trained while the simulated robot navigates randomly in the environment. A random navigation has been used to mimic the way an unsophisticated autonomous robot would navigate in the absence of any previous knowledge. More sophisticated ways of exploring the environment to improve model building have been proposed (e.g. cf. Duckett and Nehmzow, 1999; Schmidhuber, 1999). These are not tested here. The idea of using "expert" networks for the predictor, each specialised to predict the consequences of a specific action, has been used in Lin and Mitchell (1992).

Chapter's Outline. S. 8.2 presents the task used to test the controller. S. 8.3 presents the architecture of the system and in particular the planning components and the algorithm that manages planning and decides when to plan and to act. S. 8.4.1 shows that the neural planner presented here is taskable. S. 8.4.2 shows how information gained with simulated and real experience merges nicely within the policy. S. 8.4.3 analyses the details of how the predictor works. Finally s. 8.5 highlights the drawback of the controller and s. 8.6 draws the conclusions.

8.2 Scenario of the Simulations

The simulated scenario and robot used to test the controller presented later are the ones illustrated in s. 6.2. For convenience, Figure 8.1 reports the scenario and the goals that the simulated robot has to pursue.

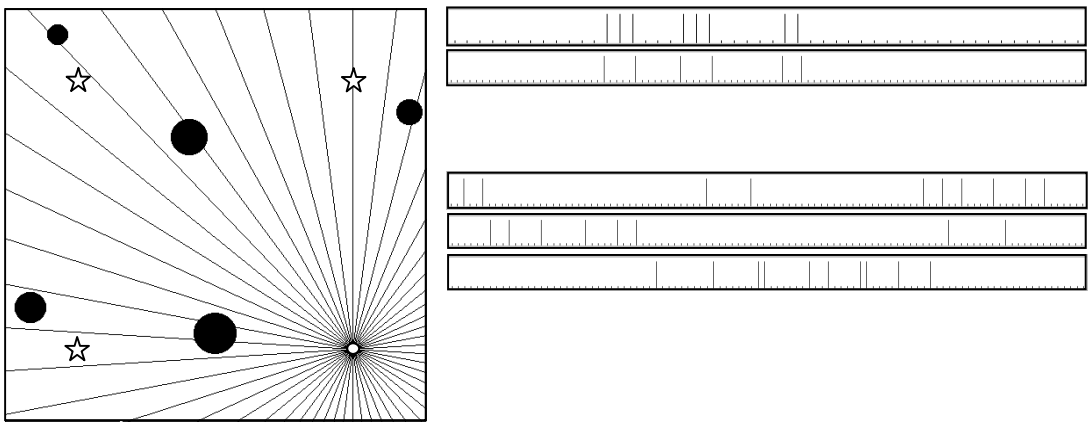


Figure 8.1: Left: the simulations' scenario containing the three goals (stars), five landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), the simulated robot at the start position (white circle). Right, in order from the top: the activation of the simulated robot's sensors at the start, the corresponding contrasts, the three goals (as contrasts relative to the images viewed from the goal positions).

8.3 Architectures and Algorithms: Reactive and Planning Components

This section will first analyse the differences between the reactive components of the controller presented here, and those of the controllers presented in the previous chapters. Then it will analyse the components added to them to obtain the neural forward planner.

8.3.1 The Reactive Components of the Architecture

Figure 8.2 shows both the reinforcement learning and the planning components of the simulated robot's neural controller. A description of the reinforcement-learning components is now given. This part of the model is the same as the one reported in s. 6.3, and in particular in Figure 6.3 (Figure 8.2 represents each neural network as a box that does not shows the internal details about the single units and connections as Figure 8.2 does).

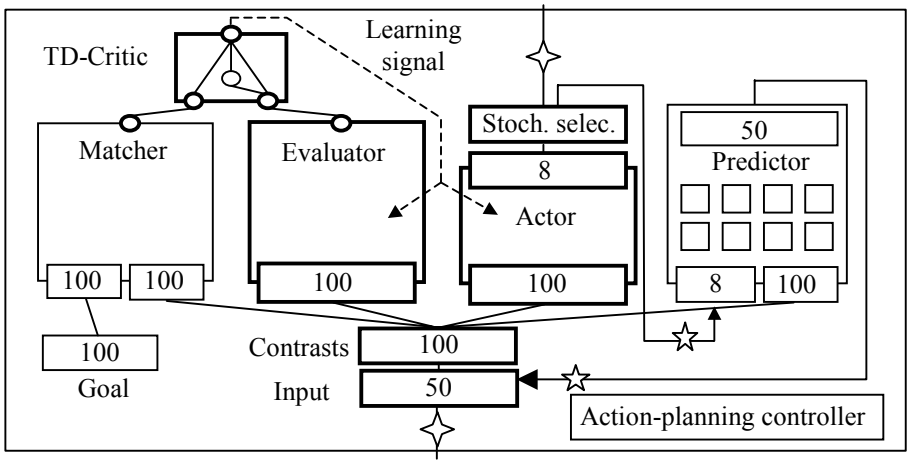


Figure 8.2: The controller of the simulated robot. Networks with a bold and thin border implement reinforcement learning and planning respectively. Arcs and arrows respectively indicate forward and backward connections that "copy" a pattern from one layer to another. The four and five spike stars respectively indicate the channels set open or close by the action-planning controller when acting (vice versa when planning). Dashed arrays indicate the learning signal used to update the weights of the evaluator and actor.

As previously, the actor selects the actions in a stochastic fashion, and the evaluator evaluates the states of the world in terms of expected future rewards, on the basis of the current actor's action-selection policy. The evaluator improves the quality of the evaluations by experiencing the rewards through a supervised learning algorithm. The actor improves the action-selection policy by increasing the probabilities of those actions that bring it to states with an evaluation higher than the one expected by the critic. So, as previously, the controller is perfectly capable of learning by a trial-and-error process. However now, as we shall see below, when the controller is planning the same evaluator and actor are also trained through pseudo-experience generated by using the "predictor". When this happens, the evaluator and actor function in the same way as they do when they are trained through real experience. They treat the two kinds of experience in identical way. If the model of the world is accurate enough, the effect of training with simulated experience while planning is that the performance of the controller in the world improves.

8.3.2 The Planning Components of the Architecture

Now the components added to the reactive-learning model to obtain the planning controller are explained. The "predictor", i.e. the controller's model of the environment, is a set of 8 feed-forward two-layer networks ("experts") with sigmoidal output units, each corresponding to one action. Each expert takes $\mathbf{y}_t$ as input, and is specialised to predict the following sensors' activation $\mathbf{x}_{t+1}$ if the action corresponding to it is executed. To this purpose the output of each sigmoidal unit of the expert selected is set at 0 if below 0.5, and at 1 if above, in order to obtain a binary pattern. A hand-designed algorithm chooses the expert corresponding to the selected action to yield the output of the predictor itself. This algorithm could be easily implemented with neural networks: the activation of the selected action's unit could be used to inhibit the activation of all the units of the experts different from the expert corresponding to the selected action. However this would not produce any substantial insight.

The experts are trained while the simulated robot navigates randomly in the environment for 200,000 cycles. This training brings the quadratic error per unit to about 0.24 (the "quadratic error" is computed as the square root of the average of the squared error per unit). The training is done before the main simulations illustrated later, and then the experts are used unchanged for all the simulations. At each cycle the contrast pattern $\mathbf{y}_t$ and the input pattern $\mathbf{x}_{t+1}$ observed after the execution of one action, are respectively used as input and teaching output to train the expert corresponding to the action with a Widow-Hoff rule (Widow and Hoff, 1960; cf. s. 13.3.1). Notice that, because of its architecture, the predictor yields deterministic predictions that tend to be the average of the $\mathbf{x}_{t+1}$ observed after each $\mathbf{y}_t$. This is clearly a simplification since a correct model of the environment that is stochastic should yield stochastic predictions.

It is important to explain why one expert for each action has been used instead of one monolithic neural network with current state and current action as input and predicted next input as output. At the beginning of the research, some exploratory simulations have been run with the monolithic network (a three-layer feed-forward neural network trained with the error backpropagation algorithm) and the results have been poor. They can be summarised as follows. The behaviour of the predictor has the strong tendency to get stuck in a behavioural local minimum for which the current state of the input is repeated as output. The reason is that, with the sizes of the simulated robot's movement used to implement the actions (e.g. the "go_to_north" action), the next input is identical to the current input with the exception of few bits. These few bits are not always the same even for the same action (selected at the same state) as noise affects the consequences of it. As a result, the predictor tends to treat the bits common to the current input and the next input as the actual input-output pattern association to learn, and the differences between them, caused by the different actions selected, as noise. The use of one expert for each action greatly facilitates the training of the predictor as the different bits between each "current input" and its next input tends to be consistent in time, since the same action is always selected in correspondence to a given expert (cf. Lin and Mitchell, 1992, on the "one action one network principle" according to which in reinforcement learning it is usually advantageous to use one neural network for each action).

The controller can be either in planning or acting mode. The action-planning controller is a hand-designed algorithm whose pseudo-code is showed in Figure 8.3. The action-planning controller decides when planning and when acting and directs the flows of information between the different networks of the architecture. Here, first an overview of the functioning of the action-planning controller is given, and then a detailed explanation of it is presented. The action-planning controller decides the controller's mode on the basis of its "confidence". The confidence is defined as the highest of the actions' probabilities measured at the position

currently occupied by the simulated robot. If the confidence is above a certain threshold the controller acts in the world and the predictor is not used.

```
01 IF(NewGoalHasBeenAssigned)
02   MaxStepsPlan := 1
03   ConfThresh := MaxConfThresh
04   StepPlan := 0
05   InputFromWorld := TRUE
06 IF(InputFromWorld)
07   System gets input x_t (y_t) from the robot's sensors
08   Actor gets y_t and gives m_t
09   Confidence is computed on the basis of m_t
10   IF(Confidence < ConfThresh)
11     Planning := TRUE
12   ELSE
13     Planning := FALSE
14     ConfThresh := MIN(MaxConfThresh, ConfThresh + Gain)
15 IF(Planning)
16   StepPlan := StepPlan + 1
17   ConfThresh := ConfThresh - Decay
18   IF(InputFromWorld = FALSE)
19     System uses predictor's output y_t as input
20   InputFromWorld := FALSE
21   IF(GoalReached OR StepPlan = MaxStepsPlan)
22     IF(StepPlan = MaxStepsPlan)
23       MaxStepsPlan := MaxStepsPlan + 1
24     ELSE
25       MaxStepsPlan := MIN(MaxStepsPlan, StepPlan * 2)
26     InputFromWorld := TRUE
27     StepPlan := 0
28 Evaluator gets y_t and gives V'^π[y_t]
29 Actor gets y_t and gives m_t
30 Stochastic selector gets m_t and gives a_t
31 Matcher gets y_g, y_t and gives r_t
32 TD-Critic gets V'^π[y_t-1], V'^π[y_t], r_t and gives e_t-1
33 Evaluator gets y_t-1, e_t-1 and learns
34 Actor gets y_t-1, m_t-1, a_t-1, e_t-1 and learns
35 IF(Planning)
36   Predictor gets y_t, a_t and gives x_t+1 (y_t+1)
37 ELSE
38   System executes a_t in the environment
```

Figure 8.3: Pseudo-code of the action-planning controller. This code is executed at each cycle after the activation of the actor. ":=" is the assignment operator. In the simulations these parameter settings have been used: Decay = 0.00001, Gain = 0.01, MaxConfThresh = 0.15.

If the confidence is below the threshold, the action-planning controller disconnects the robot from the world, in the sense that it generates simulated experience by using the predictor and the matcher to simulate experience (see Figure 8.4). In particular the action-planning controller uses the predictor to generate several "simulated walks", i.e. chains of predictions (images). Each simulated walk starts from the image that corresponds to the position occupied in the environment. Simulated walks tend to be different since the actor selects the actions stochastically. Simulated walks get gradually longer if the goal is not encountered, otherwise they tend to get shorter (see details below). While planning the confidence threshold decreases. This prevents the robot from getting stuck in places in which the controller is not capable of becoming "confident" enough to start to move. For example, in

some simulations where the threshold was kept fixed, the simulated robot got stuck between the arena's border and the upper-left obstacle. While acting, the threshold increases again and reaches the maximum level without exceeding it. This guarantees that the robot tends to move only when the confidence is above the maximum level of the threshold. If the threshold could only decrease, the simulated robot would tend not to plan anymore. When the simulated walks are generated, the actor and the critic are trained as if the robot were acting in the environment. This allows the evaluator to improve its evaluating capacity and the actor to become capable of reaching the goal when the robot starts to act.

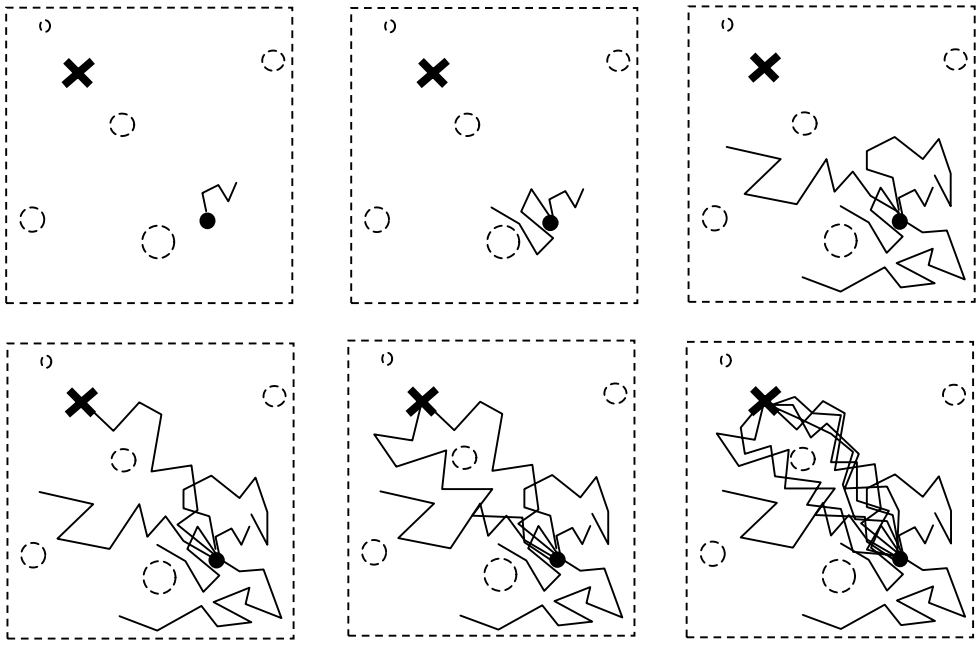


Figure 8.4: This sequence of graphs gives a general idea of the nature of the simulated walks (thin broken lines) generated by the action-planning controller from the current position. The simulated walks get longer until they start to encounter the goal. When this happens, their length tends to stabilise. Initially the simulated walks are directed in every direction, while after some time they become oriented from the start to the goal due to the training of the actor and critic. When the confidence measured in correspondence to the image of the position currently occupied by the robot reaches the threshold, the robot starts to act (not shown in the graphs).

Now the pseudo-code illustrated in Figure 8.3 is explained in detail. At the beginning of the simulation, when a new goal is assigned to the simulated robot, the variable `NewGoalHasBeenAssigned` is `TRUE`, and the algorithm does some variable settings (line 1 to 5). The whole algorithm is executed at each cycle of the simulation. This implies the execution of either one cycle of action or one cycle of planning depending on the system's mode (planning or acting mode). The mode (variable `Planning`) is decided each time the system receives an input from the world (line 6 and 7) on the basis of the system's "confidence" (line 8 to 13). The confidence is defined as the highest of the actions' probabilities measured at the position currently occupied by the simulated robot. If the confidence is above a certain threshold the system acts in the world and the predictor is not used (line 38). If the confidence is below the threshold, the action-planning controller "disconnects" the robot from the world (line 10, 11, 15 and 20), in the sense that it starts to generate simulated experience by using the predictor and the matcher (in line 36 the predictor produces one of the predictions that make up the chain of predictions, and in line 31 the

matcher checks if the chain encounters the goal). Each chain of predictions starts from the image that corresponds to the position currently occupied by the simulated robot. In fact when the variable `Planning` is set at true (line 11), $\mathbf{x}_t$ and $\mathbf{y}_t$ come from the simulated robot's sensors (line 6, 7). Notice that chains tend to be different since the system selects actions stochastically (line 29 and 30). Prediction chains get gradually longer if the goal is not encountered (line 2, 22 and 23), otherwise they tend to get shorter (line 25). While planning, the confidence threshold decreases (line 17). This prevents the robot from getting stuck in places in which the system is incapable of becoming "confident" enough to start to move (for example, without this mechanism the simulated robot got stuck between the arena's border and the northwest landmark). While acting, the threshold increases again and reaches the maximum level without exceeding it (line 14). This guarantees that the simulated robot tends to move only when the confidence is above the maximum level of the threshold. In the simulations the parameters of the algorithm are set as follows: Decay = 0.000001, Gain = 0.01, MaxConfThresh = 0.15. Each time a chain of prediction is terminated (either because the goal has been encountered or because it has reached a maximum length, line 21) the system "connects" again to the sensors and effectors (line 26, 6 and 7), updates the mode (line 8 to 13), and starts to act or to generate another chain of predictions. While the simulated walks are generated, the actor and the evaluator are trained with reinforcement learning as if the robot were acting in the world (line 28 to 34). This allows the evaluator to improve its evaluating capacity and the actor to shape the action probabilities. When the system stops planning and acts in the world (line 10, 13 and 38) it reaches the goal following a path that tends to be straight.

8.4 Results and Interpretation

8.4.1 Taskable Planning vs. Reactive Behaviour

The first two simulations have been run to test the taskability of the planning controller. This has been done by comparing the performance of the planning controller with the performance of the controller with reactive components only. During a simulation the simulated robot is set at the start, and its task is to reach the northwest goal. Each time the simulated robot reaches the goal it is set at another randomly-drawn position of the arena. This is done for 50,000 cycles. Then the simulated robot is set again at the start and is assigned the northeast goal, pursued for 50,000 cycles with the same modalities (random start after each success). The same is done for the southwest goal. Each time a new goal is assigned to the simulated robot, the weights of the evaluator and actor are randomised in the interval [-0.001, +0.001] so they can be used for the new goal.

Figure 8.5 reports the results of these simulations (averaged over 10 random seeds). For both the reactive and planning controllers the number of actions taken to reach the goal has been measured and plotted against the cumulated simulation cycles (this measure has been sampled every 100 cycles, and then smoothed with a 10-step moving average). Each cycle reported in the graph implies the execution of one action and eventually, if the controller is planning, several planning cycles. In the case of planning the number of planning cycles per action has also been measured and plotted in the graphs.

Several facts emerge from these simulations. When a new goal is assigned to the reactive controller, it reaches it in about 2000 steps on average (this approximately corresponds to the performance of a controller yielding a random walk). After repeated trials the reactive controller learns to reach the target in fewer steps, about 40 on the average, from any position

of the arena (the optimal path, not considering noise and obstacles, is about 10-15 steps long on the average). This same pattern is repeated for the three goals assigned to the reactive controller in a sequence. The standard deviation of the performance over the 10 simulations run with different random seeds is quite high (see graph).

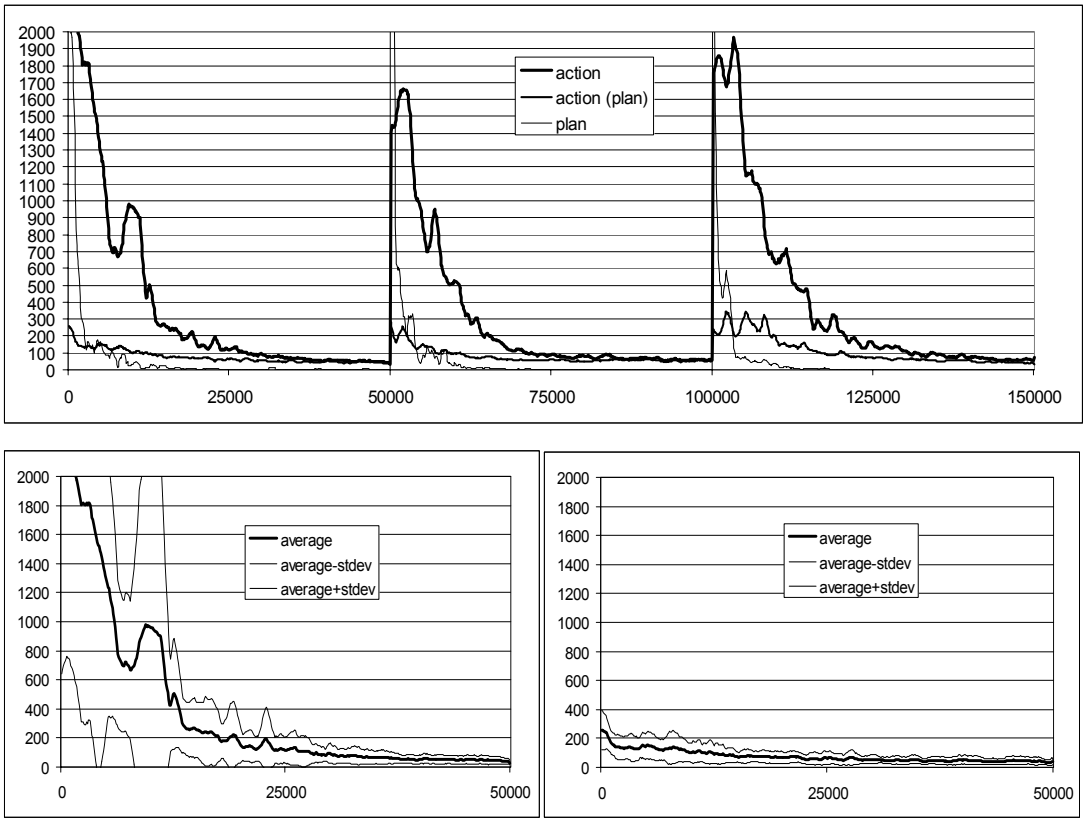


Figure 8.5: Top: Performance (y-axis: sampled every 100 cycles, and then smoothed with a 10-step moving average) for the three goals of the learning controller and planning controller, against the cumulated cycles (x-axis). "action" = steps per success; "action (plan)" = steps per success; "plan" = planning cycles per success. Each plot is an average over 10 simulations run with different random seeds. Bottom: average performance, average performance minus the standard deviation and average performance plus the standard deviation of the learning controller (left) and planning controller (right) for the northwest goal.

When a new goal is assigned to the planning controller, it reaches it in about 200 steps *from the very first time it pursues the goal.* This result is achieved through a considerable amount of planning processing: the planning cycles that the controller spends planning before reaching the goal *the first time* (averaged over 10 random seeds) are 62,004 40,116 and 17,840 for the northwest, northeast, southwest goals respectively. During this planning activity the skills of the evaluator and actor improve so that when the controller decides to act in the world it can achieve the goal with a performance superior to the performance of the reactive controller (random walk). If the confidence threshold is set at a higher value, 0.25, the performance of the planning controller is even better: it takes about 50 steps to reach the goal (see Figure 8.6). The standard deviation of the performance over the 10 simulations run with different random seeds is quite low (see Figure 8.5). This implies that planning not only

improves the performance on the average, but also drastically improves the consistency of success.

The number of planning cycles required by the north-west goal for the first success (62,004) is high in comparison to the cycles required by the other two goals because the north-west goal is more distant from the start. The difference of planning cycles between the other two goals (40,116 and 17,840) is probably caused by the configuration of the landmarks/obstacles and the functioning of the model of the environment for different areas of the environment. The fact that, for all the three goals, so many planning cycles are required is discussed in s. 8.5.

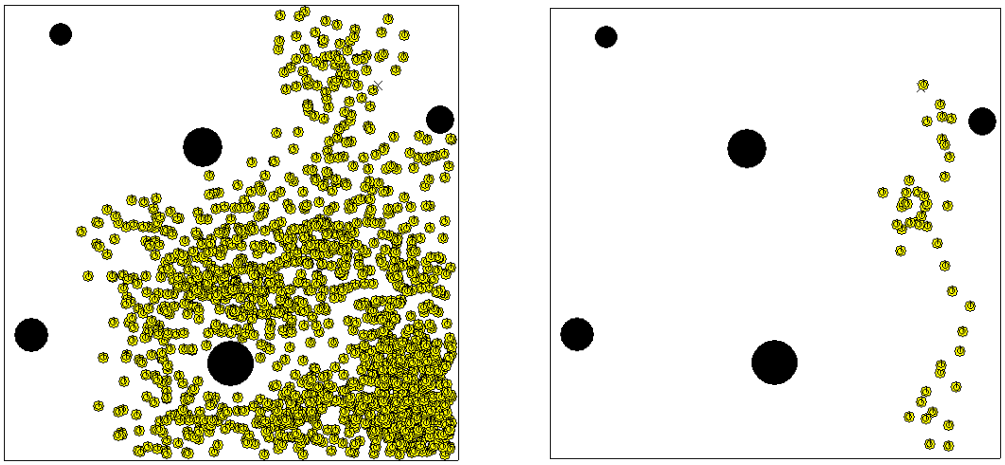


Figure 8.6: Left: Positions occupied by the simulated robot the first time that it reaches the northeast goal by reinforcement learning. Right: positions occupied by the simulated robot the first time that it reaches the northeast goal by planning. In this particular experiment the confidence threshold was set at a higher value than in the other experiments (0.25) to stress the fact that since the first time the robot reaches the goal it follows a quite efficient and straight path.

The difference of the performance of the reactive controller and planning controllers is particularly relevant because it shows that the planning controller is taskable. If fact: (a) it reaches the goal only on the basis of the information about the goal (plus the information stored in the predictor, i.e. the model of the world); (b) it is more efficient than the corresponding reactive controller from the first time the goal is pursued (cf. s. 5.1).

It is interesting to frame the results in terms of the formalism introduced in s. 3.1. The planning controller is capable of building the mapping $S \times A \rightarrow [0, 1]$ relative to the particular goal pursued, without reaching the goal. Moreover the controller solves the more complex mapping $S \times A \times S_g \rightarrow [0, 1]$ dynamically, i.e. for the particular goal assigned, by using the small memory capacity of the evaluator and actor's weights. On the other hand, the reactive controller can solve the $S \times A \rightarrow [0, 1]$ mapping problem only at the cost of repeated experience of the goal itself. This is also true when the reactive controller has to solve the $S \times A \times S_g \rightarrow [0, 1]$ mapping problem (chapter 7 has shown how a modular architecture can improve the learning speed of the system).

Some caveats are needed to qualify these results. First, the planning controller has storage capacity (weights of the predictor) that could be used to increase the storage capacity of the reactive controller's actor and critic. Second, the acquisition of a good model of the environment requires experience that could be used to learn to reach specific goals. Notwithstanding these caveats, an important fact remains true: the planning controller is

capable of storing information in the predictor that is independent of the specific goal pursued, and this information can be flexibly used for any goal. This is not true of the reactive controller.

8.4.2 Focussing, Partial Policies and Replanning

Direct observation of the behaviour of the simulated robot makes it possible to understand how the controller works in terms of partial policies and replanning. As mentioned, when assigned a goal the planning controller spends many cycles planning before reaching it. After some time it begins to act. Two behaviours have been observed:

- Sometimes when the simulated robot starts to act, it reaches the goal along a quite straight path without ever stopping to plan again.
- Some other times, when the simulated robot starts to act, it arrives in some states where the confidence is low. For example, sometimes it goes right past the goal, or ends up in states far from the direct start-goal path. In these cases the simulated robot stops and the controller starts to plan again.

These two behaviours are interesting because:

- The first behaviour shows that when the confidence at the start reaches the threshold, the confidence for the states closer to the goal and along the direct start-goal path is also higher than the threshold (indicated by the fact that the simulated robot does not stop to re-plan). This can be explained by observing that reinforcement learning works by updating the evaluations (and hence the actions' probabilities) from the goal backwards towards other states. The resulting behaviour is desirable because it avoids the controller planning, executing one action, re-planning, executing another action, etc.; i.e. it assures that the partial policy prepared at the beginning allow the simulated robot to reach the goal with high probability. This probability can be set indirectly by setting the maximum confidence threshold.
- The second behaviour shows that the policy prepared is really a partial policy. In fact if the simulated robot reaches “unexpected” states it starts to re-plan. Another result confirms that the policy is partial and focussed on the goal and the current start. When the simulated robot reaches the goal, it is set at a new position of the environment chosen at random. When this happens, the controller always starts to re-plan (except when it is set along the old direct start-goal path). This shows that the old policy is not adequate for states that lie far away from the old direct start-goal path, i.e. that the old policy was actually a partial policy. It should be noticed that this property of the planner descends from the fact that reinforcement learning methods are being used to implement planning.

With repeated experience the planning controller shows two other relevant changes in behaviour. First, the amount of planning needed to reach the goal decreases sharply with experience, and soon falls to zero. This happens because the outcome of the planning process is stored (“compiled”) in the weights of the evaluator and actor, so that the reactive components of the controller become “confident” enough to reach the goal without further planning. This shows how the controller is capable of finding a balance between acting and planning. Second, experience in the world further improves the performance, bringing it from about 200 to about 50 steps. This means that the outcomes of the simulated and real experience merge suitably in the weights of the evaluator and actor. This is a typical property of Dyna-like architectures.

8.4.3 Neural Networks for Prediction: "True" Images as Attractors?

The predictor incorporates the state transition-function part of the model of the environment. This is a critical component of the controller because the whole process of planning relies on it, so it is important to analyse how it works. This is done in this section.

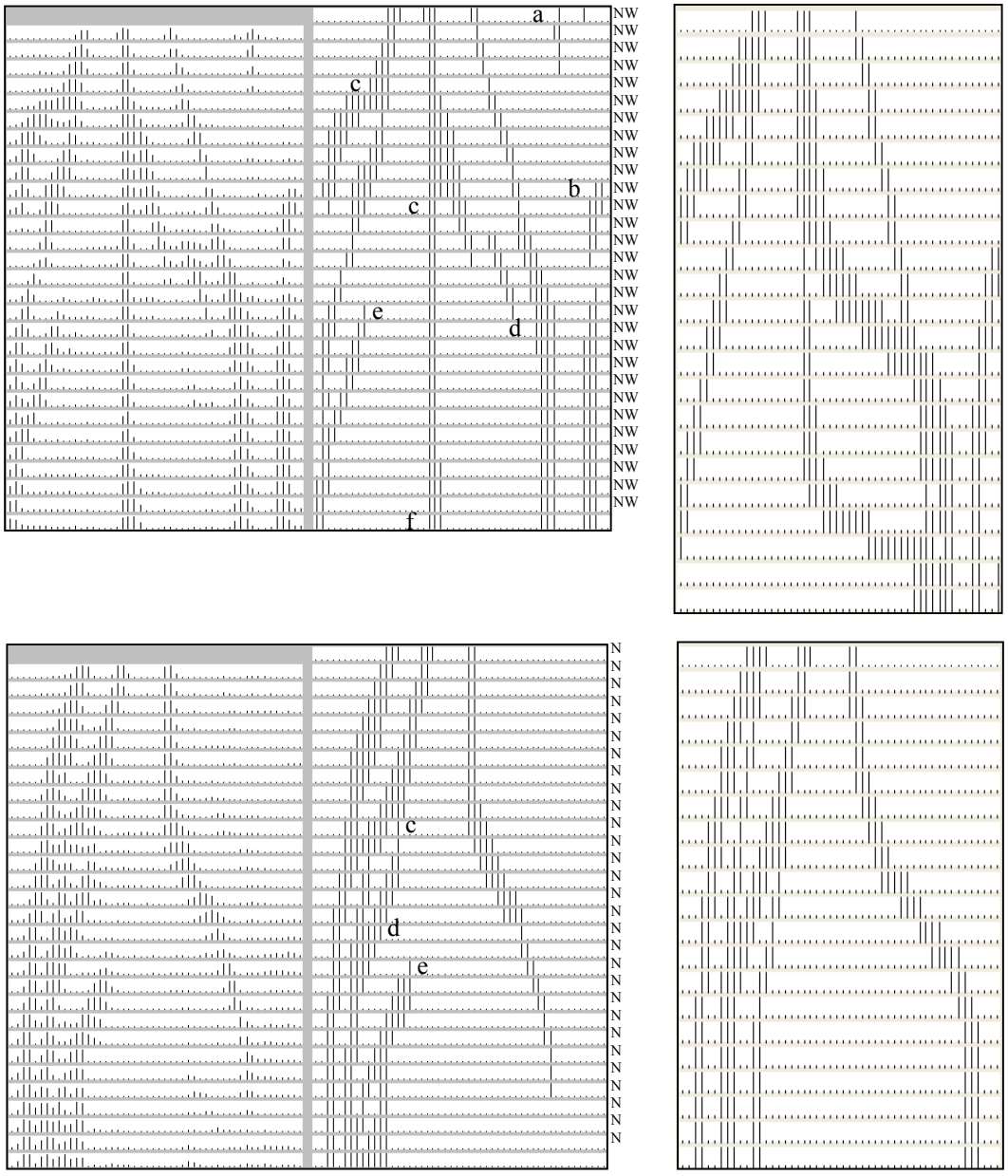


Figure 8.7: The graphs on the left report two examples of simulated walks from the start to the northwest goal (top left graphs) and to the northeast goal (bottom left graphs). The action selected has been kept fixed to northwest and north respectively for the two goals. For each step the continuous output of the predictor (left part of the graphs on the left), the binary prediction (right part of the graphs on the left), and the selected action (columns of letters of the graphs on the left) are shown. The succession of the images is reported from the top to the bottom of the graphs. The first binary image corresponds to the image that the simulated robot perceives at the position currently occupied in the world, while the other images are generated through the predictor. The graphs on the right report the true images perceived by the robot while it moves from the start to the northwest (top left graph) and to the north east goal (top right graph) respectively. They have been reported to allow the evaluation of

the quality of the simulated walks generated by the predictor. Notice that they have been stretched to have a better correspondence with the simulated walks: this means that the simulated walks tend to be made up by more steps than the real walks. Notice that the predictor is capable of coping with noise (a: an activation that is generated by noise and not by a landmark, is suppressed after some time), is capable of predicting the appearance of landmarks in the scene (b), is capable of predicting the appearance of a landmarks form behind other landmarks (c) and the disappearance of landmarks behind other landmarks (d); however, the predictor also produces distorted images, for example it generates non-existing landmarks (e), has the tendency to generate persistent landmarks in the centre of the scene (f: this happens because the stability of a landmark's image in the middle of the scene when the simulated robot approaches it is a very strong regularity), has biases (in the first example the simulated walk leads to the left of the northeast landmark, while the simulated walk leads towards it; in the second example the simulated walk leads further away from the north east landmark compared to the real walk), tends to generate images that change more slowly than real images (as mentioned, the number of real images on the right are less than images generated by the predictor).

The behaviour of the predictor can be understood by investigating the nature of the predictions that it generates while planning. Figure 8.7 shows the simulated walks generated by the planning controller. The simulated robot begins to move at the start. The "selected" action has *been fixed* to northwest (or north for the second graph shown in the figure) by suitably changing the simulation program. A sequence of 29 successive predictions has been recorded for both cases and plotted in the figure. A good capacity of the predictor to anticipate the consequences of the actions is apparent from the graphs. For example the predictor is capable of coping with noise, is capable of anticipating the appearance of landmarks from behind other landmarks, or the disappearance of them. To check this, consider the graphs reported on the right of the figure: they report the images perceived by the simulated robot while it moves toward northeast or north (alternatively consider Figure 8.1 and consider how the landmarks should appear to the simulated robot while it moves along a straight path that goes from the start to the northwest or the north goal). The predictor also makes some mistakes (that, incidentally, are quite interesting). For example it has biases, e.g. it tends to keep fixed images of landmarks in the middle of the scene because this is a strong regularity observed in the environment, it "loses" the image of some landmarks and predicts to see non-existing landmarks.

To collect other data on the predictor's behaviour, a simulation where the simulated robot selects the actions autonomously has been run. Figure 8.8 reports these data. The simulated robot guided by the planning controller was set at the centre of the arena and had to plan to go to the northwest goal. The confidence threshold was set at a high value (1) so that the controller kept planning without ever moving. After 200,000 cycles of planning the predictions starting from the current real visual input (i.e. the one corresponding to the centre of the arena) were recorded until the goal was "mentally" reached. The two graphs reported in the figure refer to two runs of the experiment that differ in the starting image because of perception noise. These and other runs show the coherence of the simulated walks generated by the predictor. For example notice that in the bottom graph of the picture, the image of a wrongly "lost" landmark (the one at the northwest corner, cf. Figure 8.1) is recovered (you can see this by comparing the two graphs). The same type of simulations also show mental walks that fail to reach the goal, for example because they converge to images that do not correspond to positions in the environment.

The predictor's capacity to generate images that approximately correspond to real situations for 29 succeeding steps is quite surprising. In fact one would expect that noise would *accumulate* when some noise predictions are used as input patterns to generate further images. Instead, it seems that there is a mechanism that keeps this noise under control. A

hypothesis can be formulated about this mechanism. The images that correspond to real situations tend to be “attractors” for the images of the simulated walk generated by the predictor, with “basins” of attraction that capture the “noisy” images. Notice that here these concepts are used because they facilitate the description of the results of the experiments, but are not intended in a rigorous technical sense (e.g. cf. Wuensche, 1998, for a rigorous definition and use of these concepts). This property of the predictor depends on the fact that it is trained in the environment, and on the neural networks' property of prototype extraction (cf. s. 4.4.2). In fact when the predictor is trained, the images used as input and teaching output are the ones that correspond to real views of the environment. As a consequence of this and the prototype extraction property, even when some images corrupted by noise are sent to the predictor as input, the output will tend to be an image that corresponds to a real view and the noise will tend to be filtered out.

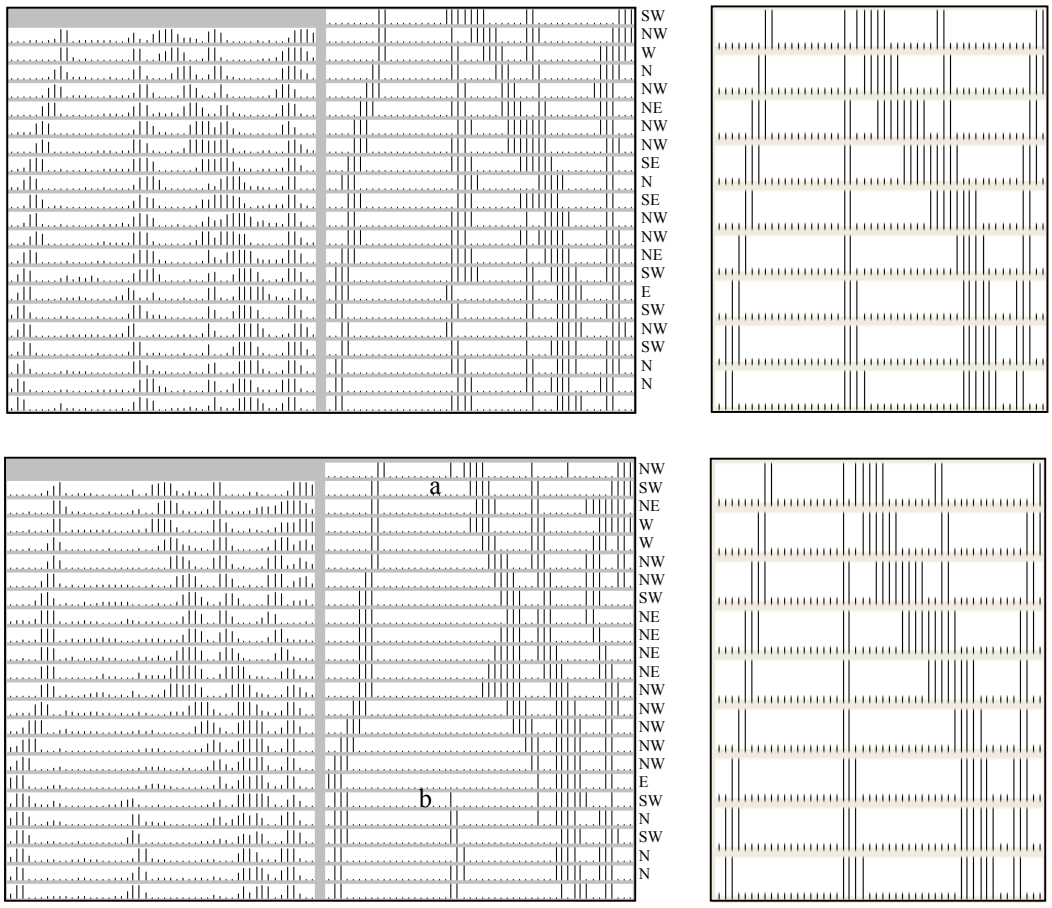


Figure 8.8: The graphs on the left report two examples (recorded after 200,000 planning cycles) of a successful simulated walk that starts from the centre of the arena and ends in the northwest goal. The two examples differ in the initial image because of the sensors' noise. For each step the continuous output of the predictor, the binary prediction, and the selected action are shown. The graphs on the right show the images of the environment perceived by a simulated robot that goes from the centre of the arena to the northwest goal following a straight line. Notice how a landmark lost in a is recovered in b.

If true, this would be a fundamental mechanism to preserve the correspondence of the predicted images with positions in the environment. Further investigation is needed to test this explanation and to study ways of avoiding spurious attractors.

8.5 Limitations of the Neural Forward Planner

The functioning of the planner depends on the possibility of building a reliable model of the environment. In fact first the predictor is suitably trained, then planning starts on the basis of the assumption that the model of the world is sufficiently accurate and that training the evaluator and actor through it will improve their abilities to evaluate and act. This assumption is not true in general for all task domains. Moreover, one might want that planning start in any moment of the "life" of the agent, so the planner should be capable of evaluating the quality of its predictions and deciding if planning or not accordingly. This drawback of the planner also affects all the planners presented in the following chapters and the planners reviewed in s. 4.5.

The controller presented in this chapter has some limitations. One is that the first reaching of the goal within the model of the environment is achieved as the result of a random-walk search. This is very inefficient (cf. s. 8.4.1). Given that the goal and the model of the environment are known it should be possible to carry out a goal-oriented search. This will be done in chapter 9.

There is another limitation connected with the previous one. The current controller predicts and plans at the same level where it acts, i.e. at the level of the "primitive-actions" available to the controller. At this level of detail it is possible that the model of the environment cannot be accurate enough because the environment is intrinsically unpredictable. A solution would be to plan at a coarse level, where details are ignored, and to act at a fine level. In fact, abstraction can allow a better prediction (cf. Russell and Norvig, 1998, p. 409). How would it be possible to implement this kind of "coarse planning"? The thesis starts to tackle this problem in chapter 11.

8.6 Conclusion

This chapter has presented a neural network controller ("neural forward planner") that is inspired by Dyna-PI architectures, but that, unlike them, is taskable. It has achieved this result by introducing the "matcher", a new neural network for goal detection. This has eliminated the need, present in the original Dyna-PI architecture, for the part of the model of the environment related to the reward. The simulations have shown that taskability allows the planning controller to reach the goal in fewer steps than the underlying reactive controller *from the first time* the goal is pursued. This is one of the real advantages of planning controllers vs. reactive controllers.

Planning is executed by using the knowledge stored in a modular network, the "predictor". The predictor models the effects produced by the actions when they are executed in the environment. When the controller is planning, the predictor is used to generate sequences of future states starting from the current state (trajectory sampling, cf. s. 3.4). During planning these sequences are generated iteratively and have a length that is increased if the goal is not encountered, and is decreased when it is encountered. This planning strategy allows the system to focus planning on states concentrated around the start-goal path, and to build partial policies.

The experiments have also shown how the controller not only builds partial policies, but also uses re-planning when necessary. In particular they have shown that when the controller encounters relatively novel states it executes planning, while when it encounters relatively familiar states it acts reactively. This also implies that after the controller has enough experience about one goal it does not need to plan anymore.

The planning processes use representations consisting of "images" generated by the predictor and learnt autonomously. Some simulations have shown that the predictor has a significant capacity to maintain the consistency between the simulated trajectories generated and the possible trajectories experienced in the world when the policy is executed. A possible explanation of this is that images that correspond to views of the environment are "attractors" for the images generated by the predictor.

The experiments have also demonstrated some limitations of the controller, such as the need to assume the accurateness of the model of the world when planning is executed, and the large number of planning cycles needed to reach the goal the first time due to the fact that the initial search is carried out on the basis of a random walk. They have also shown that planning takes place at the same fine level of actions, and hence it is not very efficient.

9 The Neural Bidirectional Planner

9.1 Introduction: More Efficient Exploration

Problems Tackled. The problem addressed by this chapter is how to further focus planning on relevant regions of the state space within the model of the environment. The planning controller presented in chapter 8 executed forward "simulated walks" within the model of the environment. These simulated walks started from the image of the position currently occupied by the simulated robot and tried to reach the goal within the model of the environment. This controller had a basic problem shared with reinforcement learning methods and Dyna architectures in general: the first time that the goal was pursued the controller tried to reach the goal on the basis of a *random walk*. With large state spaces this random walk encountered the goal rarely, hence the whole process of planning was very slow. The controller proposed and implemented in this chapter tries to solve this problem.

Overview. The controller proposed here implements planning by generating simulated experience both forward from the current state and backward from the goal. The simulated forward walks are as the ones executed by the forward planner of chapter 8. The simulated "backward walks" are based on two new neural components, the back-actor and the back-predictor, respectively capable of "guessing which action could have brought to the current state" and of "guessing what was the state from which the system has reached the current state". The simulated backward walks start from the goal and explore other states from it in all the directions (as we shall see, the controller learns to "escape" in straight lines from the goal). While this is done, the evaluations, the policy, and the "back-policy" are updated. The backward walks produce two important advantages when compared to the forward walks:

- An efficient exploration of the model of the environment: the goal is "found" immediately, given that the backward walks start from it.
- A quick propagation of the evaluations backward from the goal, given that the evaluation of each state is updated on the basis of an evaluation of a state that has just been updated.

The simulations will also show that the bidirectional planner has the strengths of the forward planner of chapter 8. First, it is taskable and it is even more "goal-oriented" because the planning activity focuses around the goal. Second, it is capable of transferring skills between problems with same goal and different starts. Third, when it solves a problem several times it is capable of accumulating "skills" within the reactive components so that planning is no longer necessary.

What is New and Related Literature. The general idea of planning backward from the goal is not new. Literature on problem solving has already showed the advantages of searching forward from the start and backward from the goal by studying "bidirectional search" (Pohl, 1971; cf. s. 13.1.1). STRIPS planning (Fikes and Nilsson, 1971; cf. s. 2.3) is completely based on backward searches from the goal. However, the specific mechanisms that have been proposed by these two branches of research are not applicable to stochastic environments such as the ones considered here.

The idea of the backward updating of the evaluations has also been investigated within the reinforcement learning literature. In particular Lin (1992), Thrun (1992), Reynolds (2002), have shown how updating evaluations backward from goal is a powerful strategy because state-evaluations are updated on the basis of evaluations updated in the previous time steps. However, the systems proposed by these authors use memory structures to store sequences of states that led to the goal, or other type of experiences, in order to use them for iterated "backward" backups. If one wants to use neural networks, this strategy would raise the problem of how implementing these memory structures and how using the information stored in them. Prioritised sweeping (Moore and Atkenson, 1993; Wiering et al., 1998; Dearden, 2001; cf. s. 3.4), by updating states or state variables whose evaluations would change a lot if updated, often propagates evaluations backwards from states close to the goal.

Chapter's Outline. S. 9.2 presents the task used to test the controller. S. 9.3 presents the details of the components of the bidirectional planner. S. 9.4.1 shows that the backward planner has the same strengths of the forward planner. S. 0 shows the advantages of the bidirectional planner vs. the forward planner in terms of exploration and propagation of evaluations. Finally s. 9.5, 9.6 and 9.7 respectively analyse the drawbacks of the models, propose a controller simpler than the bidirectional planner, and draw conclusions.

9.2 Scenario of Simulations

The scenario and the simulated robot used in this chapter are the ones illustrated in s. 6.2 (cf. Figure 6.2). Figure 9.1 shows the scenario and the particular goal and start positions used in this chapter.

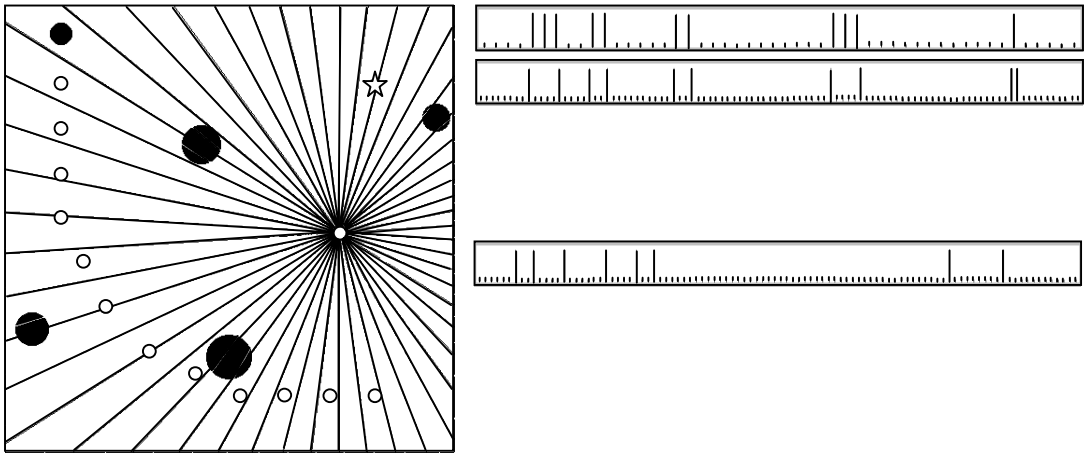


Figure 9.1: Left: The scenario of test containing the goal (star), five landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), the simulated robot (circle at origin or rays), and the 12 start positions (white circles) at which the simulated robot is repeatedly set in ordered succession. Right, in order: The activation of the simulated robot's sensors at its current position (affected by noise), the corresponding contrasts, and the goal (contrasts).

The task the simulated robot has to accomplish is to reach the goal position from the start position at the northwest corner. All the 12 start positions are used, from the one at the northwest corner to the one at the south east corner. When the last start position at the southeast corner has been used, the whole cycle is repeated starting from the start position at

the northwest corner. The particular start states have been chosen to guarantee the same distance from the goal. This was important for the measurements reported below.

9.3 Architectures and Algorithms

9.3.1 The Reactive Components of the Architecture

Figure 9.2 shows both the reinforcement learning and the planning components of the architecture. The reactive components of the controller have the same architecture and function as the components of the forward planner presented in chapter 8.

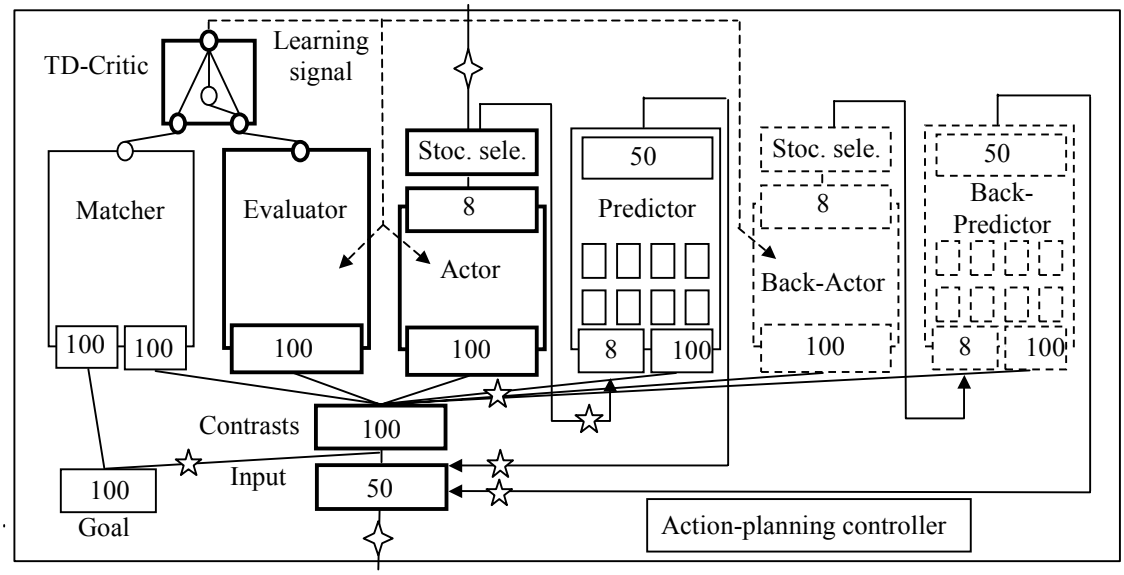


Figure 9.2: The controller of the simulated robot. Networks with a bold, thin and dashed border implement reinforcement learning, forward planning, and backward planning respectively. Arcs and arrows indicate forward and backward connections respectively. They "copy" an activation pattern from one layer to another. The four and five spike stars indicate the channels respectively set open and close by the action-planning controller when acting (vice versa when planning). Dashed arrays indicate the learning signal used to update the weights of the evaluator, actor and back-actor.

9.3.2 The Planning Components of the Architecture: Forward Planning

The predictor, that allows "forward planning", is the same as the one employed in chapter 8. Recall that it is a set of 8 feed-forward two-layer networks ("experts") with sigmoid output units, each corresponding to one action. Each expert takes $\mathbf{y}_t$ as input, and is specialised to predict the following sensors' activation $\mathbf{x}_{t+1}$ if the action corresponding to it is executed. The experts are trained while the simulated robot navigates randomly in the environment for 200,000 cycles. This training is done before the main simulations illustrated below take place.

The action-planning controller is a hand-designed algorithm that decides when the controller has to plan or to act, directs the flow of information among the different components of the whole system, and manages the generation of the forward and backward

simulated walks. The pseudo-code of this algorithm is illustrated in Figure 8.3. The main differences between this algorithm and the one employed in chapter 8 are as follows:

- When the controller is planning, it has to decide when to execute forward planning and when to execute backward planning.
- It has to control the flow of information between the different components of the system when the controller is executing backward planning.

```
01 IF(NewGoalHasBeenAssigned)
02   MaxStepsPlan := 1
03   ConfThresh := MaxConfThresh
04   ForwardPlanning := TRUE
05   StepPlan := 0
06   InputFromWorld := TRUE
07 IF(InputFromWorld)
08   System gets input x_t (y_t) from the robot's sensors
09   Actor gets y_t and gives m_t
10   Confidence is computed on the basis of m_t
11   IF(Confidence < ConfThresh)
12     Planning := TRUE
13   ELSE
14     Planning := FALSE
15     ConfThresh := MIN(MaxConfThresh, ConfThresh + Gain)
16 IF(Planning)
17   StepPlan := StepPlan + 1
18   ConfThresh := ConfThresh - Decay
19   IF(ForwardPlanning)
20     IF(InputFromWorld = FALSE)
21       System uses predictor's output y_t as input
22     ELSE
23       InputFromWorld := FALSE
24     IF(GoalReached OR StepPlan = MaxStepsPlan)
25       IF(StepPlan = MaxStepsPlan)
26         MaxStepsPlan := MaxStepsPlan + 1
27       ELSE
28         MaxStepsPlan := MIN(MaxStepsPlan, StepPlan * 2)
29       InputFromWorld := TRUE
30       IF(BidirectionalPlanning)
31         ForwardPlanning := FALSE
32         ForwardSteps := StepPlan
33         GoalAsInput := TRUE
34         InputFromWorld := FALSE
35       StepPlan := 0
36   ELSE
37     IF(GoalAsInput = TRUE)
38       System uses goal y^g as input
39       GoalAsInput := FALSE
40     ELSE
41       System uses back-predictor's output y_t as input
42     IF(StepPlan = ForwardSteps)
43       ForwardPlanning := TRUE
44       InputFromWorld := TRUE
45       StepPlan := 0
46 IF(Planning)
47   IF(ForwardPlanning)
48     Evaluator gets y_t and gives V'^π[y_t]
49     Actor gets y_t and gives m_t
50     Stochastic selector gets m_t and gives a_t
51     Predictor gets y_t, a_t and gives x_t+1 (y_t+1)
52     Matcher gets y_g, y_t and gives r_t
```

```
53      TD-Critic gets V'^π[y_{t-1}], V'^π[y_t], r_t, gives e_{t-1}
54      Evaluator gets y_{t-1}, e_{t-1} and learns
55      Actor gets y_{t-1}, m_{t-1}, a_{t-1}, e_{t-1} and learns
56      IF(BidirectionalPlanning)
57        Back-Actor gets y_t and gives m_{t-1}
58        Back-Actor gets y_t, m_{t-1}, a_{t-1}(actor), e_{t-1} and learns
59    ELSE
60      Back-actor gets y_t and gives m_{t-1}
61      Back-stochastic selector gets m_{t-1} and gives a_{t-1}
62      Back-predictor gets y_t, a_{t-1} and gives x_{t-1} (y_{t-1})
63      Evaluator gets y_{t-1} and gives V'^π[y_{t-1}]
64      Matcher gets y_g, y_t and gives r_t
65      TD-Critic gets V'^π[y_{t-1}], V'^π[y_t], r_t and gives e_{t-1}
66      Evaluator gets y_{t-1}, e_{t-1} and learns
67      Back-actor gets y_t, m_{t-1}, a_{t-1}, e_{t-1} and learns
68      Actor gets y_{t-1} and gives m_{t-1}
69      Actor gets y_{t-1}, m_{t-1} (actor), a_{t-1} (back-actor), e_{t-1} and
learns
70  ELSE
71    Evaluator gets y_t and gives V'^π[y_t]
72    Actor gets y_t and gives m_t (already done in line 9)
73    Stochastic selector gets m_t and gives a_t
74    Matcher gets y_g, y_t and gives r_t
75    TD-Critic gets V'^π[y_{t-1}], V'^π[y_t], r_t and gives e_{t-1}
76    Evaluator gets y_{t-1}, e_{t-1} and learns
77    Actor gets y_{t-1}, m_{t-1}, a_{t-1}, e_{t-1} and learns
78    System executes a_t in the world
79    IF(BidirectionalPlanning)
80      Back-Actor gets y_t and gives m_{t-1}
81      Back-Actor gets y_t, m_{t-1}, a_{t-1} (actor), e_{t-1} and learns
```

Figure 9.3: Pseudo-code of the planning-acting controller. The algorithm is executed at each cycle. ":=" is the assignment operator. In the simulation the parameters are set as follows: Decay = 0.000001, Gain = 0.01, MaxConfThresh = 0.15

When the variable `BidirectionalPlanning` is set at `FALSE` the whole controller is equivalent to the forward planner investigated in chapter 8. Recall that in this case the controller can be either in planning or acting mode. The mode is decided on the basis of the controller's "confidence", the highest of the actions' probabilities. If the confidence is above a certain threshold the controller acts in the world, otherwise it simulates experience by using the predictor. When the controller is forward planning the evaluator and actor function and learn in the same way as they do when acting in the world. As for the previous chapter, the parameters are set as follows: Decay = 0.000001, Gain = 0.01 and MaxConfThresh = 0.15.

9.3.3 The Planning Components of the Architecture: Bidirectional Planning

If the variable BidirectionalPlanning of the algorithm illustrated in Figure 9.3 is set at `TRUE`, the algorithm implements bidirectional planning. As the forward planner, the bidirectional planner decides if planning or acting on the basis of the measure of confidence at the position currently occupied by the simulated robot (line 7 to 14). The major difference between the two algorithms is that while planning the bidirectional planner generates prediction chains alternately forward from the current position image (line 47 to 55 implement one cycle of forward chain) and backward from the goal image (line 38; line 60 to 69 implement one cycle of backward chain). Planning always starts with a forward simulated walk and ends with a

backward simulated walk. The length of each backward chain is the same as the last forward chain (line 32 and 42). Forward chains are executed as in forward planning. Backward chains are executed through the "back-predictor" and "back-actor".

The back-predictor is a network with the same architecture as the predictor. While the predictor is trained to produce the association $\mathbf{y}_t$, $a_t \rightarrow \mathbf{x}_{t+1}$, the back-predictor is trained to produce the association $\mathbf{y}_t$, $a_{t-1} \rightarrow \mathbf{x}_{t-1}$ (the time indexes are used backward) i.e. to remember (or guess) which situation $\mathbf{x}_{t-1}$ led the system to the situation $\mathbf{y}_t$ after executing action a_{t-1} (line 62). Notice that each couple of experts of the predictor and of the back-predictor corresponding to a particular action could have been integrated in one bi-directional network associating $\mathbf{x}_t \leftrightarrow \mathbf{x}_{t+1}$ under action a_t. This has not been done since for simplicity only feed-forward networks have been used.

The back-actor has the same architecture as the actor, and is used to generate actions for the backward chains (the a_{t-1} of the association $\mathbf{y}_t$, $a_{t-1} \rightarrow \mathbf{x}_{t-1}$, see line 60 and 61). Before the tests shown below the back-actor weights are randomly drawn in the interval [-0.001, 0.001], so initially it selects actions randomly. During a back cycle that leads from $\mathbf{y}_t$ to $\mathbf{y}_{t-1}$ (from $\mathbf{x}_t$ to $\mathbf{x}_{t-1}$), after the back-actor selects the a_{t-1}, the merit of this action is updated according to the same formula used for the actor (see equation 3) and with the usual error $e_{t-1} = (r_t + \gamma V'^{\pi}[\mathbf{y}_t]) - V'^{\pi}[\mathbf{y}_{t-1}]$. However, now the merit of the action is updated using $\mathbf{y}_t$ as input for the back-actor (and not $\mathbf{y}_{t-1}$ as for the forward actor, line 67). Notice that with this training the back-actor learns to generate actions that lead to states with the *lowest possible evaluation* $V'^{\pi}[\mathbf{y}_{t-1}]$, i.e. states *far* from the goal and visited few times. When backward chains are generated, the actor and evaluator are also updated using the error e_{t-1}. In particular the actor produces the actions' merit $\mathbf{m}_{t-1}$ in correspondence to $\mathbf{y}_{t-1}$, and then its weights are updated on the basis of those merits and the action a_{t-1} selected by the back-actor and back-stochastic selector (line 68 and 69). During forward planning and acting, the back-actor is also trained by using e_{t-1} (line 56 to 58 and 79 to 81). To this end, the back-actor yields the actions' merit $\mathbf{m}_{t-1}$ in correspondence to $\mathbf{y}_t$, and then its weights are updated on the basis of those merits and the action a_{t-1} selected by the actor and stochastic selector for $\mathbf{y}_{t-1}$. The overall functioning of the backward planning algorithm can be summarized as follows. The back-actor learns to yield backward walks that "escape" from the goal in "straight" lines, hence creating a big area of positive evaluations around the goal. This area is easily "found" by the actor's forward walks that, as a consequence, expand the area toward the position occupied by the simulated robot. At the same time the actor becomes competent in the area where positive evaluations diffuse.

Some remarks about backward planning are due. Updating the evaluator when the back-actor is selecting the actions may cause some problems. In fact the actor-critic methods require that state evaluations reflect the expected reward averaged over the actions *selected by the current policy* (i.e. the actor, cf. s. 13.2.3). Notwithstanding this, the choice made here is justified because the actor's policy and the back-actor's "back-policy" tend to be quite similar for the following reasons:

- The actor and the back-actor have the same structure and are trained an equal number of times with the same error signals.
- A state from which the back-actor selects an action is perceptually very similar to the state to whom this action brings, and from which the actor selects its action.
- The direction of the maximum slope of the evaluation gradient field built by the actor and by the back-actor tends to be the same (i.e. toward the goal).

Incidentally notice that these observations suggest that maybe it is possible to integrate the actor and back-actor in a unique network.

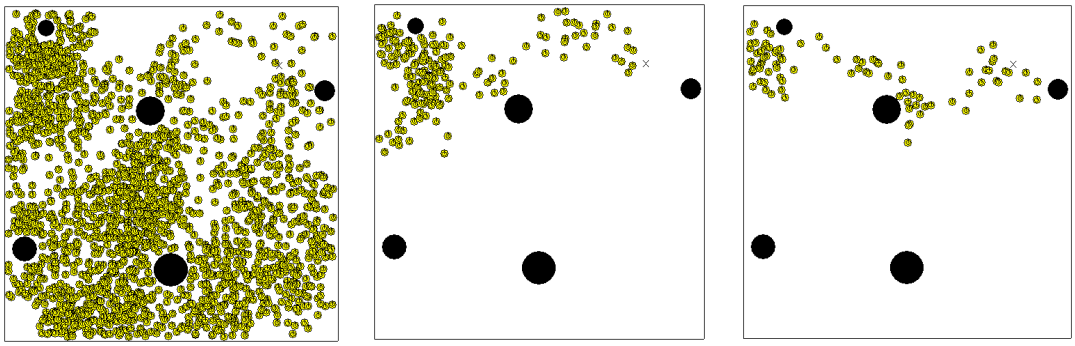


Figure 9.4: Positions occupied by the simulated robot the first time that it reaches the northeast goal by reinforcement learning (left) forward planning (centre) and bidirectional planning (right).

Backward planning should present two important advantages vs. forward planning. The first advantage is in terms of exploration. Updating the evaluations backward from the goal brings to immediately change the evaluations of states close to the goal. On the contrary, forward planning starts to update the evaluations only after the goal is encountered for the first time. Since the first search of the goal is usually done by random walk (but cf. Sutton, 1990; Thrun, 1992; Wyatt, 1997), the event can take very long to occur (the expected time is exponential in the number of steps separating the start from the goal, Thrun, 1992). The second advantage is in terms of propagation of evaluations between states. This is particularly fast if done backward from the goal because newly updated evaluations of states are used to update the evaluations of other states.

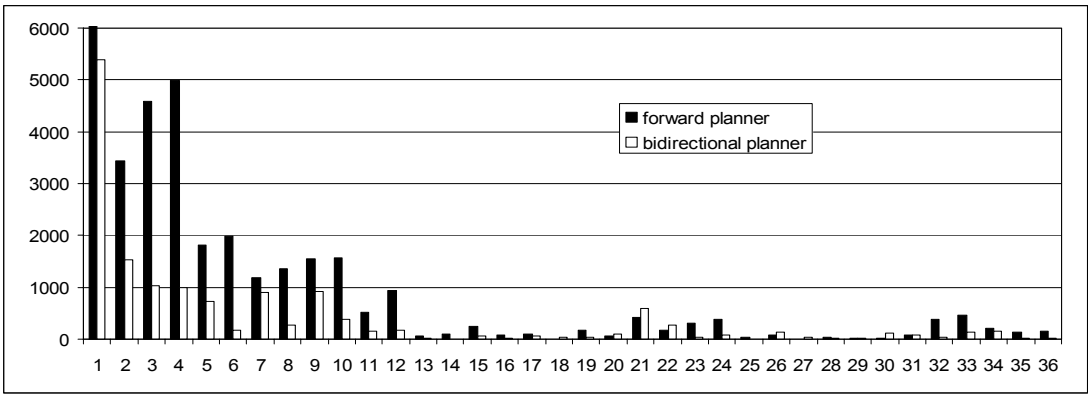


Figure 9.5: The graph reports the performance for both the forward planner and the bidirectional planner, averaged over 10 simulations run with different random seeds. Y-axis: number of cycles of planning for each success. X-axis: the first 36 consecutive successes. For graphical reasons the vertical axis has been cut at 6000: the forward planner took 52,923 planning cycles to achieve the first success.

9.4 Results and Interpretation

9.4.1 Common Strengths of the Forward-Planner and the Bidirectional Planner

The forward planner and the bidirectional planner have been tested and compared with the scenario and task illustrated previously. The test for each planner has been done 10 times with

different random seeds: the graphs shown below will report the averages over these 10 runs. Each time the simulated robot reaches the goal, the number of actions executed, and the number of planning cycles used to reach it, are measured.

The results of the simulations show that the two controllers share the following strengths. In order to plan, both controllers only need to know the start (or current) position and the goal position. Planning is carried out on the basis of the information stored in the predictor and the back-predictor. Moreover, the goal is reached efficiently from the first time it is pursued: 245 and 186 actions respectively executed by the forward and backward controller vs. an average of 1432 cycles executed by the random walk (Figure 9.4). This implies that the two planning controllers are taskable.

When the goal is pursued several times from the same start position, the performance improves both in terms of planning cycles (cf. Figure 9.5) and actions executed (Figure 9.6). Notice that when enough experience accumulates the goals are achieved reactively.

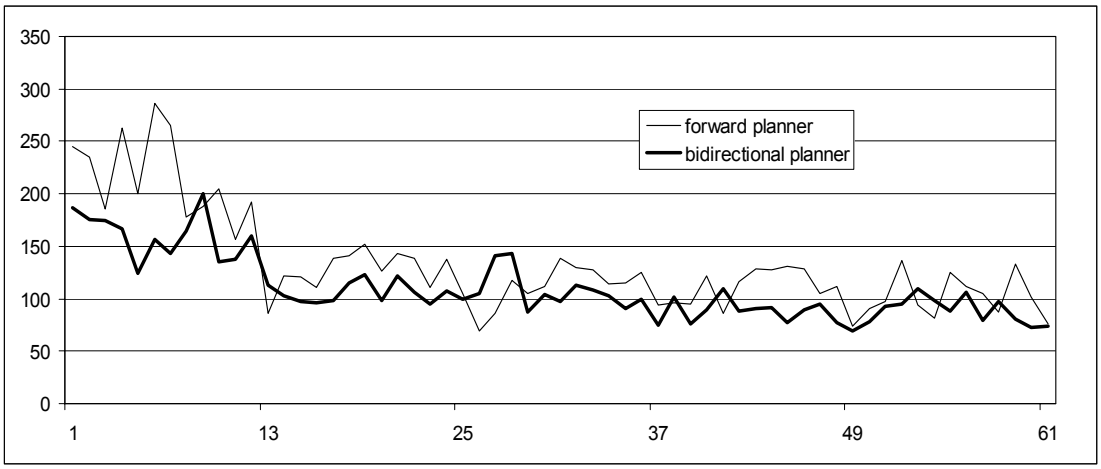


Figure 9.6: The graph reports the performance for both the forward planner and the bidirectional planner, averaged over 10 simulations run with different random seeds. Y-axis: number of actions executed for each success. X-axis: the first 61 consecutive successes. Curves have been used instead of a histogram to ease the comparison between the two conditions.

The knowledge that is gathered while planning to reach the goal from a given start is used to reach the same goal from different starts. Figure 9.5 and Figure 9.6 show that the planning and acting cycles needed to reach the goal starting from the different start positions, decrease steadily when the robot is set at the succeeding start positions (the optimal path, not considering noise and obstacles, is about 15 steps long). This happens even if the start positions are new (cf. data about the first 12 successes). The reason why this happens is that when pursuing a goal, the controller visits several states within the model of the world and learns what to do in order to reach the goal from them. This knowledge is also useful when the controller has to reach the same goal departing from new starts.

9.4.2 The Forward Planner Versus the Bidirectional Planner

When the performance of the two controllers is compared, the following differences become apparent. Backward planning is more “goal oriented” than forward planning. The forward planner spent nearly ten times more planning cycles than the bidirectional planner (52,923 vs. 5,397 cycles) to reach the goal for the first time in the environment. After the first success in the environment, the bidirectional planner maintained its superiority for the following trials

(Figure 9.5). This difference was caused by the fact that the forward planner took several cycles to find the goal for the first time *when planning* (i.e. while generating the simulated walks): 18,892 planning cycles on the average over the 10 simulations. In comparison the backward planner was particularly efficient: in 6 simulations out of 10 it reached the goal *in the environment* without having ever reached it while planning. This happened because the evaluations started to be updated immediately when the controller started to plan since the goal was immediately "found". In these regards it can be said that the bidirectional planner implements a better exploration of the state space.

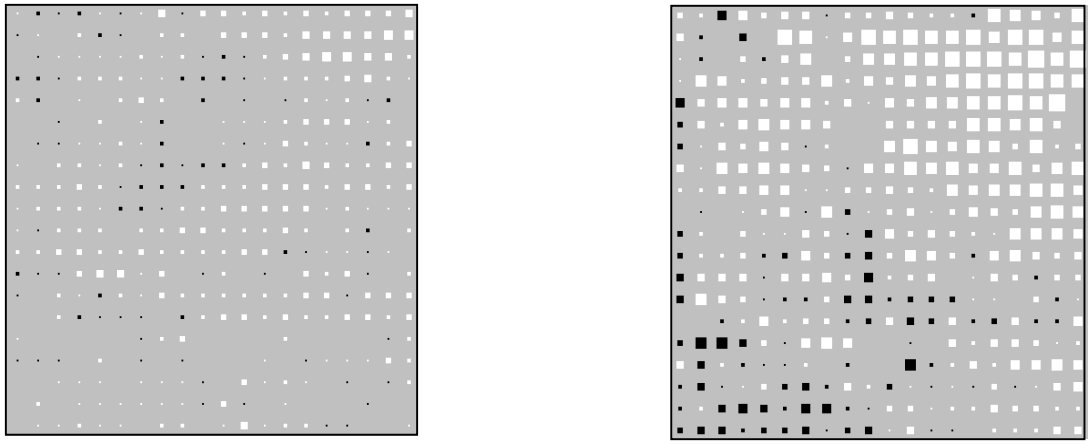


Figure 9.7: Evaluations yielded by the controller when set at 20×20 different positions of the arena. The area of a white square (positive evaluations) and a black square (negative evaluations) is proportional to the absolute evaluation yielded at that position. The evaluations have been recorded at the cycle after the first simulated reaching of the goal in the case of forward planning (left), and after 18,892 cycles in the case of the backward controller (right). 18,892 is the average number of cycles that *forward* planning took for reaching the goal for the first time while planning.

This efficiency in exploration leads to a faster propagation of values and updating of the policy. Figure 9.7 shows the evaluations yielded by the two controllers in 20×20 positions of the arena after some cycles of action and planning. The bidirectional planner yields evaluations much closer to the optimal ones than forward planning (recall that the optimal evaluations are equal to γ to the power of the number of steps to the goal). This exploration efficiency should be compared with that of other controllers that implement other forms of "undirected" and "directed" exploration (Thrun, 1992).

Backward planning is also more effective than forward planning in propagating the evaluations. Direct observation of the dynamics of the graph of Figure 9.7 drawn while the simulation was running, showed that at the beginning of the simulations with the forward planner the evaluations fell again to 0, as they were at the beginning of the simulation, between a (simulated) success and the next one. This happened because in the absence of positive rewards, forward exploration brought the evaluations toward 0. In fact on the average the decay coefficient γ lowers the "targets" of the evaluations updated. On the contrary, in the case of the bidirectional planner positive evaluations were continuously "injected" into the graph starting from the goal. In fact the evaluation of each state was updated backward from the goal, so each state's evaluation was updated on the basis of the goal or on the basis of the evaluation of a state that had just been updated. As a consequence the evaluation gradient field approached its final shape quicker.

9.5 Limitations of the Neural Bidirectional Planner

The bidirectional planner has three drawbacks. The first one, shared with the forward planner, is that it relies on the assumption that the model of the environment is enough accurate when planning starts. Here this assumption is fulfilled since a long training of the predictors is accomplished before planning starts, and is enhanced by the task in hand. However, it cannot be guaranteed in general for all other possible situations and for all other task domains (cf. s. 8.5).

The second drawback concerns the backward planning process of the controller. The generation of backward simulated walks starting from the goal on the basis of the back-actor and back-predictor may not be possible with some problem domains different from navigation.

The third drawback is that the bidirectional planner has a quite complex architecture compared to the forward planner, because it needs the back-actor and back-predictor to generate simulated walks backward from the goal.

9.6 A New "Goal Oriented Forward Planner" (Not Implemented)

This section proposes a neural planner that is only slightly more complex than the forward planner, but that might have interesting properties in common with the bidirectional planner. The key idea of this planner is that one way to exploit the knowledge of the goal without doing backward search, is to use the goal state to update the evaluation of the goal state itself. Recall that within the forward planner presented here when the goal is reached a new simulated walk starts. No "succeeding state's evaluation" is available to update the goal state's evaluation on the basis of the TD-Error formula. The consequence is that the evaluations of all states are 0 and all the backups have no effect until the goal is reached. The planner proposed here updates the evaluation of the goal state toward 1 *each time that the forward simulated walks fails* to reach the goal. This "goal oriented forward planner" should have several strengths in comparison to the forward planner:

- The evaluation of the goal state would become close to 1.
- Because of the generalisation properties of neural networks, all the states having some resemblance with the goal state would have a positive evaluation and the higher the resemblance the higher the evaluation. These evaluations would "guide" the forward searches towards the goal, similarly to what happens with the bidirectional planner.
- The evaluations of the goal and the states similar to it would be continuously renewed. This is important because the experiments have shown that the evaluations tend to decay if the goal is not reached continuously.
- The direct updating of the evaluations of the goal state and of the states similar to it, would progressively be eliminated when the simulated walks start to reach the goal regularly. This would replace the initial arbitrary evaluations of states similar to the goal with the more correct evaluations based on the actor's policy.

This planner could be particularly powerful for "assembly planning", i.e. planning for tasks where the goal state is made of an "assembly" of objects organised in a pattern, and where each object in the correct "position" can be considered as a sub-goal (Russell and Norvig, 1995). The planner would also incorporate the idea exploited by some landmark planners that build the evaluation gradient field on the basis of the similarity between the states and the goal (e.g. Schmajuk and Blair, 1993). Unfortunately this planner has been envisaged at the end of the PhD research, when there was no time to implement and test it.

9.7 Conclusion

This chapter presented a new neural bidirectional planner. When it plans this planner is more focussed on the goal than the forward planner presented in chapter 8. In particular it does not rely on a random-walk search when the goal is pursued for the first time. In fact the bidirectional planner executes a sequence of explorations (of the model of the environment) that start both from the current state and from the goal. In this way backups are focussed on states around the area of the start/current position and around the area of the goal. As in the previous controller, during these explorations the state evaluations and the action policy are updated. Moreover, during the backward exploration the back-actor learns to select actions that bring quickly away from the goal when producing backward simulated walks.

The controller has been shown to have the same strengths of the controller proposed and implemented in chapter 8. These strengths are: (a) taskability; (b) improvement of performance when the goal is encountered several times; (c) skill transfer when the same goal is pursued from different starts.

The simulations have also shown that the controller converges faster than the forward controller of chapter 8 for two reasons: (a) it is more efficient in exploring the state space within the model of the environment; (b) it is quicker in propagating the evaluations backward from goal.

The bidirectional planner has some drawbacks. First, its functioning assumes the accurateness of the predictors. Second, to work it needs to generate backward walks, and this may not be possible in some problem domains. Third, it has a quite complex architecture and functioning. With regards to the latter point, a new “goal-oriented forward planner” has been proposed that has a complexity similar to the forward planner's one, but that might have some of the strengths of the bidirectional planner.

10 Neural Network Planners and Multi-Goal Tasks

10.1 Introduction: Neural Planners, Interference and Modularity

Problems Tackled. This chapter evaluates how the neural planners implemented in the previous chapters deal with the problems of generalisation, interference and modularity, introduced in chapter 7. To this purpose it compares the performance and behaviour of the reactive systems, forward planner, and bidirectional planner presented in the previous chapters by using the multi-goal scenario presented in chapter 7 as a test.

What is New and Overview. Chapter 8 and 9 have implemented two controllers developed within the framework of the Dyna architectures (Sutton, 1990). These controllers are capable of operating in "reactive mode" or "planning mode". While planning the controllers execute a sequence of forward "explorations" from the current state (forward planner) or both forward from the current state and backward from the goal (bidirectional planner) within the model of the environment. During these explorations the state evaluations and the action policy are updated. The action probabilities are used to build a measure of the controller's "confidence" in the policy, and to switch between acting and planning mode.

Chapter 7 has implemented a modularised version of the basic neural-network actor-critic architecture capable of coping with asynchronous multi-goal problems. The idea was to use a modularised neural-network model in order to; (a) exploit generalisation; (b) avoid interference between input-output associations and problems that did not share common structure.

The novelty of this chapter (cf. Baldassarre, 2001c) is that the planning controllers showed in chapter 8 and 9 are integrated with the modular architecture proposed and implemented in chapter 7, originating two planning modular neural-network controllers. The performance of these two controllers and reinforcement learning is compared using the asynchronous multiple-goal task proposed in chapter 7. The comparison aims at verifying if the results obtained with the controllers engaged in a single goal task still hold for a multiple-goal task. These results showed that: (a) planning allowed the controller to reach the goal with improved efficiency in comparison to reinforcement learning the very first time the goal was pursued; (b) both planning controllers improved their performance when the goal was encountered several times; (c) the bidirectional planner outperformed the forward planner in terms of planning and acting cycles needed to achieve the goals, thanks to a more efficient exploration policy and a faster propagation of evaluations.

The test run in this chapter could appear unnecessary. In fact if the modular reinforcement-learning architecture shown in chapter 7 is capable of dealing with the multi-goal task, and the planning controllers introduced in chapter 8 and 9 work well independently of the underlying reinforcement-learning architecture used, then a controller based on both should not have problems with multi-goal tasks. This is not the case: planning controllers could be affected more seriously by catastrophic interference than the corresponding reinforcement learning systems. The reason is that in the case of reinforcement learning, when a goal is reached a new goal is pursued. Instead, in the case of planning the controller focuses

on the same goal for a long time before pursuing a new one. For example, recall that the neural forward planner needed to reach the same goal several times in planning mode before starting to act, reaching it in the environment, and finally pursuing another goal. This might exacerbate the effect of catastrophic interference because it lengthens the period of time elapsed between two experiences with the same goal.

Chapter's outline. S. 10.2 briefly describes the test used to compare the controllers. S. 10.3 explains how the planning controllers showed in chapter 8 and 9 have been integrated with the modular reactive system of chapter 7. S. 10.4 presents the results of the tests, and in particular shows that modularity also helps to deal with interference problems in the case of planning, and that the forward planner has problems dealing with multi-goals while the bidirectional planner does not. Finally s. 10.5 and s. 10.6 analyse the drawbacks of the modular planners and draw conclusions.

10.2 Scenario: Again the Asynchronous Multi-Goal Task

As mentioned, the task used to test the algorithms is the one illustrated in chapter 7 (cf. Figure 7.1 for the scenario used for this task). Recall that such task requires that the simulated robot pursue three goals asynchronously. At the beginning the simulated robot has to pursue one goal from a start position. Then each time the simulated robot reaches a goal, another goal randomly drawn from the three goals is assigned to it until the simulation ends.

10.3 Architectures and Algorithms

10.3.1 Modular Reactive Components

The components of the forward planner and the bidirectional planner are shown in Figure 10.1. The reinforcement learning components (evaluator, actor, TD-critic) of the controllers are the same as for the controller illustrated in chapter 7 (but now all learning rates have been set at 0.02).

The evaluator and the actor contain 6 experts each. Recall that the evaluator is based on a mixture of experts network, suitably modified to cope with the bootstrapping nature of the evaluator's learning process. The actor is based on a novel hierarchical architecture that repeats at two levels (gating network and single expert networks) the generation of the "merits" and the stochastic "winner-take-all" competition to select the experts and the actions. Notice that both the actor and the evaluator get as input not only the input (contrasts) about the current state, but also the (contrast) pattern encoding the goal. This allows the evaluator and the actor to yield state evaluations and actions that depend not only on the current state, but also on the goal.

As usual the matcher produces the internal reward, and functions both when learning and when planning, while the TD-Critic produces the learning signal e_t.

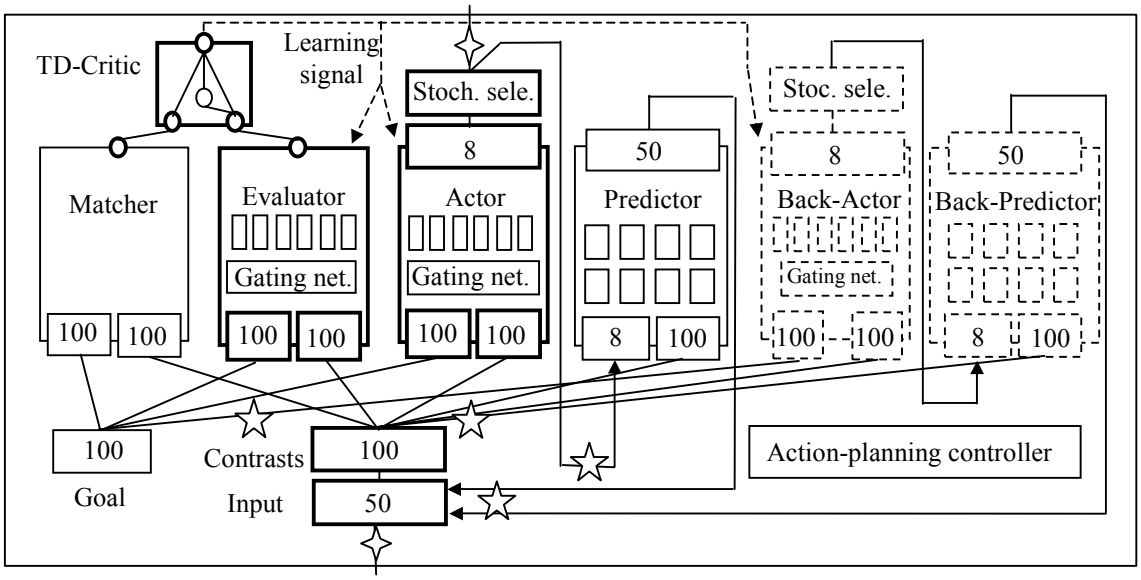


Figure 10.1: The controller of the simulated robot. Networks with a bold, thin and dashed border implement reinforcement learning, forward planning, and backward planning respectively. Arcs and arrows indicate forward and backward connections that "copy" a pattern from one layer to another. The four and five spike stars indicate the channels respectively set open and close by the action-planning controller when acting (vice versa when planning). Dashed arrays indicate the learning signal used to update the weights of the evaluator, actor and back-actor.

10.3.2 Neural Modular Forward Planner

The functioning of the forward planner is the same as in chapter 8 and 9. The fundamental component of this planner is the predictor (the "model of the environment"). An important thing to stress here is that even if here the planner has to pursue different goals, the predictor yields predictions on the basis of the (contrast) input $\mathbf{y}_t$ and the selected action a_t only. It does not need the information about the goal $\mathbf{x}_g$ ($\mathbf{y}_g$): the knowledge that it stores about the consequences of actions is independent of the particular goal pursued. As a consequence, while with multi-goal tasks the actor and the evaluator need information about the goal pursued to function, the predictor does not. Indeed, the predictor is identical in both the single (cf. s. 9.3.2) and multi-goal tasks. In chapter 8 and 9 we have seen that this fact is at the basis of the taskability of the planners proposed here.

The "action-planning controller" controls the flow of information among the different components of the whole controller when it is acting, planning forward (forward planner), and planning forward and backward (bidirectional planner, cf. s. 10.3.3). Its functioning is the same as the action-planning controller illustrated in s. 9.3.2 and the parameters of the algorithm are set at the same values used there: Decay = 0.000001, Gain = 0.01, MaxConfThresh = 0.15.

As in chapter 8 and 9, when the controller is forward planning or acting (this is true both for the forward planner and the bidirectional planner) the actor and evaluator function and learn in the same way they do when acting in the environment. We have already seen in s. 8.3.2 the events that take place in a forward planning cycle. Here these events are slightly different because the actor and the evaluator take into account the goal pattern $\mathbf{y}_g$. The new situation is summarised in Figure 10.2 (refer to s. 8.3.2 for an explanation).

```
01 Evaluator gets y_g, y_t and gives V'^π[y_t, y_g]
02 Actor gets y_g, y_t and gives m_t
03 Stochastic selector gets m_t and gives a_t
04 Matcher gets y_g, y_t, and gives r_t
05 TD-Critic gets V'^π[y_t-1, y_g], V'^π[y_t, y_g], r_t and gives e_t-1
06 Evaluator gets y_g, y_t-1, e_t-1 and learns
07 Actor gets y_g, y_t-1, m_t-1, a_t-1, e_t-1 and learns
08 IF(Planning)
09   Predictor gets y_t, a_t and gives x_t+1(y_t+1)
10 ELSE
11   System executes a_t in the environment
```

Figure 10.2: Pseudo-code for a cycle of forward planning.

10.3.3 Neural Modular Bidirectional Planner

In the case of the bidirectional planner the action-planning controller (illustrated in s. 9.3.2) generates simulated walks alternately forward from the current state and backward from the goal. Forward walks are executed as in forward planning. Backward walks are executed through the "back-actor" and "back-predictor".

```
01 Back-actor gets y_g, y_t and gives m_t-1
02 Back-stochastic selector gets m_t-1 and gives a_t-1
03 Back-predictor gets y_t, a_t-1 and gives x_t-1 (y_t-1)
04 Evaluator gets y_g, y_t-1 and gives V'^π[y_t-1, y_g]
05 Matcher gets y_g, y_t and gives r_t
06 TD-Critic gets V'^π[y_t-1, y_g], V'^π[y_t, y_g], r_t and gives e_t-1
07 Evaluator gets y_g, y_t-1, e_t-1, and learns
08 Back-actor gets y_g, y_t-1, a_t-1, e_t-1 and learns
09 Actor gets y_g, y_t-1 and gives m_t-1
10 Actor gets y_g, y_t-1, m_t-1 (actor), a_t-1 (back-actor), e_t-1, and
learns
```

Figure 10.3: Pseudo-code for a cycle of backward planning.

Recall from chapter 9 that the back-actor has the same architecture as the actor, and is used to generate actions for the simulated backward walks. Now, similarly to the actor, the back-actor has to take into account the goal and implement the association: $\mathbf{y}_t$, $\mathbf{y}_g \rightarrow a_{t-1}$. Notice that the back-actor is also trained while performing forward planning, so in the case of the bidirectional planning controller, a line of code has to be added to the algorithm of Figure 10.2:

```
Back-Actor gets y_g, y_t, m_t-1 (back-actor), a_t-1 (actor), e_t-1 and learns
```

The modular back-predictor is a network with the same architecture as the modular predictor, and functioning as the back-predictor illustrated in s. 9.3.3. Before the main tests, the back-predictor is trained to implement the association that yields the "expected previous input": $\mathbf{y}_t$, $a_{t-1} \rightarrow \mathbf{x}_{t-1}$.

During the backward walks also the evaluator and actor are trained using e. The events that take place in one cycle of backward planning are summarised in Figure 10.3. Notice that

unlike the controller of chapter 9, the functioning and learning of the evaluator and actor depend on the goal pattern $\mathbf{y}_g$, while the back-predictor functions exactly as in that chapter. In fact, like the predictor, the back-predictor stores knowledge that does not depend on the particular goal pursued.

With training the back-actor learns to yield backward walks that "escape away" from the goal in "straight" lines, hence creating a big area of positive evaluations around the goal. This area is "easily" found by the actor's forward walks that, as a consequence, progressively expand the area itself toward the start. At the same time the actor becomes competent in the whole area where positive evaluations diffuse.

10.4 Results and Interpretation

10.4.1 Modularity and Interference

The reinforcement learning system, the forward planner, and the bidirectional planner were tested with the multi-goal task. 10 simulation were run with different random seeds for each controller, each for a sequence of 2000 achievements of the goals. For all the three controllers, 7 out of 10 runs were successful, i.e. the system converged to a quite efficient path from the start to the goal (cf. Figure 10.6). In these successful runs the evaluator used three different experts to encode three different evaluation gradient fields corresponding to the three goals (more precisely: in each position of the arena, and for each goal, the evaluator had a probability above 99% of selecting the same expert). Figure 10.4 shows the gradient fields relative to the three goals in one of the successful simulations.

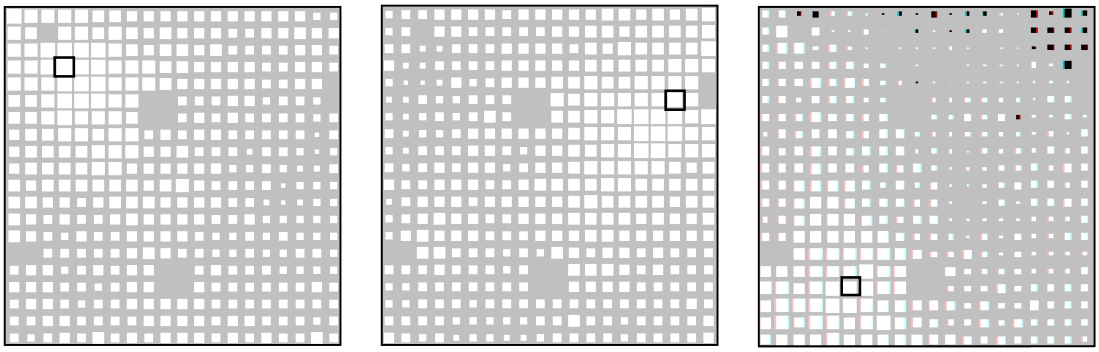


Figure 10.4: Evaluation gradient fields for two of the three goals (north-west, east, and south-west goals). For each goal the robot was set at 20×20 different positions of the arena, and the evaluator's output for that position was measured. Each cell of each graph is drawn in a position corresponding to the position in the arena where the evaluation was measured. The area of each cell is proportional to the evaluation. White cells represent positive evaluations and black cells represent negative evaluations. Cells with a bold border mark the goals' positions.

The 3 runs that failed (each controller did the same) did so because the evaluator employed the same expert to yield the evaluations related to two different goals. As a consequence, the evaluation gradient field had two peaks, the actor was trained to go to both peaks, and the resulting behaviour was dithering. This shows that in the task that is considered here the specialisation of the evaluator's experts for the different goals is crucial for the

correct functioning of the architecture (cf. chapter 7 on this. In this chapter a parameter search was done to avoid this problem).

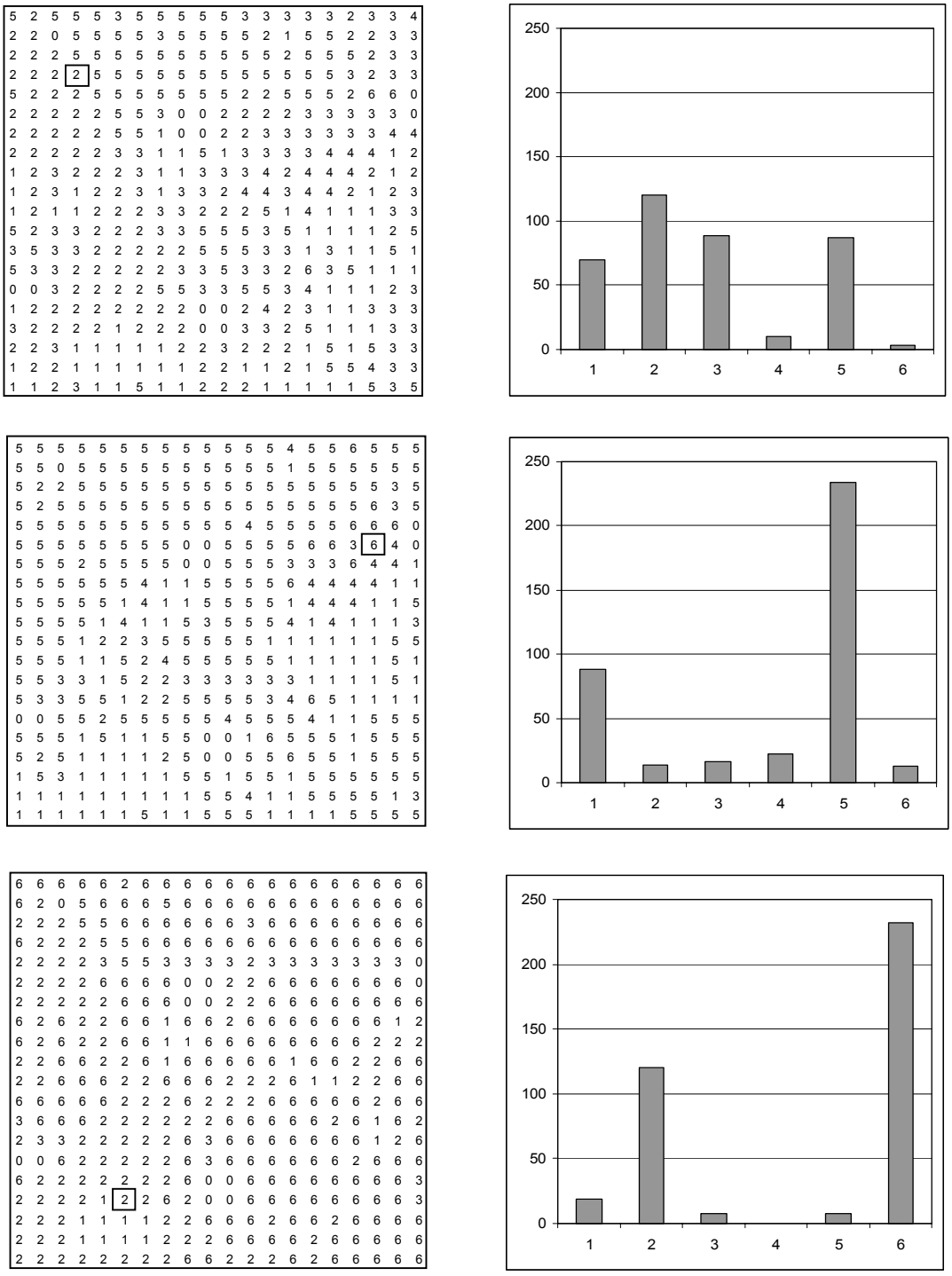


Figure 10.5: Left (grids of numbers): actor's expert with maximum probability of use in 20×20 positions of the arena in correspondence to the three goals. The position of the goal is marked with a square. Each number indicates the expert with the highest probability of being selected, in the position occupied by the number itself. Right: the histograms summarise the frequencies of use (y-axis) of the 6 experts (x-axis) in the whole arena.

In the 7 successful simulations, the actor learned to reach the goals in few steps from any point of the arena (see Figure 10.6, explained later). The function of the actor does not seem to require a precise specialisation as for the evaluator: more than one expert is used to achieve one goal, and the same expert is used to achieve many goals. Figure 10.5 shows the actor's expert that has the highest probability of being selected in 20×20 positions of the arena for the three goals. Clearly the actor uses different experts to handle different areas of the arena for the same goal (cf. chapter 7).

10.4.2 Taskability

For each simulation the number of *actions* per "success" (achievement of the assigned goal) was measured for each goal reached and then plotted against the cumulated number of successes. Figure 10.6 shows this measure for the three controllers (averaged over the 7 successful random seeds; a forward moving average of 20 steps has also been used to smooth the curves. Recall from s. 7.2, that the optimal path to the goals, not considering noise and obstacles, is about 10 steps long). The results confirm the results previously obtained with single goal tasks (cf. chapter 8 and 9).

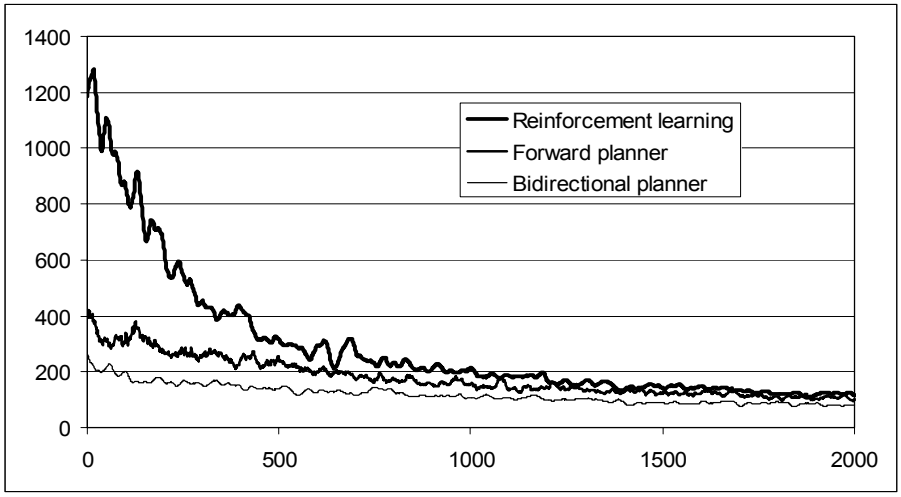


Figure 10.6: Performance of the three controllers (averaged over 7 random seeds per controller; smoothed with a 20-step moving average). Y-axis: number of actions per success. X-axis: successes.

The comparison between the performance of planning vs. reinforcement learning shows that planning allows the controller to reach the goals with improved efficiency *from the very first time* each goal is pursued (cf. Table 10.1): reinforcement learning takes 719 actions on average, the forward planner about 286 actions and the bidirectional planner about 199 actions. The situation is similar for the second goal pursued (see the explanation below for the reason why the performance with the second goal is worse than the one with the first goal). These results show that the planners are taskable (cf. s. 5.1.3).

10.4.3 From Planning To Reaction

Figure 10.6 shows another interesting result. Pursuing a goal several times also improves the performance of the controller in the case of planning (cf. s. 8.4.2). This happens because the information gathered with planning and with direct experience is merged appropriately and

incrementally in the weights of the evaluator and actor. This is a typical strength of the Dyna architectures (Sutton, 1990).

	Reinforcement learning	Forward planning	Bidirectional planning
1st goal	719	286	199
2nd goal	1407	420	297

Table 10.1: Number of actions taken by the three controllers to reach the first and second goal the first time they are assigned to them (averaged over *20* simulations run with different random seeds).

10.4.4 The Forward Planner Versus the Bidirectional Planner

As regards the comparison between the forward planner and the bidirectional planner, Figure 10.6 shows that before convergence the bidirectional planner outperforms the forward planner in terms of number of actions taken to reach the goals. One reason for this, probably of minor importance, is that the forward planner spends more cycles planning than the bidirectional planner (see below). In fact recall that while planning some actions are executed to avoid that the simulated robot gets stuck in situations where it does not succeed in becoming "confident" enough (cf. the algorithm illustrated in s. 9.3.2, Figure 8.3). These actions are in addition to the actions that are executed when the simulated robot becomes enough confident.

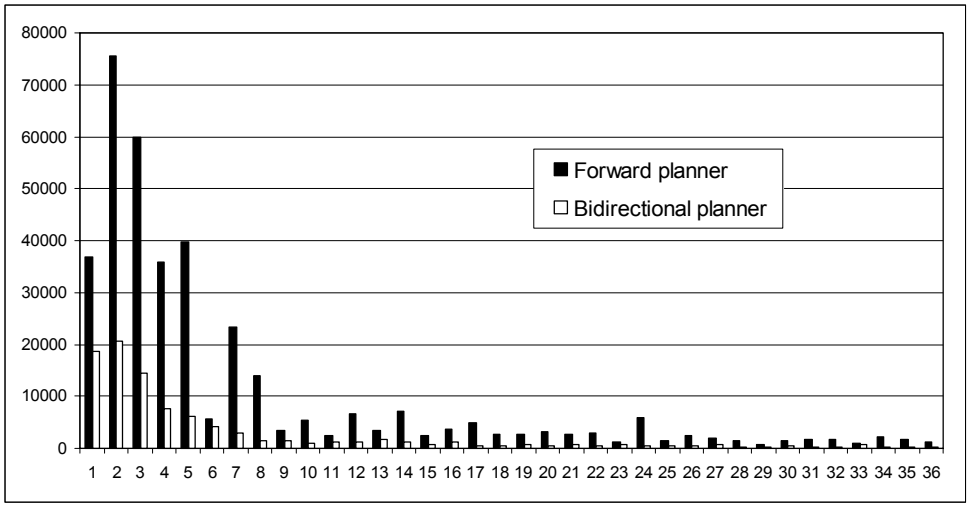


Figure 10.7: Cycles spent planning by the forward planner and bidirectional planner (y-axis) to achieve the first 36 goals (x-axis). Average over *24* simulations run with different random seeds.

Another reason is that back planning focuses exploration and learning around the goal. Direct observation of the behaviour shows that this area, where the simulated robot has to move to the specific position corresponding to the goal, is a particularly difficult part of the task: the simulated robot engages in a kind of random walk around the goal area searching for the goal. The reason seems to be that when the simulated robot is very far from the goal,

about 50% of the moves take it towards the goal, while when very close to the goal only 12.5% of moves (1 out of 8) takes it towards the goal. So when the controller is enough confident at the current state and starts to act, its actual competence for the area near the goal is higher in the case of the bidirectional planner than in the case of the forward planner. Further investigation should verify this explanation and check if this result holds for problem domains different from navigation.

Figure 10.7 shows the number of *planning cycles* per success taken by the two planning controllers for the first 36 goals reached (average over 24 successful simulations out of 32 run with each controller with different random seeds: 8 runs out of 32 failed). Many runs have been executed to obtain more reliable data (and the quite regular graph shown in Figure 10.7). An important result is that the planning cycles fall close to 0 when the controller experiences the same goals several times. The explanation of this is that when the same goal is encountered several times, the "confidence" associated with it increases over the confidence threshold, so that planning is no longer required and the goal is achieved reactively.

Figure 10.7 also shows that the bidirectional planner outperforms the forward planner in terms of number of cycles spent planning before reaching the goals. This happens for three reasons (cf. s. 0). The first reason depends on the way the two algorithms explore the model of the environment. The forward planner spends a lot of time searching for the goal unsuccessfully given that the goal is searched by a random walk, while the bidirectional planner "finds" the goal from the very first cycles of planning. So, in contrast to what happens for the bidirectional planner, the forward planner wastes a lot of planning cycles before starting to update the evaluations and, consequently, to update the action policy.

The second reason is that the bidirectional planner is more efficient than the forward planner in propagating the evaluations backward from the goal to the other states. In fact it updates the evaluation of each state on the basis of the evaluation of a state that has just been updated (cf. Lin, 1992, and Thrun, 1992).

The third reason is very important because it involves the different functioning of the two controllers with multi-goal tasks, and a problem caused by the generalisation property of neural networks. After enough confidence is attained and the first goal is reached, the action probabilities are quite biased in favour of that goal. When the goal changes, *half* actor's input pattern (the part $\mathbf{y}_g$ that encodes the goal) is changed for each state visited (cf. Figure 10.1). Given that the experts are not yet specialised for the different goals and that in general the actor is responding to the second goal mostly with the same weights used for the first goal, the probabilities are still quite biased in favour of the first goal. This implies that the random walk used to explore the model of the environment is biased away from the second goal. Figure 10.7 shows that the forward planner spends nearly a double number of cycles planning to reach the second goal than to reach the first goal, while the bidirectional planner only spends few cycles more.

Simulations not reported here show that if the level of the confidence threshold is increased so that the number of cycles spent planning for the first goal increases before achieving enough confidence, this drawback of the forward planner gets worse. In fact planning searches around the area of the first goal, and never reaches the area of the second goal. In contrast, bidirectional planning forces the search around the newly assigned goals. This leads the evaluations, and hence the action probabilities, to change according to the new goal.

10.5 Limitations of the Modular Planners

These results are encouraging, but the planning controllers presented have also some drawbacks. The forward planner and even more the bidirectional planner have a quite complex architecture composed of inhomogeneous neural networks.

The functioning of both planners depends on the possibility of building a reliable model of the environment. When planning starts, the planners assume that the model of the world is sufficiently accurate, and that training the evaluator, actor and back-actor through it will necessarily improve their abilities to evaluate and act (cf. s. 8.5). This assumption is not true in general for all task domains.

Backward planning relies on the possibility of training the back-actor to "guess" what action could have led to a particular state. It is not clear if this can be done within problem domains different from navigation.

The experts of the modular evaluator fail to specialise in 8 simulations out of 32 run with different random seeds. This raises interference problems and impairs the policy learning (cf. s. 7.6 and 12.3 for a discussion and a possible solution to this problem).

10.6 Conclusion

This chapter has tested the two planning controllers presented in previous chapters with a simulated robot engaged in a new multi-goal stochastic shortest-path problem. In order to allow the architecture to cope with multi-goal tasks the evaluator, the actor and the back-actor components, previously implemented with monolithic neural networks, have been replaced by modular networks.

The results of the test have shown that the specialisation of the evaluator's experts (one for each goal) and the partial specialisation of the actor (one prevailing expert for each goal) allows the controllers to cope with multiple goals. The other results confirmed the results obtained with single goal tests. The planners showed an efficiency higher than that of the reinforcement learning controller from the very first time a goal was pursued. With successive experiences the performance further improved, and the planners became "confident" enough and did not need to plan anymore. The bidirectional planner outperformed the forward planner both in terms of actions and planning cycles needed to achieve the goals, thanks to its capacity to focus exploration around the start and the goal and to propagate the evaluations quickly. Moreover, focussing around the goal has been shown to be of crucial importance when dealing with multiple goals because it allows breaking a wrong bias of the random walk search towards previously pursued goals.

Notwithstanding these encouraging results, the planners are affected by some drawbacks such as the necessity to rely on a sufficiently accurate model of the environment (planning starts after suitably training the predictor and back-predictor), a high complexity of the overall architecture, and an imperfect functioning of the evaluator's gating network that prevents a correct specialisation of the neural "experts".

11 Coarse Planning

11.1 Introduction: Abstraction, Macro-actions and Coarse Planning

Problems Tackled. This chapter addresses the problem of how implementing abstract planning with neural network systems. The benefits of abstraction are well known in the classic artificial intelligence literature: fast exploration of alternatives, creation of large plans without incurring in combinatorial costs, possibility of planning on the basis of a model of the environment that ignores the details (cf. Sacerdoti, 1974, and several works in Allen et al., 1990). Is it possible to exploit some of these benefits within the neural planners implemented in this thesis? This chapter attempts to give a first answer to this question within the problem domain of navigation. It also shows that discounted reinforcement learning has some problems in handling the long periods of time typically involved with abstract planning.

Overview. In order to tackle the problem of abstraction with neural systems, the forward and backward neural planners introduced in chapter 8 and 9 have been used to carry out planning on the basis of "macro-actions", defined as sequences of identical "primitive-actions", e.g. "north-north-north-north". The macro-actions have first been used to train the predictor, and then have been used to execute planning. This form of planning, based on this special (simple) kind of macro-actions has been called "coarse-planning" for ease of reference. As for the other neural planners proposed in the previous chapters, the planning process has been used to improve the evaluations and the policy of the controller. Then the policy has been used to act at the level of primitive-actions.

A key issue of abstraction is the possibility of dealing with long periods of time. In chapter 6 it has been shown that discounted reinforcement learning may have some problems handling long periods of time because the optimal evaluations for states far from the goal are near zero. With the presence of noise, the gradient field over such states does not give enough information to create the policy. The fact that these problems have been encountered again while running the simulations of this chapter related to abstract planning, confirms their relevance. In the following sections first the effects of the problem on abstract planning are shown. Then a solution is proposed and implemented based on the use of different discount coefficients for planning and for acting. Unluckily, this solution is not fully satisfactory because it compromises the useful complementarity between planning and reactive behaviour of the controllers proposed here and highlighted in the previous chapters.

What is New and Related Literature. The idea of macro-actions made up by a certain number of actions of the same kind, and their use for abstract planning, is new (Baldassarre, 2001d). This idea is closer to the idea of "abstract operators" of classical artificial intelligence planning (cf. s. 2.4.3) than to the idea of options as sub-policy of the framework of reinforcement learning and options (Sutton et al., 1998; cf. s. 13.2.7). The issues related to the different discounted rates have been developed within the framework of reinforcement learning and options, but the analysis of the "discounted reinforcement learning delay problem" within abstract planning is new.

Lin (1993) presents a work related to macro-actions. This work studies a simulated robot that has to move from one room to another room of an office. The simulated robot is first trained to develop separate simple policies as "follow the wall", "enter the door", etc. on the basis of simple primitive-actions. Then all these building-block policies are treated as single macro-actions by a higher-level reinforcement learning procedure that is trained to trigger them in order to solve a path-finding problem. The results show the considerable advantages that this technique has versus learning the policy on the basis of the primitive-actions. Even if this work is different from the simulations presented in this chapter because it does not deal with planning, the idea of applying reinforcement learning to "hardwired" macro-actions (this chapter) or sub-policies (Lin, 1993) is common to the two.

Nehmzow et al. (1991) have proposed an experiment where a robot, following a wall through a pre-programmed behaviour, uses information about the commands sent to the motors and their duration to recognise locations in the environment through a self organising neural network (Kohonen, 1982). This is an interesting form of indirect abstraction with regards to the rich sensory information: it is very compact but still sufficient to accomplish the location recognition task. Tani and Nolfi (1999) use self-organising neural networks to abstract information with regards to perception. Interestingly, they also exploit signals repeated over long periods of time (in this case sensory signals) to enhance abstraction.

Before passing to consider the details of coarse planning, it is interesting to review the work of McGovern et al. (1997), because it presents some results that are useful in interpreting the results of this chapter. The work of McGovern et al. (1997) has been developed within the framework of reinforcement learning, options and macro-actions (cf. s. 13.2.7). It presents the result of an interesting empirical research directed to investigate the advantages and disadvantages of reinforcement learning controllers based on "macro-actions". The first result shown by the authors is that macro-actions influence the exploratory behaviour of the controller such that more relevant states are visited more often. The second result is that macro-actions allow the system to propagate rewards more rapidly.

Chapter's Outline. S. 11.2 presents the scenario of simulation and s. 11.3 illustrates the architecture and functioning of the "coarse planner". Then two groups of results are presented. The first group of results shows the functioning of coarse planning and the advantages that it produces vs. planning at a primitive level (s. 11.4.1 to s. 11.4.3). The second group of results shows how different discount coefficients, used for planning at a coarse level and for acting at a primitive level, affect the evaluations, the quality of action, and the speed of learning (s. 11.4.4). Finally s. 11.5 and s. 11.6 analyse the drawbacks of the coarse planner and draw conclusions.

11.2 Scenario of Simulations: A Simplified Navigation Task

The environment used in the simulations is the simple environment considered in s. 6.2 (cf. Figure 6.1). This scenario is reproduced in Figure 11.1 for convenience. This figure also shows the goal used in the simulations. This simple environment has been used to simplify the resulting gradient field and ease the interpretation of the results (see below).

The simulated robot is the usual one (cf. s. 6.2). The size of the steps of the simulated robot has been one of the parameters investigated with the simulations. The simulated robot's diameter has always been set at the same size of the step's length. Notice that while the simulated robot's step size is very important, the simulated robot's diameter has only graphical effects (however, the diameter of the robot affects the way one thinks about the relative size of the robot and the arena). Perception and action are affected by the usual noise.

The simulated robot's task is to reach the goal position from the start position and then from other positions chosen randomly.

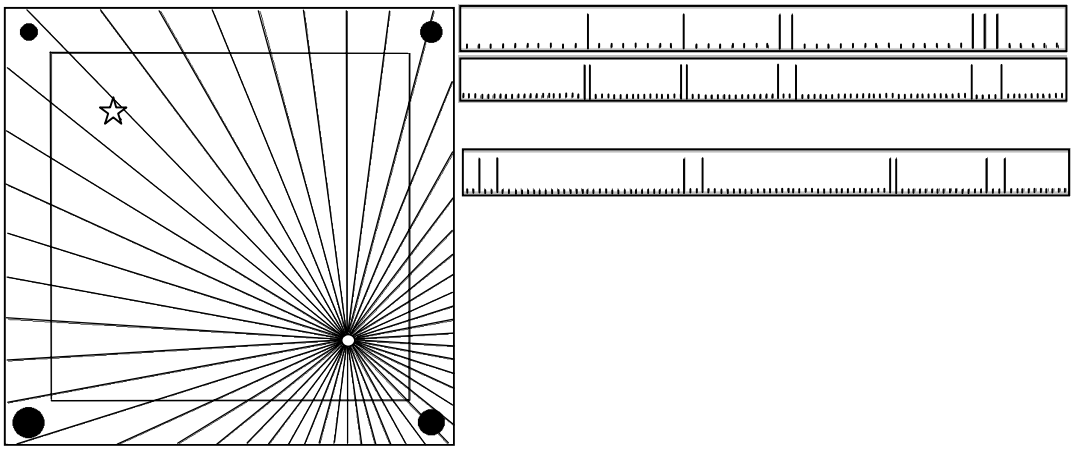


Figure 11.1: Left: the scenario of the simulations containing the goal (star), four landmarks (black circles), the scope of the simulated robot's 50 visual sensors (delimited by the rays), the simulated robot at the start position (white circle at origin of rays). Right, from top to bottom: the simulated robot's activation of the visual sensors at the start position (the position occupied by the robot in the graph on the left), its re-mapping into contrasts, and the goal (contrasts).

11.3 Architectures and Algorithms: Coarse Planning with Macro-actions

The controller used for the experiments of this chapter is the forward planner investigated in chapter 8 and reported in Figure 11.2 for convenience.

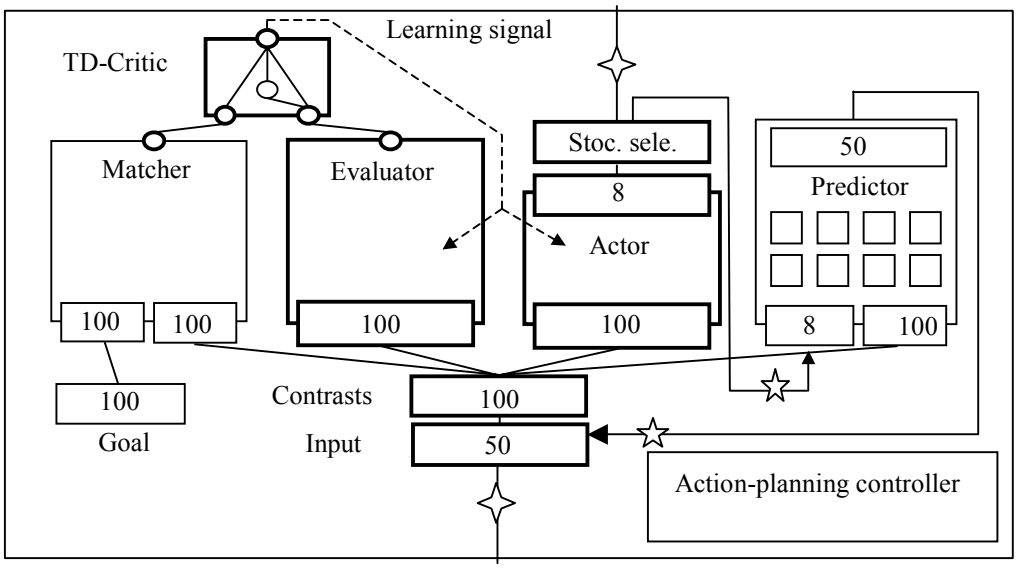


Figure 11.2: The controller of the simulated robot. Networks with a bold and thin border implement reinforcement learning and planning respectively. Arcs and arrows indicate forward and backward connections respectively. These "copy" a pattern from one layer of units to another. The four (five) spike stars indicate the channels set open (close) by the action-planning controller when acting (vice versa when planning). Dashed arrays indicate the learning signal used to update the weights of the evaluator and actor

The parameters are set at the same values used in chapter 8 with the exception of the learning rates of the actor and critic, set at 0.05 in the simulations of this chapter, the Decay and Gain parameters of the planning algorithm, set both at 0.00001, and the MaxConfThresh parameter, set at 0.15 (see below for the justification of these choices).

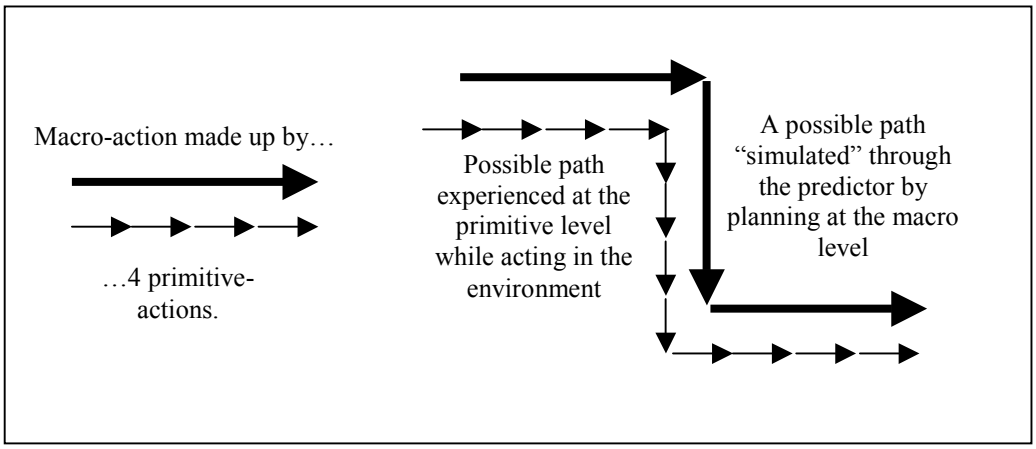


Figure 11.3: Left: The relationship between a macro-action and the primitive-actions that make it up. Right: A possible path simulated while planning and the path that would be followed while acting in the environment.

The key difference between the functioning of the planner used in this chapter and the functioning of the planner of chapter 8 is that the "granularity" of planning is coarser than the granularity of the actions' execution. In particular (cf. Figure 11.3):

- Planning at a coarse level is obtained by training the predictor in a way different from the one employed in chapter 8. In the latter case the predictor was trained to predict the input pattern following the execution of a single (primitive) action in correspondence to the current input pattern. Now the predictor is trained to predict the input pattern that follows the execution of a macro-action. A macro-action is made up by a sequence of a certain number of primitive-actions (2, 4, 10 in the simulations reported here). The primitive-actions that make up a macro-action are identical. An example of macro-action that is made up of 4 identical primitive-actions is this: "north-north-north-north".
- When the controller is planning and the actor selects an action (e.g. "north"), this action is "executed" within the predictor and its effect results in a macro-action's effect, i.e. in the prediction of the effects of a long movement. This happens because the predictor has been trained in terms of macro-actions.
- When the controller is acting in the environment, the actor selects an action (e.g. "north") but now the effects of this selection result in a small movement in the environment corresponding to the execution of the a primitive-action.

As we shall see, the experience accumulated by the evaluator and actor while planning at a coarse level can be used to act at a fine level. This is possible because of the generalisation property of neural networks and because of a property of navigation tasks (in particular navigation tasks in open spaces) for which the direction of optimal primitive-actions and optimal macro-actions often coincide (see below).

11.4 Results and Interpretation

11.4.1 Reinforcement Learning at a Coarse Level

The first simulation tested the performance of the controller running in reinforcement learning mode (the confidence threshold was fixed to a low value, 0, so the controller never planned). The step of the simulated robot was set at 0.025 (in all the previous chapters the step measured 0.05). The optimal path to the goal, not considering the negative effects of noise, is about 20 steps (= 0.5/0.025). The simulated robot was set at the start, and had to learn to reach the goal. When the simulated robot reached the goal it was set at a new random location in the arena, and had to reach the goal again from there. This was done until 60,000 cycles had been executed. Figure 11.4 shows the performance of the controller, measured as the number of cycles per success. This and all the graphs reported in this chapter refer to an average over 10 simulations run with different seeds of the random number generator. The performance improves from about 8000 to about 120. The graph also reports the performance, equal to 7712, of the simulated robot following a random walk (it was obtained setting the learning rates of the critic and actor to 0).

Other simulations were run to test the learning capabilities of the reactive components at different levels of the granularity *of actions execution* given the scenario's dimensions (1 by 1). This was done by changing the dimension of the simulated robot's step. In particular, in order to interpret the results of the simulations shown below, it was important to evaluate the learning capabilities of the reactive controller when its step was small in comparison with the arena/landmarks.

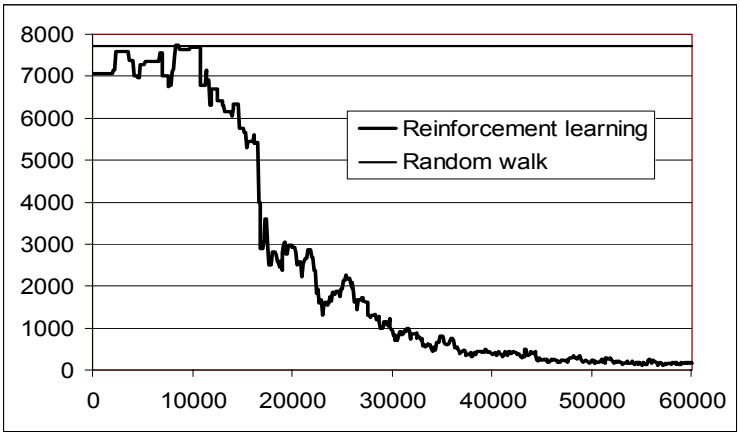


Figure 11.4: Performance of the controller in the reinforcement learning mode (y-axis) plotted against the actions executed (x-axis). Average over 10 runs of the simulation repeated with different random seeds.

The results show that if the step's size is reduced to 0.0125, i.e. it is divided by 2 with regards to the previous simulation (where it was 0.025), the reinforcement learning algorithm is not capable of learning anymore. The reason is that the emergent evaluations' gradient field is rugged and has several local peaks (see below for some examples). The consequence is that the actor tends to learn to take the robot to local peaks. The evaluations' gradient field is rugged because of the simplicity of the sensory system and the neural architecture used for the evaluator (cf. s. 6.4.2, 6.4.3, and 6.4.5). Analogous negative results are obtained with a step

smaller than 0.0125. With a step measuring 0.05 or 0.1 the controller learns properly. The reason why training is successful with long steps (0.025, 0.05, and 0.1) is that they allow the simulated robot to "jump" away from the local peaks and to reach positions with differences in the evaluations that on average tend to reflect the actual distances from the goal.

11.4.2 The Advantages of Coarse Planning

The next two simulations have been run to test the controller's planning capabilities when doing coarse planning. The predictor was trained before this test while the simulated robot was exploring the environment with a random walk lasting for 100,000 cycles. As mentioned the key point here was that the random walk was made up by *sequences of a given length of primitive-actions*. In particular 2 primitive-actions with the same direction (e.g. "north-north") were used in the first simulation, and 4 primitive-actions (e.g. "north-north-north-north") were used in the second simulation. Consequently, the total size of the macro-actions was 0.05 and 0.1 respectively for the two simulations. The predictor was trained to predict the consequences of these kinds of macro-actions; i.e. the teaching output was the one *at the end* of the execution of the 2 or 4 action sequences. In particular, when a macro-action was executed the predictor's expert of the corresponding primitive-action was trained. For example if a "north-north" macro-action was executed, the expert corresponding to the "north" primitive-action was trained. If during the execution of the 2 or 4 action sequences the simulated robot hit the edge of the arena, the input pattern was used as teaching output.

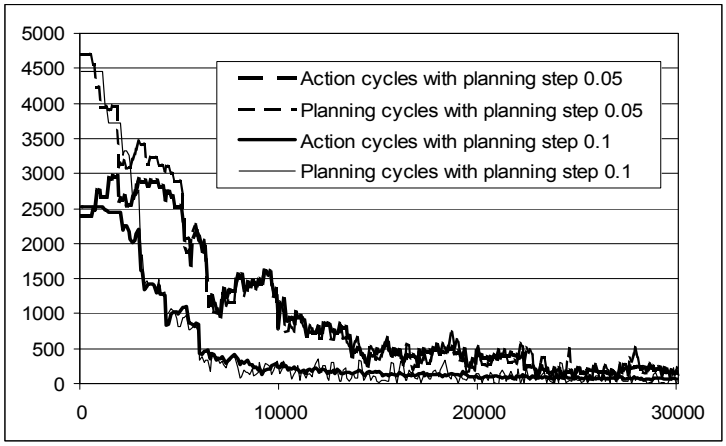


Figure 11.5: Performance of the coarse planner with two levels of coarseness. Y-axis: number of actions (or planning cycles) per success for the two levels of coarseness (2 and 4 primitive-actions). X-axis: cumulated actions. All the curves are averaged over 10 runs repeated with different random seeds.

Figure 11.5 shows the results of the two simulations, with the two different predictors (trained with 2 or 4 step macro-actions respectively) and run with the same modalities as the previous reinforcement learning simulation. Notice that, given the way the predictor was trained, planning took place at two different levels of coarseness in the two simulations (2 and 4 primitive-actions per macro-action). The graph reports both the number of actions (one per simulation cycle) per success, and the number of planning cycles (eventually many per simulation cycle) per success. Recall that the value of the Decay and Gain variables of the planning algorithm were set at the same value, 0.00001. This was done so that the number of action executions and planning cycles had the same average and the analysis of the results

was simpler. This is also the reason why the acting and planning plots overlap most of the time. The initial difference between them is caused by the fact that the initial value of the variable MaxConfThresh was set at 0.15 (slightly higher than the initial average actions' probability equal to 0.125) to allow the iterative deepening planning algorithm to reach a depth of about 50 before the simulated robot started to move. To maintain the consistency of the discount factor between the primitive and macro level (cf. McGovern, 1998, and s. 13.2.7) while planning a value of 0.9025 ($\approx 0.95^2$) and 0.8145 ($\approx 0.95^4$) was used respectively in the two simulations.

Why Coarse Planning Works. A first fact that is apparent from Figure 11.5 is that the policy learned while planning at a coarse level is appropriate for acting at a fine level (this is confirmed by the simulations shown below, where the controller updates the evaluator and actor's weights only while planning). This happens for the following reasons.

- In navigation tasks like the one shown here (especially if in open-space) the directions of the optimal macro-action and of the optimal primitive-action coincide. Here "optimal" is intended as the macro-action or primitive-action that take toward the goal following a direct path. As a consequence, if while planning the actor learns to select a particular macro-action (e.g. "north-north-north-north") at a particular position, the primitive-action with the same direction (e.g. "north") has a high chance of being adequate. Once the primitive-action is executed and a new position is reached, the actor can produce another selection that again is likely to be suitable both for the macro and primitive level. If at a given position the optimal macro-action and the primitive-action differ, the following time step the error will be corrected by the policy that is suitable to deal with the most likely outcomes of action execution.
- The use of neural networks allows the controller to generalise. This means that the controller is capable of yielding appropriate evaluations and action probabilities when encountering positions never met previously, on the basis of the exploration of similar positions while planning. This property of neural networks is also useful for planning and acting at the primitive level (cf. chapter 6), but now it is even more important. In fact with coarse planning the number of steps spent planning is less than with standard planning and so is the number of states visited (in simulation mode) before acting. The simulated experience accumulated by visiting this limited number of states is sufficient because it is extended to non-visited but similar states by generalisation.
- Notice that although the area that covers the positions recognised as goal could be small, this does not undermine coarse planning thanks to actions' noise. In fact even if planning takes place at a coarse level, it can still reach all the points in the arena because of the actions' noise. In the absence of noise, the simulated robot would visit few points on a fixed grid, eventually missing the goal area.

The Strength of Coarse Planning. The second fact that emerges from Figure 11.5 is that planning at the higher level of coarseness causes an improvement of the performance. There happens for two reasons.

First, exploration with coarse planning covers the whole arena in less time, i.e. less steps are taken to reach the goal in planning mode. This is shown in Figure 11.6. Here the simulated robot, set at the start, executes 1000 actions selected randomly. The experiment is repeated three times with different size of the step: 0.025, 0.05, and 0.1. From the graphs it is apparent that a bigger step improves the exploration of the environment.

Second, the evaluations are updated more quickly. In fact during coarse planning the evaluations of states are updated on the basis of the evaluation of a state 2 or 4 steps away.

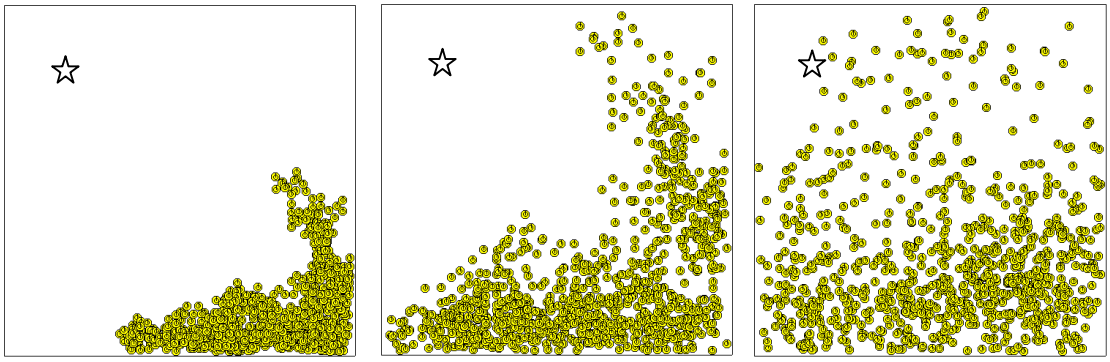


Figure 11.6: Positions occupied by the simulated robot moving 1000 times with random actions from the start. The steps measure 0.025, 0.05, and 0.1 respectively for the three graphs. The star indicates the position of the goal.

In order to compare them, Figure 11.7 presents the plots of the reinforcement learning and planning simulations (only plot of action cycles) in the same graph. The plots have been smoothed with a moving average over 3000 cycles to ease the comparison. Planning clearly shows a better performance. Even in terms of total number of cycles (planning cycles plus action cycles) planning shows its superiority. This can be seen by multiplying by 2 the plots of planning on the vertical axes (recall that given the settings of the parameters, the number of cycles spent planning is roughly the same as the number of cycles spent acting).

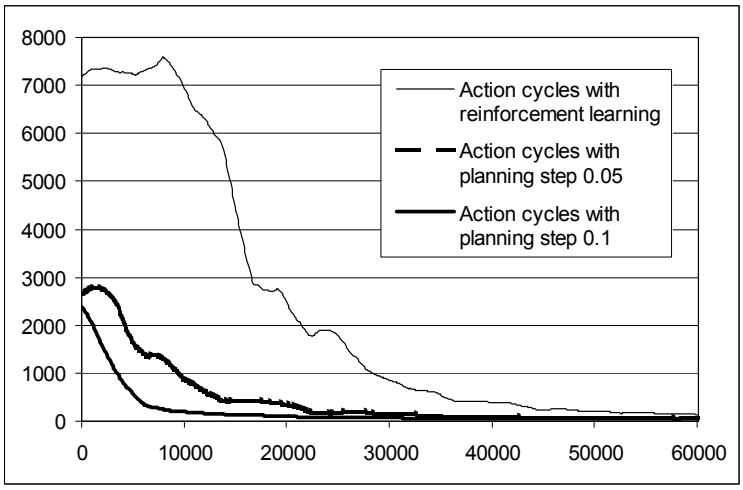


Figure 11.7: Comparison between reinforcement learning and planning at two levels of coarseness. Y-axis: number of actions per success, smoothed with a moving average over 3000 cycles. All the curves are averaged over 10 runs repeated with different random seeds. X-axis: cumulated actions.

11.4.3 Predicting at a Coarse Level

To complete the picture regarding prediction, other simulations have attempted to train the predictor with steps smaller than 0.05 (0.025, 0.0125, etc.) and have shown that this is not possible. In fact in these cases the predictor predicts a pattern similar to the input pattern. This

is because after each action the sensory input changes only slightly. Instead the predictor works well with bigger macro-actions made up of 8 primitive-actions in sequence. Table 11.1 summarises the outcome of the training of the predictor at different levels of planning coarseness, and reports the error (measured as $(\Sigma_i[d_i - o_i]^2 /n)^{1/2}$, where d_i and o_i are the desired and effective activation of the n output units) averaged over the last 5000 cycles obtained at the end of training.

Level of coarseness; size of macro-action	Error of prediction after 100000 training cycles	Outcome of training of predictor
0.5 ; 0.0125	0.1903	failure
1 ; 0.025	0.1873	failure
2 ; 0.05	0.1967	success
4 ; 0.1	0.2075	success
8 ; 0.2	0.2350	success

Table 11.1: Outcome of training of the predictor at different levels of coarseness.

11.4.4 Coarse Planning, Discount Coefficient and Time Limitations of Reinforcement Learning

Now the results of a different set of simulations involving planning are shown. They are directed to understanding the relationship between level of coarseness, discount coefficient and learning speed. In these simulations the controller does not learn while acting: the actor and evaluator's weights are updated only while (coarse) planning. This allowed the exploration of the effects of different discount coefficients during coarse planning, without having problems of consistency with the discount coefficient used at the acting level. The results suggest the idea that coarse planning should be independent of the primitive-action level in terms of discount coefficient used: this should be tuned with the coarseness of planning, not with the coarseness of action.

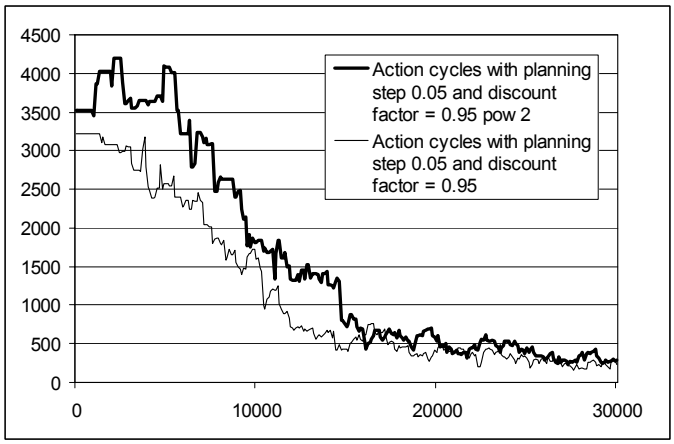


Figure 11.8: Planning with different discount coefficients and planning coarseness corresponding to a 0.05 step. Y-axis: number of actions per success. The curves are averaged over 10 runs repeated with different random seeds. X-axis: cumulated actions.

Figure 11.8 and Figure 11.9 show some simulations that compare, respectively for the 0.05 and 0.1 step levels of coarseness of the planning cases seen previously, the performance

of the system with a discount coefficient reduced according to the size of the macro-actions (0.95^2 and 0.95^4 respectively, in accordance with the theory of options and macro-actions, cf. s. 13.2.7) versus the performance with a discount coefficient of 0.95 (independent of the coarseness). It is apparent from these graphs that a discount factor suited to the level of coarseness used brings an improvement in the performance.

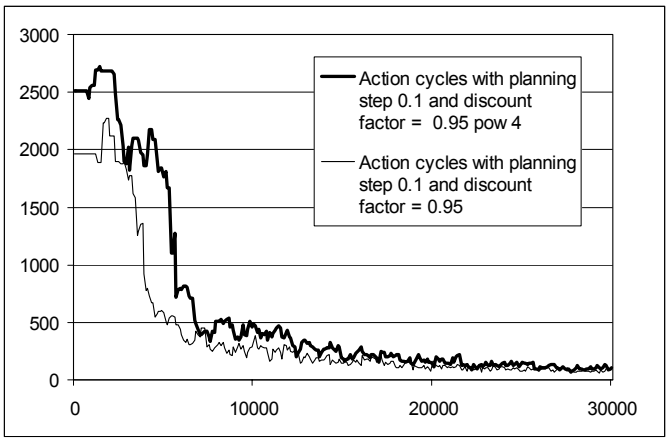


Figure 11.9: Planning with different discount coefficients and planning coarseness with step 0.1. Y-axis: number of actions per success. The curves are averaged over 10 runs repeated with different random seeds. X-axis: cumulated actions.

A further simulation with coarseness of 8 steps per macro-actions (with a macro-action measuring 0.2) and a discount coefficient set at 0.95, showed that this principle has an upper limit: the system learned but was quite unstable. The reason for this instability can be understood observing Figure 11.10. This figure reports the gradient fields of the evaluations obtained in the two previous simulations (2 and 4 step macro-actions and a discount coefficient set at 0.95) and the one being discussed here (8 step macro-action). The second graph reported in the figure shows that almost all evaluations are near the maximum level (i.e. 1) and have a very shallow gradient, while the third graph shows high evaluations without a defined gradient.

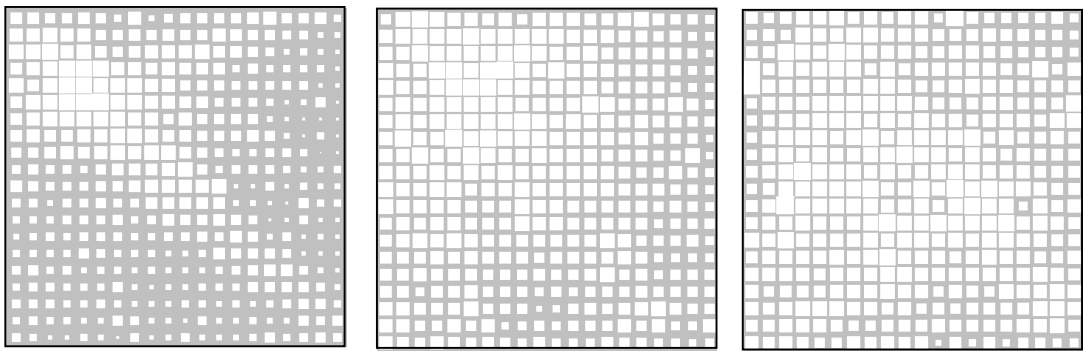


Figure 11.10: From left to right: gradient field of evaluations with planning at three coarseness levels (macro-actions of 2, 4, and 8 steps). The simulated robot has been set at 20×20 different positions of the arena, and the evaluation yielded by the evaluator for each of them has been measured. The size of the white cells is proportional to the evaluation given. The big white cells scattered irregularly in the graphs are caused by temporary noise of sensors.

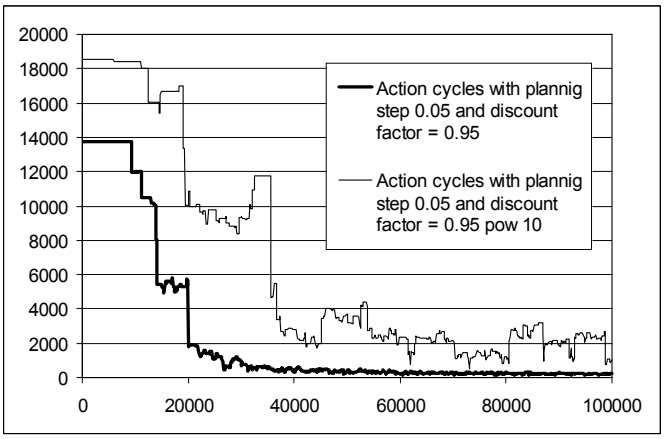


Figure 11.11: Performance with primitive step 0.005 and coarse planning step 0.05 (10 primitive-actions) with two different discount coefficients (0.95 and 0.95^{10}). Y-axis: number of actions per success. The curves are averaged over 10 runs repeated with different random seeds. X-axis: cumulated actions.

In order to further support the idea that the discount coefficient should be tuned with the coarseness of planning, another simulation has been run where the primitive-actions' size and simulated robot's diameter were 0.005, and the macro-action used for coarse planning was made of 10 primitive-actions (a total size of 0.05). The simulations were run two times with different discount coefficients: 0.95 and $0.5987 \approx 0.95^{10}$ (in accordance with the primitive level action) respectively. The results (Figure 11.11) show that the first discount coefficient results in a good performance, while the second discount coefficient makes learning and behaviour quite "unstable" (cf. Figure 11.12 shows the path followed by the simulated robot at the end of training in the two situations).

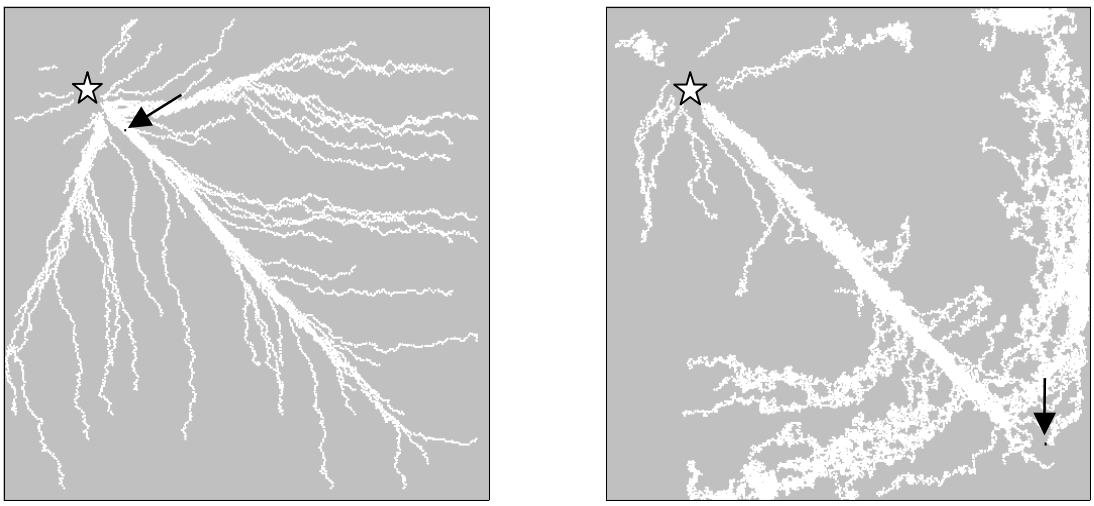


Figure 11.12: Paths followed by the simulated robot with coarse planning and different discount coefficients (left graph: 0.95; right graph: 0.95^{10}). The stars indicate the position of the goal. The arrows indicate the tiny simulated robot in the arenas: the size of the simulated robot gives an idea of the dimension of the robot's step with regards to the arena.

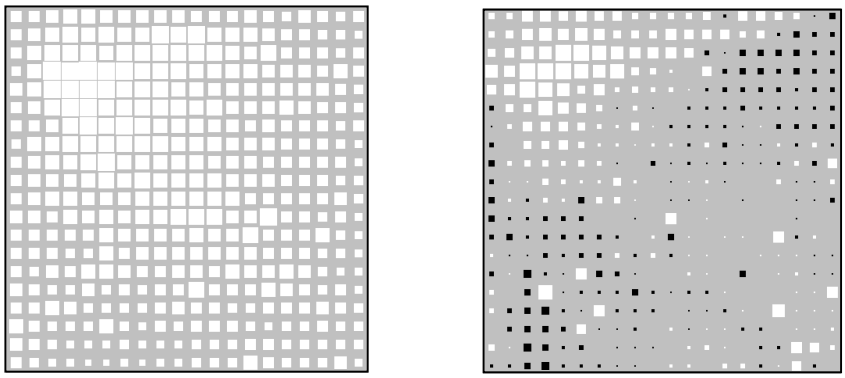


Figure 11.13: Evaluation gradient field with primitive-action steps measuring 0.005, coarse planning of 10 steps, and different discount coefficients (left graph: 0.95; right graph: 0.95^{10}). The black cells indicate negative evaluations. Big white cells scattered in the graph are caused by noise of the sensors.

The reason for this instability is that the primitive-actions have a very fine granularity, so the areas far away from the goal should receive a very small evaluation because of the decay coefficient (100 steps away from the goal, the evaluation should be equal to $0.95^{100} \approx 0.005920$). When evaluations are very close to 0 their differences become extremely small. Now, the TD-error, on the basis of which the evaluator and actor are trained, is computed as the difference between the evaluations of two contiguous positions. As a consequence if noise affects the sensors and function approximation is used to approximate the evaluation and policy functions, the TD-error signal becomes completely overwhelmed by noise and learning cannot take place in a proper manner (cf. also s. 6.5 on this). Figure 11.13 demonstrates that this is the case for the two simulations we are examining. The figure shows the evaluation gradient fields relative to the two simulations. In the case of a discount coefficient equal to 0.95^{10} the nature of the gradient appears to be very irregular and close to 0 for positions far from the goal. This disrupts the TD-error learning signal (this should also be the explanation for the instability observed in the simulation illustrated in Figure 11.11). Overall these simulations suggest that a discount coefficient suited to the level of coarseness of actions improves the functioning of reinforcement learning algorithms.

11.5 Limitations of the Neural Coarse Planner

Coarse planning has some limitations. The way coarse planning has been implemented here can be applied only to problems domains with certain properties (cf. the conclusion for an attempt to define these properties).

As we have seen, in open-space navigation tasks the optimal primitive-action and the optimal macro-action corresponding to a given position share a common direction most of the times. This assumption is not true in general. For example it would be interesting to analyse to which extent this assumption holds within a more complex navigation tasks, e.g. one containing obstacles.

When coarse planning uses a discount coefficient different from the one used at the action level, the evaluations developed at the two levels are different, so it is not possible, within the framework about options and macro-actions presented in s. 13.2.7, to use them in synergy. This weakens the opportunity to exploit the advantages rendered by the use of different discount rates at different levels of coarseness.

11.6 Conclusion

Coarse Planning. This chapter has presented an empirical investigation of the problem of planning with macro-actions and acting with primitive-actions. A significant fact apparent from the simulations is that planning at a level coarser than the level of action execution is possible and useful. The main problem in doing so is to find a way to allow the two levels to interface. Given a plan developed at a coarse level, how can its details be specified so as to implement it at a fine level? Similarly, given some experience accumulated by acting at the fine level, can it be exploited to plan at a coarse level?

Future research will focus on the possibility of using the same controller with navigation tasks with obstacles, and on the possibility and utility of using coarse planning for other "linear" problem domains.

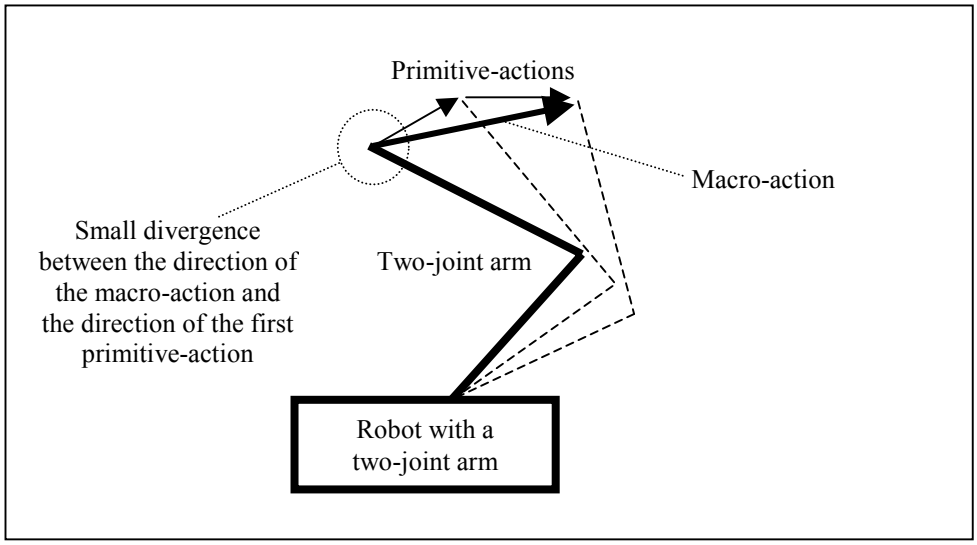


Figure 11.14: An example of another problem domain where the solution of macro-actions could be adopted: a robot with a two-joint arm. In this case the divergence of the direction of a macro-action (made of two primitive-actions) and the direction of its first component primitive-action is small and would produce acceptable errors.

The present chapter has proposed and implemented a solution to these problems for a particular problem domain, that of navigation in open areas. This solution is based on macro-actions made up by small sequences of primitive-actions of the same kind (e.g. "north-north-north-north"). These macro-actions are used to train the predictor and for planning, while the action execution takes place using primitive-actions. The harmony between the macro-level of planning and the primitive level of acting is guaranteed by three properties of the problem domain and the controller used: (a) the (likely) coincidence of the direction of the optimal macro-action and the direction of the optimal primitive-action; (b) the generalisation property of neural networks used by the controller; (c) the noise of action execution that guarantees the exploration of the whole state space.

It is important to understand whether other problem domains share these properties. Property (b) is guaranteed by the nature of the controller. Property (c) is guaranteed either by noise or by using actions that bring to reach all the states. Property (a) is the most delicate. At a first analysis it appears that a similar property is guaranteed by all problem domains where the effects of the execution of a macro-action are linear (or close to linear) with regard to the

effects of the single component primitive-actions. For example this could be the case of a two-joint robot arm if the macro-action is made of few primitive-actions (cf. Figure 11.14).

What should be done when we have problem domains that do not allow us to build macro-actions by simply assembling sequences of primitive-actions of the same kind? A first answer is given by planning based on logical representations (e.g. Sacerdoti, 1974; cf. s. 2.4.3). Unfortunately we have seen that there are a number of difficulties that make it impossible to adopt the principles developed in this area of research to design neural network planners (cf. s. 2.3.3).

A second possible answer is given by the literature about abstraction and options (that can be applied to Dyna architectures as well; cf. review in s. 13.2.7; see Sutton et al., 1998). Here the solution is to consider a macro-action (option) as a policy capable of dealing with a certain number of states. As has already been observed, the research in this field is quite new, and few results have been obtained so far.

A last observation on the issue of abstraction is that this chapter has focussed on forward planning only. However, it is straightforward to extend the mechanisms we have explored here to the bidirectional controller analysed in chapter 9 and 10.

Discount Coefficient and Time Limitations of Discounted Reinforcement Learning. The simulations of this chapter have also shown another important result: the opportunity to use different discount coefficients at the coarse and primitive level. In particular they have shown that the choice of the discount coefficient should be done on the basis of the size of the space faced by coarse planning itself, and not on the basis of the level of primitive-actions. As we have seen, this can help the process of planning, but unfortunately it also opens again the problem of interfacing planning and action.

It is important to notice that the problems caused by a discount coefficient that is not adequate for the level of coarseness chosen depends on the problems that affect discounted reinforcement learning when long periods of time are considered. These problems have been introduced in chapter 6 (cf. s. 6.5). These problems implied that the (optimal) evaluations of states far (in time and state space) from the goal are very close to zero. If the environment and the perceptual apparatus of the robot are noisy, the gradient field over these states becomes completely noisy and cannot guide the policy updating. The experiments of this chapter confirm, within the new context of abstract planning, that these problems can impair discounted reinforcement learning severely. These problems are even more important for abstract planning since it, for its nature, tends to be applied to long courses of action.

12 Conclusion and Future Work

12.1 Conclusion: What Have We Learned from This Research?

The goal of this thesis was to implement and investigate predictive planning controllers implemented with neural networks, inspired by the Dyna-PI architecture, and capable of controlling a simulated robot in noisy environments. Pursuing this goal, the thesis has delivered insights that can be organised in six groups. These insights are highlighted and discussed in the following sections, and then summarised in s. 12.2.

12.1.1 Ideas for Neural-Network Reinforcement-Learning Planning

Part 1 has searched for principles and ideas proposed by blind search, heuristic search and planning that could be applied to neural-network planners inspired by the Dyna-architectures. The following ones have been isolated:

- Iterative deepening exploration of the model of the environment. If one has to plan to reach a goal in a large state space, and if the outcome of the actions' execution is not fully predictable (stochastic environment), the goal can be searched by executing searches in depth. To avoid to get stuck in dead-ends or areas of state space too far from the current state it is necessary to "cut" the search to a given depth, and to iteratively start again from the current state. These ideas underlie iterative deepening search, and are applicable to any planner based on the "exploration" of a model of the environment.
- Bidirectional exploration from the start and the goal to focus the evaluations' updating. If one has the goal state and knows that the problem domain studied allows backward search, the strategy of the previous point can be enhanced by searching both from the start and from the goal in parallel.
- Limits of the concept of policy (universal planners) and need for focussing. The debate on universal planners is very useful to highlight a crucial limitation of the concept of policy, namely the requirement to be defined for every possible state. The solution to this problem is to have partial policies, more accurate for the states that one is more likely to visit while executing the actions.
- Importance of the balance between the accuracy of the partial policy (conditional planning) and the possibility of re-planning. If planning is possible, why should we think about all the possible outcomes of actions' execution in advance? It is better to think about what to do for the states that are more likely to be encountered and to re-plan when necessary. This intermediate strategy, "partial policy for the states likely to be visited + re-planning when necessary", can be exploited by neural network planners.

The final chapter of part 1 has presented two "unified views". The first one has tried to isolate the essential characteristics of learning of behaviour and taskable planning. In particular it has underlined that taskable planning consists in the reorganisation of "behaviours" through a *search* of their possible *combinations* (action sequences/policies), on the basis of their capacity to predict the consequences of their execution, in order to achieve a goal (taskability

in a strong sense). It has also shown that learning of behaviour requires a reward function to train the system, and a special "motivational" signal to specify the goal to achieve (taskability in a weak sense). Even if a system can do planning without being taskable in a strong sense, strong taskability is probably a fundamental element for planning. In fact it renders planning flexible and capable of reaching a new goal with improved efficiency from the first time it is pursued, as compared to reactive behaviour. On the basis of this analysis it has been concluded that dyna-PI is not taskable in a strong sense. This analysis was necessary because when building planners with neural networks it is difficult to trace a clear boundary between learning to achieve multiple "desired" states and planning to achieve goals.

The second unified view has shown that the majority of the neural planning controllers proposed in literature is based on some form of evaluation gradient field. On one side this has positive effects for neural planning since neural networks are suitable to implement evaluation functions. On the other side it raises the question about the possibility of building neural network planners based on different principles. We have seen that STRIPS planning, using a different mechanism, "assembles" sequences of actions that lead to the goal on the basis of the (logical) matching between their preconditions and consequences, and a search of their possible combinations. Is it possible to find a neural equivalent of this mechanism? A speculative hypothesis about how to do it, could be the following one. Reciprocal activation and inhibition between clusters of units, where each cluster corresponds to a macro-action, could substitute the "matching" between the preconditions and consequences of actions. A mechanism that brings the activation of the network of clusters to converge towards a maximum level by "trying" different combinations of clusters (similarly to simulated annealing?) could substitute the "search" process.

12.1.2 Landmark Navigation, Reinforcement Learning and Neural Networks

The implementation of reinforcement learning with neural networks, applied to landmark navigation tasks, has produced interesting results. It is well known that reinforcement learning needs some kind of function approximation to be used for robots' control (e.g. Sutton and Barto, 1998, p. 193). The experiments of Chapter 6 have confirmed that neural networks are a powerful function approximation device that can be used for this purpose. In particular the experiments have shown the kind of generalisation that arises from the sight of the landmarks in particular directions with respect to the simulated robot.

The controllers designed and implemented in this research have been tested with a landmark navigation problem. Chapter 6 has shown that generalisation speeds up learning, but also that it can exacerbate the "perceptual aliasing" problem. The simulations have shown that for navigation path-finding tasks this problem is particularly impairing if it affects two or more positions one of which is the goal. In this circumstance the evaluation of the positions different from the goal, but similar to it, tend to be higher than they should be, and to "attract" the simulated robot.

12.1.3 A New Neural Forward Planner

Taskability, given for guaranteed in the classic artificial intelligence planning literature (cf. Allen et al., 1990), relies on general knowledge stored in a "model of the environment" that can be used to pursue *different goals* (s. 5.1). The basic Dyna-PI architecture is not taskable. In fact it requires a model of the environment composed of two parts: one about the consequences of actions and one about the rewards. In order to build the second component of the model, the controller needs to reach the goal several times (or alternatively this component

of the model has to be furnished by the user/designer). This is the reason why the Dyna-PI architecture has been used to speed up learning, but not to implement "genuine" predictive planning (e.g. Sutton, 1990; Lin, 1992).

In order to avoid this problem a new *taskable neural planner*, inspired by the Dyna-PI architecture but different from it, has been implemented ("neural forward planner"). The most interesting aspects of this controller can be described as follows:

- The part of the model of the environment relative to rewards has been eliminated with a double choice. First, the applicability of the new controller has been restricted to "stochastic path-finding problems" with one goal, a subset of the "reinforcement learning problems". This has made it possible to have only one state with a positive reward (the goal), while all the other states have reward 0. Second, a "matcher" has substituted the component of the model related to rewards. The matcher is a neural network capable of generating a reinforcement signal by comparing the goal with the current input pattern. This choice represents an important departure from the reinforcement learning approach from a theoretical point of view.
- The second component of the model of the environment, the state transition function, has been implemented with a neural network, the "predictor". The predictor is capable of autonomously building a model of the environment through experience. This is an important difference in comparison to the traditional artificial intelligence planners where the model of the environment is given to the system a-priori (cf. s. 4.5 for some other examples where the model of the environment is learned). In fact it can be used to enhance the autonomy of intelligent agents.
- The predictor has the capacity to recover from noise when a long sequence of predictions is generated ("mental walks"). This capacity is based on an interesting mechanism: the images corresponding to states of the environment tend to be "attractors" for the predictor's output. Further investigation is needed to test the robustness of this principle that is so important for a core function of the planners presented here, the generation of long sequences of predicted future states.
- The prediction capacity of the predictor can be enhanced by using hardwired modularity, where each "expert" module is specialised in predicting the consequences of the execution of one specific action. This principle is important since any powerful hetero-associative neural network capable of mapping states into states can be used to implement the predictor. This principle can be applied within any controller that uses a limited number of actions (cf. also Lin and Mitchell, 1992, on this).
- The forward planner uses a new mechanism to find a balance between "conditional-planning" and "re-planning" (cf. s. 2.4.4). This mechanism relies on the "confidence" that the controller has in action, measured as the highest of the actions' probabilities. The controller plans when the confidence is below a certain threshold and acts when it is above the threshold. The problem of when acting and when planning has been shown to be a crucial problem for controllers that incorporate both functions.
- The process of planning is controlled by an algorithm that executes a number of forward explorations of the model of the environment with increasing depth. At the beginning these searches follow a random walk, then they become biased toward the goal. This method is inspired by the idea of "trajectory sampling" proposed in Sutton and Barto (1998, pp. 246-249; cf. s. 3.4). The algorithm is new in that it regulates the length of the search so as to: (a) avoid that the planning process gets stuck in dead-ends; (b) focus the exploration on the critical areas around the start, the goal, and between them.
- Controllers for robots often use different types of knowledge representation to execute planning and to act reactively (e.g. Arkin, 1989; Gatt, 1992), for example they use

STRIPS-like representations to plan and numerical functions to act (cf. Arkin, 1998, for a review). This approach raises the difficult problem of creating a suitable interface between the "deliberative" and the "reactive" components of the system (Arkin, 1998, p. 234). The controllers implemented here follow a different approach: both the deliberative and reactive layers function on the basis of numerical representation and processing of them. For example the predictor, the core component of the planning process, works on the basis of neural activation patterns ("images" of the environment). Few other planning controllers share this feature (cf. s. 4.5 and 2.6 for some examples). This topic touches issues as the "symbol grounding problem" and the symbolic/subsymbolic representation problem, long debated in the literature (e.g. cf. Harnad, 1990; Harnad, 1993; Sun, 2000). Even if these issues are interesting, they have been avoided because outside the scope of this thesis.

The neural forward planner represents an important departure from the original Dyna-PI architecture and from the traditional artificial intelligence planners, and hopefully this thesis has succeeded in showing its potentialities and limitations.

12.1.4 A New Neural Bidirectional Planner

The neural forward planner has a major drawback. When it pursues a goal for the first time, it explores the model of the environment on the basis of a random walk. This is typical of the majority of systems based on reinforcement learning. In fact, if no heuristic is available, and if the environment is stochastic and a systematic exploration of it is not possible, this is one of the few alternatives available (cf. Thrun, 1992, for a review of other techniques that use the frequency of visit of action-state pairs to bias exploration). However, in the case of the neural forward planner the fact that the controller is applied to stochastic path-finding problems makes the *goal state available* to the controller. This can be used to explore the model of the environment backward from the goal. The new controller proposed in chapter 9, the "neural bidirectional planner", exploits this possibility. The neural bidirectional planner is different from the neural forward planner in the following aspects:

- A "back-actor" has been added to the controller. This is a neural network capable of selecting actions in order to generate backward explorations from the goal. This neural network is trained to select actions so that the backward searches quickly "escape" from the goal. A "back-predictor" has been added to the controller. This is a neural network that, together with the back-actor, is used to generate backward explorations.
- The controller alternately generates forward explorations from the current state, and backward explorations from the goal.

The neural bidirectional planner maintains the strengths of the neural forward planner and has also two advantages in comparison to it. The first is that it is superior in terms of exploration because it "finds" the goal immediately. The second is that it is superior in terms of propagation of the evaluations away from the goal because it updates evaluations on the basis of states whose evaluations have just been updated. These advantages become more important when the size of the problem space increases.

Unfortunately the bidirectional planner also has some drawbacks. In particular it is not clear to which problem domains different from navigation it is applicable, and it has a complex architecture and functioning. A new "goal oriented forward planner" has also been proposed (but not implemented) that may have the simplicity of the forward planner and some of the strengths of the bidirectional planner. This might be tested in the future.

Bidirectional planning represents a further step in focussing evaluation updating while planning on relevant states. Focussing has a crucial importance for reducing the time

complexity of algorithms inspired by the Dyna-architectures. By referring to a navigation task, Figure 12.1 graphically summarises all the steps that have been done in this direction: (a) using planning only when necessary on the basis of the controller's confidence, and acting otherwise; (b) using sample backups instead of full backups; (c) using trajectory sampling (d); searching forward from the start (or current state); (e) searching backward from the goal.

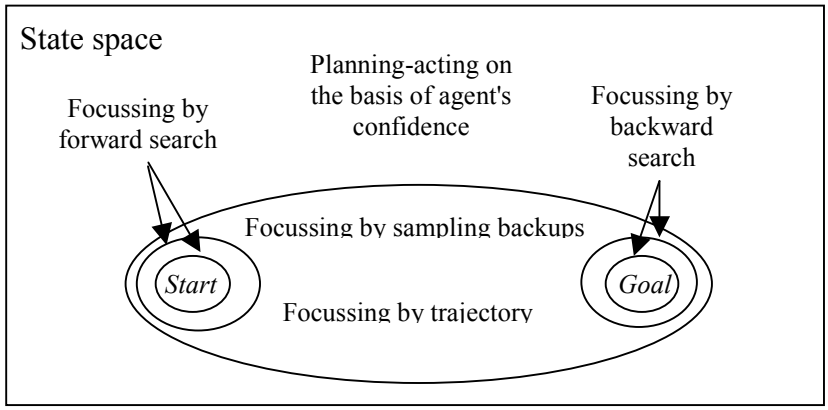


Figure 12.1: Graphical summary of the steps followed to minimise the time spent planning and to focus planning backups on relevant states

12.1.5 Common Structure, Interference, and Modular Networks

Neural networks' generalisation property allows controllers to compress information when solving different tasks, and to overcome problems such as the large size of the state space. Unfortunately, this property also causes interference problems. Preserving the advantage of generalisation and limiting the problem of interference is a crucial target for a system based on neural networks.

In order to study this problem an asynchronous multi-goal navigation task, where different goals are pursued in distinct non-overlapping long periods of time, has been introduced in chapter 7. The experiments of chapter 7 with monolithic neural networks have shown that interference slows down their learning speed severely. The same chapter has explored the utility of modularity for keeping the interference problem under control. In particular it has proposed a controller with two new features in comparison to the basic actor-critic architecture, namely: (a) a mixture of experts neural network is used to implement the evaluator (this is a novel application of this neural network); (b) a novel two level hierarchical architecture is used to implement the actor. The experiments with this controller have produced the following positive results:

- By using modularity the controller limits the problems caused by interference and keeps the generalisation property. This result have already been achieved for supervised learning tasks, for example Jacobs and Jordan (1991) used a mixture of experts network to control a simulated two-joint robot arm with a variety of payloads, moving along a desired trajectory. However, the application to a multi-goal reinforcement learning problem is new.
- The evaluator's modular architecture uses different modules to deal with different goals, while the actor's modular architecture shows a fuzzy specialisation. It is not clear if these different levels of specialisation of the evaluator and actor have been influenced by their

different role played within the system or by the different algorithms used to train them (supervised learning and unsupervised learning guided by the reward signal respectively).

From these experiments it appears that modularity can make an important contribution to exploiting the advantages rendered by the generalisation property and to avoiding interference at the same time. From the experiments it also appears that a key aspect for the successful application of modularity is the functioning of the "gating mechanisms". These are the mechanisms that decide when two different input-output associations share "enough common structure" and can be handled by the same module, and when they do not and have to be handled by different modules. Further research needs to focus on this aspect (cf. Ramamurti and Ghosh (1997) for the possibility of implementing the gating network with a local function approximator that should facilitate the specialisation of the experts).

Chapters 10 has extended the investigation on modularity by integrating the modular version of the actor-critic architecture just considered, and the planning controllers described in chapter 8 and 9. The tests with the modular planning controllers have produced positive results:

- The planning controllers are capable both of generalising and of limiting the negative effects of interference.
- The planning controllers retain the positive strengths shown when dealing with single-goal tasks: taskability, accumulation of experience, capacity to focus exploration.

These tests were necessary because planning focuses on the same goal for long periods of time, and this can augment interference problems. Overall, chapters 10 should have given enough evidence that interference is an important problem for neural network planners, and that modularity may offer a solution for it.

12.1.6 Coarse Planning and Time Limits of Reinforcement Learning

The benefits of abstractions are well known in the classic artificial intelligence literature. How can abstract planning be implemented with neural networks? Chapter 11 has proposed and implemented a "coarse-planner", based on the neural planner introduced in chapter 8, that gives a first simple solution to this problem. The coarse planner executes planning on the basis of "macro-actions", defined as sequences of primitive-actions with the same nature (e.g. "north-north-north-north"), and then selects and executes primitive-actions in the environment. The simulations with the coarse planner have produced the following results:

- The policy and the evaluations generated when planning at a coarse level are suitable for controlling action at a primitive level. This is possible because the "direction" of the optimal macro-action and the optimal primitive-action for a given state and goal, tend to be the same.
- Thanks to the generalisation property of neural networks, the controller is capable of dealing with states that have not been considered while planning at a coarse level but are similar to those that have been experienced (this capacity is also exploited while doing planning at the primitive level). Moreover, the noise that affects actions allows the planner to explore all possible states while training the model of the environment.
- Coarse planning shows a better performance than planning executed at a primitive level. The reason for this is that coarse planning allows a quicker exploration of the whole state space and a faster diffusion of the evaluations between different regions of the state space.

Unluckily coarse planning appears to be applicable only to domains where the effects of actions' execution are approximately linear. How to implement more general forms of abstract planning with neural planners remains an open question.

Chapter 11 should have shown that there might be original ways to implement abstract planning with neural networks that are different from the ones proposed in the classic planning literature. It should also have shown that, within the context of neural-network Dyna-inspired controllers, abstraction might produce unexpected and different advantages that add to the known ones.

Time Limitations of Reinforcement Learning. In comparison to reactive behaviour, planning, and in particular abstract planning, expresses its full potentiality when it is extended over long periods of time. The investigation of coarse planning (chapter 11) has re-examined an important limitation of discounted reinforcement learning (introduced and investigated in chapter 6) from a different perspective. The limitation is that in discounted reinforcement learning evaluations drop exponentially for states progressively more distant (in time) from the goal state, and so they are close to 0 for states very far from it. Given that the one-step learning signal is built on the basis of the difference between two of these evaluations, and given that with function approximation and robots with noisy sensors and effectors the evaluations are affected by noise, this learning signal is low and noisy itself. This can severely disrupt the process of learning of the policy. Notice that the slow rate of learning of reinforcement learning in problems where the start is far from the goal, is usually attributed to the number of states separating them, or to problems of exploration, not to an intrinsic limitation of discounted reinforcement-learning (e.g. see Thrun, 1992).

By using macro-actions with different time-length, the experiments with coarse planning have shown that the reinforcement learning's time limitations just described have important negative effects on planning. A solution has been implemented that is based on using different discount coefficients for planning and for action. Unfortunately this solution causes problems to the interface between planning and acting. The conclusion is that time limitations of discounted reinforcement learning need further investigation and adequate solutions if we want to use such type of reinforcement learning to deal with long lasting tasks.

12.2 A List of the Major "Usable" Insights Delivered

This research has shown that it is possible to build interesting neural planners inspired by the Dyna-PI architecture. In particular the thesis has explored this possibility by proposing a novel neural forward planner, a novel bidirectional planner, a novel modular neural-network version of these two planners, and a novel coarse planner. The investigation of these planners has delivered several insights that can be used when implementing planning with neural networks within the Dyna-architectures framework. The most important insights can be summarised as follows (in order of presentation in the chapters):

- Some principles proposed by blind search, heuristic search, and planning can be exploited to build neural networks planners inspired by the Dyna-framework. Some of these have been isolated and suitably adapted to be applied to neural controllers: iterative deepening exploration of the model of the environment; bidirectional exploration from the start and the goal to focus the backups; limits of the concept of policy (universal planners) and need for focussing; importance of the balance between the accuracy of the policy (conditional planning) and the possibility of re-planning.
- The majority of the planning algorithms that can be (or have already been) implemented with neural networks, are based on the construction of some form of evaluation gradient field increasing (or decreasing) toward the goal.

- Discounted reinforcement learning has severe limitations when behaviours with a long time-horizon are considered. These limitations have to be solved if one wants to develop successful planning systems based on reinforcement learning.
- The aliasing problem has been confirmed to be particularly impairing for reinforcement learning based navigation when it involves the goal states.
- The generalisation capacity of neural networks has been confirmed to be crucial in dealing with noise and in reducing the curse of dimensionality problem deriving from big state spaces. However, it also has been shown that it causes the undesired effect of catastrophic interference. One way of facing catastrophic interference is to use mixture of experts networks for the evaluator and a hierarchical modular neural network for the actor. This solution works both for reinforcement leaning and for planning.
- The Dyna-PI architecture is not taskable in a strong sense because when a new goal is assigned to the controller, it needs experience with that goal to acquire the reward component of the model of the environment relative to it. A solution is to limit the applicability of the architecture to shortest-path problems and to use a device, such as the matcher, to "generate internally" the reward signals.
- When planning, the learning capacity of neural networks allows planning controllers to acquire a model of the environment autonomously (predictor). The experiments have furnished some data about how hard it is for the landmark navigation task studied here, and have shown that hardwired modularity, based on the available actions, can help the process of learning.
- When iterating the predictions to generate simulated walks the neural model of the environment (predictor) has an interesting capacity to recover from noise since the images corresponding to the environment states tend to be "attractors" for the states predicted.
- The neural model of the environment is the most delicate component of the controllers proposed, and probably of any planner based on neural networks.
- For an autonomous robot it is crucial to plan and re-plan when necessary, and to act reactively the rest of the time. The decision of when to act and when to plan can be based on the controllers' "confidence", built on the basis of the probabilities assigned to the actions by the reactive components in a given state.
- A method has been proposed for implementing backward planning with neural Dyna-PI architectures. Backward planning, together with forward planning, allows the controller: (a) to focus search around the goal, the start, and the states between them; (b) to quickly diffuse the evaluations.
- A way to implement a simple form of abstract planning with neural networks, "coarse planning", has been proposed. The linearity of problems, the noise of actions, and the generalisation capacity of neural networks help to interface planning at a coarse level and acting at a fine level. Coarse planning improves exploration and evaluation updating.

The hope is that these insights will be helpful to further develop the investigation of neural-network predictive planners in general, and those inspired by the Dyna-PI architecture in particular.

12.3 Future Work

The experiments run have shown several interesting directions along which the investigation of the neural planners proposed in the thesis might continue. Some of these have been indicated in the s. 12.1. Here the most interesting remaining ones are analysed.

Simulated and Real Robots. This thesis has chosen to work with *simulated* robots and tasks to speed-up and ease the preliminary analysis of several aspects of the controllers proposed (cf. s. 1.1.2). The range of exploration of the thesis would have not been possible had a real robot and more complex tasks been used. This is a common practice (cf. Sutton and Barto, 1998; Arkin, 1998). Now that a lot of knowledge has been gathered on the general behaviour of the controllers proposed, it has become important to test the same controllers with real robots and more difficult tasks. In this case, it is likely that the following problems would arise:

- Limits of the predictor. It is likely that the predictor will show to have strong limitations when dealing with a more complex environment. Could modularity help? Modularity could be based on different actions or refer to different areas of the problem space.
- Aliasing problem. The next header considers this point in detail.
- Abstract planning. When the problem space is big and requires a prolonged activity, abstraction (over the details of actions and over time) becomes very important.
- Modularity to avoid interference and to allow a more flexible behaviour. When the task requires a large number of activities, modularity can help to deal with interference and to enhance the flexibility of the system (i.e. to use the same modules/actions for different tasks).

Aliasing Problem. For this thesis it has been decided to use a very simple "feature extractor" (it executed the extraction of "visual contrasts") in order to speed up the simulations, given that the focus of the thesis was not the efficiency and scalability of the system (cf. s. 6.4.5). We have seen that this choice is one of the main causes of the aliasing problem observed in the simulations, since the simple feature extractor recodes the input image into contrast images that have many overlapping features. The aliasing problem limits the effectiveness of the controller. Now that the main behaviour of the controller has been studied, it would be interesting to address the problem by using one or more of the following solutions:

- Adding/using sensors that allow the system to disambiguate similar states (this solution might incur in the problem of enlarging the state space).
- Rethinking the architecture of the neural networks used to make up the planners so that they can use non-overlapping internal representations for states to be associated with different evaluations and actions (the simple two-layer networks used had few degrees of freedom). In literature a large number of solutions have been proposed to achieve this result. Just to mention some: radial-basis function networks (Sutton and Barto, 1998; Haykin, 1999), growing radial-basis function networks (e.g. Samejima and Omori, 1999), Kanerva coding (Kanerva, 1988; Sutton and Whitehead, 1993), CMAC (Albus, 1981; Miller et al., 1990; Wiering et al, 1998).
- Using other kinds of reinforcement learning techniques that the literature is currently investigating (cf. Singh et al. 1994; Jaakkola et al., 1995; Lorincz et al., 2001; Wiering and Schmidhuber, 1998) that have been shown to have some potentialities with tasks made difficult by the aliasing problem.

Interference and Modularity. The problem of interference and modularity is connected related to the previous one. Chapters 7 and 10 have shown that modularity is a promising idea that can be used to limit interference and to improve the scalability of the controllers. The chapters have also suggested that the major limitation of the modular controllers proposed here is the interference that happens at the level of the evaluator's gating network (cf. s. 7.5). This limits the capacity of the evaluator to use different experts when necessary. A solution to

this problem could be the application of the architecture proposed by Ramamurti and Ghosh (1997) that is based on a gating network that implements a local function approximator.

A related issue is the possibility that the mixture of experts network is too rigid and not capable of discovering "structure" common to different problems. We refer the reader to s. 7.6 for a discussion of this issue that may be the object of future investigation.

Learning the Model of the Environment and Acting. Throughout the thesis, the controllers have learned the model of the environment before being tested. This raises the question of what would be the behaviour of the controllers if the model of the environment had to be learned while acting. The architecture as it is now would not work, because the controller would never reach the proper threshold of confidence. A possible solution would be a varying threshold that drops if the planning activity fails to increase the confidence after some time, and remain low for a considerable amount of time during which the controller acquires experience about the environment. An alternative would be to build some measure of the quality of the model of the environment, and base the planning activity on such a measure.

Acting-Planning Controller. The idea of basing the decision about when to plan and when to act on the controller's confidence compared with a threshold can be further investigated. For example the threshold could vary on the basis of some measures of the "urgency" of action, or could vary on the basis of some measure of the costs of the consequences of "wrong" actions.

Simplifying the Architecture. The architecture of the planners proposed is rather complex, especially the one of the modular bidirectional planner. As mentioned there is the possibility of integrating the predictor and the back-predictor, and maybe the actor and back-actor. Moreover s. 9.6 has proposed a planner with an architecture whose complexity is comparable with the complexity of the forward planner's architecture, but that shares some strengths with the bidirectional planner.

Improving Coarse Planning. Coarse planning has been shown to have many potentialities. New investigations should verify if it is possible to extend the kind of coarse planning presented here to problem domains different from navigation. The literature on reinforcement learning has just started to tackle this problem, but with little success so far (cf. s. 13.2.7).

Limitation of Discounted Reinforcement Learning, Sub-Goals. The results of this research suggest that discounted reinforcement learning is limited in its time scope, and so is any form of planning based on it. Planning, and especially coarse planning, expresses its full power versus reactive behaviour when it can be applied to big chunks of behaviour and long periods of time. The implication is that solutions need to be found for this problem if one wants to build successful planners on the basis of reinforcement learning. As shown here, one possibility to overcome this problem is to use abstraction, where each "step" of the abstract level covers a longer period of time than a step at the "primitive" level (cf. also Linaker, 2001). Another interesting possibility is to "break" long behavioural sequences into parts on the basis of sub-goals. Each sub-goal would possess its own gradient field and policy. The "summation" of more sub-goals would lead to the final goal (e.g. cf. Wiering and Schmidhuber, 1998).

13 Appendices

Appendix 1

13.1 Blind-Search and Heuristic-Search Strategies

This appendix reviews some major blind-search strategies and heuristic-search strategies (Korf, 1988). The quality of these strategies is evaluated through the following criteria:

- Completeness and optimality. These are two criteria used to evaluate the quality of different search strategies. A strategy is complete if it guarantees to find a solution in the case that there is one. Optimality is a measure of the quality of the solution found in terms of its cost. A search strategy is "optimal" if it finds the solution with the lowest cost among all the possible solutions.
- Time complexity and space complexity. Time complexity and space complexity are respectively the measures of the time and memory that the search strategy needs to perform the search. Usually a "complexity asymptotic analysis" and a O[.] notation is used to give an approximate measure of these complexities (cf. Russell and Norvig, 1995, pp. 851-853). O[.] indicates the approximate number of steps taken by the algorithm to process the input. Roughly speaking, this number is based on one or more parameters (arguments of the function O[.]) that describe the size of the algorithm's input space, abstracts over small constant factors, and is based on "pessimistic" values of the input space parameters. This concepts will appear clearer with the examples given below.

13.1.1 Blind-Search Strategies

Breadth-First Search. In this search strategy the root node is expanded first, then all the nodes generated in this first step are expanded, then all the nodes generated in this second step are expanded, and so on. Breadth-first search is complete and optimal. If we assume a branching factor b (the average number of branches for each node), and that the solution of the problem has a path of length d, then the time and space complexity of the strategy is $O[b^d]$. This makes breadth-first search exponentially expensive with d in terms of memory and time. This implies that the worst drawback of this search strategy is space complexity.

Uniform Cost Search. This search strategy is similar to the previous one, but now the strategy expands the lowest-cost node s on the fringe, as measured by the path cost g(s) from the initial state to s. This strategy is complete and optimal. It has time and space complexity similar to those of breadth-first search.

Uniform Cost Search from the Goal. This search strategy is similar to uniform cost search. The difference is that it expands the lowest-cost nodes starting from the goal instead of the initial state. This search strategy is usually not directly mentioned in the literature. It can be considered part of the bidirectional search (see below) if costs are taken into account. Here it

is reported explicitly because it is useful to build a unified view of planning methods based on "evaluations" (cf. s. 5.2).

Depth-First Search. This search strategy expands the nodes at the deepest level of the tree, i.e. until the goal or a dead end is reached, and then backtracks. This search strategy is neither complete, because it can get stuck in branches with infinite length, nor optimal, because it may achieve the solution through a long path before having tried the shorter ones. If m is the maximum depth, then depth-first search has O[b×m] space complexity and O[b^m] time complexity.

Iterative-Deepening Search. This search strategy (Korf, 1985a) combines the benefits of breadth-first search and depth-first. It consists of a depth-first search with a limited depth. The depth starts with a depth of 1 expansion and is iteratively increased by 1. It is complete and optimal. The number of state expansions is wasteful as some states are expanded multiple times. However, this implies little computational inefficiency, as the biggest consumption of computation is caused by the last nodes so that its time complexity is O[b^d] as for breadth-first search. However, it is very efficient with regards to memory: its space complexity is O[bd].

Bidirectional Search. This search strategy (Doran, 1966; Pohl, 1971) is based on the simultaneous expansion of nodes both forward from the initial state and backward from the goal with one of the previous search strategies. It stops when the two searches meet in the middle. It is complete and optimal. This search is very efficient in terms of time complexity given that the depth of the solution is cut by 2: its time complexity is O[$b^{d/2}$]. Its space complexity is O[$b^{d/2}$] since the outcomes of one of the two searches have to be retained in memory to detect if the two searches have met. Two problems with bidirectional search are that: (a) to be applicable it must be possible to produce a search backward from the goal (backward operators); (b) it implies difficulties if many goal states are considered.

13.1.2 Heuristic-Search Strategies

Greedy Search. Greedy search implies that the nodes with the least estimated cost h[s] to the goal are expanded first. The greedy search is not optimal and is not complete because it can go down an infinite path and never return back. Its time and space complexity are O[b^m] where m is the maximum depth of the search space.

A* Search. This search strategy (Hart et al., 1968) is probably the most popular heuristic-search strategy. It combines the advantages of uniform cost search and greedy search. It expands the nodes that have the minimum sum of path cost plus expected cost to goal: f[s] = g[s] + h[s]. The only restriction that it requires is that h[.] is "admissible". A heuristic function is admissible if it is optimistic, in the sense that it never overestimates the cost to the goal. It has been demonstrated (Dechter and Pearl, 1985) that A* search is complete and optimal. Unfortunately, even if A* provides enormous computational savings compared to blind-search strategies, its time complexity still grows exponentially with d, the distance from the goal, for the majority of heuristics of practical importance. However, space complexity is the main drawback of A* (but see below) because it needs to keep all generated nodes in memory.

IDA* - Iterative Deepening A*. IDA* search (Korf, 1985b) is a variant of A* that diminishes the memory requirements of A* search. It is basically a limited-depth-first search.

In IDA* the depth is limited on the basis of increasing fixed values of f[.], instead of the number of nodes of the explored path as in the limited-depth-first search. As A* search, IDA* search is complete and optimal. Its space complexity is O[bd]. Its time complexity depends on the number of values assumed by f[.] during the search. If n is the number of nodes expanded by A*, then, in the worst case when f[.] assumes a new value for each node expanded, the time complexity of IDA* search is $1 + 2 + 3 + \ldots + n = ((n+1)\, n) / 2 = O[n^2]$.

Learning Real-Time A*. LRTA* (Korf, 1990; Jokoo and Ishida, 1999) allows agents to interleave planning and execution (hence "real-time"). The agent *experiences the problem many times* (trials), during which it *updates (learns)* the estimate of the heuristic h[.]. At each time step the agent repeats the following procedure:

- Lookahead. Calculate f[j] = k[i, j] + h[j] for each neighbour j of the current node i, where k[i, j] is the cost from i to j and h[j] is the current estimate of the shortest distance from j to one goal node.
- Update. Update the estimate of node i as follows: $h[i] \leftarrow \min_j[f[j]]$.
- Action selection. Move to the neighbour j that has the minimum f[j] value. Ties are broken randomly.

LRTA* is complete under the assumptions that: h[.] are *initially* non-negative and *admissible*; each link has positive costs; there exist a path from every node to a goal node. It is also optimal over repeated problem solving trials, i.e. the values h[i] converge to their actual true values h*[i] computed as the cost of the optimal path from i to one goal node. Notice that if no heuristic is available, the initial h values can be set at 0. In this case what will happen is that with repeated updates the h values will grow starting from the ones around the initial state and will create a wave-front of values decreasing from the initial state toward the goal. Through repeated trials this wave front will eventually reach the goal and the algorithm will build the correct heuristic function to reach the goal through the an optimal path.

Appendix 2

13.2 Markov Decision Processes, Reinforcement Learning and Dynamic Programming

13.2.1 Markov Decision Processes

Markov decision problem. The main aspects of "Markov decision processes" are now presented (cf. Puterman, 1994, for details). A Markov decision problem implies that a learning agent interacts with an environment at some discrete time steps t = 1, 2, 3, … . On each time step t the agent perceives a state of the world $s_t \in S$, and on the basis of this it selects an action $a_t \in A$. The environment produces a numerical reward r_{t+1} and a next state s_{t+1} in response to each action a_t executed by the agent. The dynamics of the environment can be modelled by one-step "state-transition probabilities" defined as follows:

$$p^a_{ss'} = Pr[s_{t+1} = s' \mid s_t = s, a_t = a] \qquad \forall s, s' \in S \qquad \forall a \in A \qquad \text{Eq. 13.1}$$

and "one-step expected reward" (a stochastic variable whose probability distribution depends on s, a and s'):

$$r^a_{ss'} = E[r_{t+1} \mid s_{t+1} = s', s_t = s, a_t = a] \qquad \forall s, s' \in S \qquad \forall a \in A \qquad \text{Eq. 13.2}$$

These two sets of quantities together constitute the "one-step model of the environment". In functional terms the "model of the environment" is composed of two functions, the "transition-probability function" and the "reward function". The model's transition-probability function, MTP, maps the current state s_t, the current action a_t and a next state s_{t+1} into the probability of having that particular next state:

$$\text{MTP: } S \times A \times S \rightarrow [0, 1] \qquad \text{Eq. 13.3}$$

When the environment is deterministic, then this part of the model maps the current state s_t and the current action a_t into the next state s_{t+1}:

$$\text{MTP: } S \times A \rightarrow S \qquad \text{Eq. 13.4}$$

The model's reward function, MR, maps the current state s_t, the current action a_t, the next state s_{t+1} and the expected (average) reward $r^a_{ss'}$, into the probability of obtaining this reward:

$$\text{MR: } S \times A \times S \times \Re \rightarrow [0, 1] \qquad \text{Eq. 13.5}$$

In the simpler cases in which a particular reward is deterministically associated with each state, the reward function becomes:

$$\text{MR: } S \rightarrow \Re \qquad \text{Eq. 13.6}$$

This case is relevant when we want to frame the problems involving the achievement of a goal with Markov Decision Processes (cf. s. 3.1).

The agent's objective is to learn a "policy" π, i.e. a mapping from states and actions to probabilities ("actions' probabilities") of selecting each particular action:

$$\pi: S \times A \rightarrow [0, 1] \qquad \text{Eq. 13.7}$$

Notice that when the policy "converges", the probabilities tend to be either 0 or 1, and the policy becomes deterministic (but cf. Jaakkola, 1995).

In the case of deterministic action policies, the policy is a direct mapping from the states to the actions:

$$\pi: S \rightarrow A \qquad \text{Eq. 13.8}$$

The State Evaluation Function $V^{\pi}[s]$. For each state s a "state evaluation function" $V^{\pi}[s]$ is defined that depends on the policy π and is calculated as the sum of expected discounted future rewards starting from t+1:

$$\begin{gathered} V^{\pi}[s] = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s] = \\ = E[r_{t+1} + \gamma (r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \ldots) | s_t = s] = \\ = \Sigma_{a \in A}[\pi[a, s] \Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma E[r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \ldots | s_{t+1} = s'])]] = \\ = \Sigma_{a \in A}[\pi[a, s] \Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma V^{\pi}[s'])]] \end{gathered} \qquad \text{Eq. 13.9}$$

where $\pi[s, a]$ is the probability that the policy selects a in s, E[.] is the mean operator, and $\gamma \in [0, 1]$ is a "discount coefficient". The last of the Eq. 13.9 is the "Bellman equation" (Bellman, 1957). The agent's aim is to find an "optimal policy" π^* that maximises $V^{\pi}[s]$ for all $s \in S$. A policy is defined "optimal" if it yields the "optimal state evaluation function" $V^*[.]$:

$$V^*[s] = \max_{\pi} V^{\pi}[s] = \max_a \Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma V^*[s'])] \qquad \text{Eq. 13.10}$$
$$\forall s, s' \in S \quad \forall a \in A$$

that is called "optimal Bellman equation".

Notice that if the model of the environment is known (transitions probabilities and expected reward), if we treat $V^{\pi}[.]$ as unknowns, then the set of Bellman equations for all $s \in S$ forms a system of |S| equations in |S| unknowns whose unique solution are the values of $V^{\pi}[.]$. Some reinforcement learning methods (Sutton and Barto, 1998; cf. s. 13.2.4) and dynamic programming methods (Ross, 1983; Bertsekas, 1995; cf. s. 13.2.9) estimate these values through iterative algorithms.

The State-Action Evaluation Function $Q^{\pi}[s, a]$. A parallel set of value functions for state-action pairs, rather that for states, is particularly important for learning methods. The value of taking action a in state s under policy π, denoted by Q[s, a], is the expected discounted future reward starting in s, taking a, and henceforth following π:

$$\begin{gathered} Q^{\pi}[s, a] = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s, a_t = a] = \\ = \Sigma_{s' \in S} [p^a_{ss'} (r^a_{ss'} + \gamma V^{\pi}[s'])] = \\ = \Sigma_{s' \in S} [p^a_{ss'} (r^a_{ss'} + \gamma \Sigma_{a' \in A}[\pi[s', a'] Q^{\pi}[s', a']])] \end{gathered} \qquad \text{Eq. 13.11}$$

$Q^{\pi}[s, a]$ is known as the "action-evaluation function" for policy π, and the last formula is the "Bellman equation" for it. The optimal "action-evaluation function" $Q^*[s, a]$ and the corresponding "optimal Bellman equation" are:

$$Q^*[s, a] = \max_{\pi} [Q^{\pi}[s, a]] = \Sigma_{s' \in S} [p^a_{ss'} (r^a_{ss'} + \gamma V^{*\pi}[s'])] = \quad \text{Eq. 13.12}$$
$$= \Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma \max_{a'} Q^*[s', a'])] \qquad \forall s, s' \in S \quad \forall a, a' \in A$$

Notice that if the transition probabilities are known (transitions probabilities and expected reward), and if we treat $Q^{\pi}[.]$ as unknowns, then the set of Bellman equations for all the states $s \in S$ forms a system of $|S|$ equations in $|S|$ unknowns whose unique solution are the values of $Q^{\pi}[.]$. Some reinforcement learning methods (cf. s. 13.2.4) estimate these values through iterative algorithms.

13.2.2 Markov Property and Partially Observable Markov Decision Problems

The whole theory of Markov decision processes is based on the assumption that the problem faced by the agent satisfies the "Markov property". This assumption implies that the information contained in the signal of a state s carries a complete description of the environment at that moment: no memory of previous states or actions selected is needed to generate the perfect transition probabilities and rewards. Formally this is expressed in the following way:

$$Pr[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, \quad s_{t-1}, a_{t-1}, r_{t-1}, \quad s_{t-2}, a_{t-2}, r_{t-2}, \ldots] = \quad \text{Eq. 13.13}$$
$$Pr[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t] \qquad \forall s', r, \quad s_t, a_t, r_t, \quad s_{t-1}, a_{t-1}, r_{t-1}, \ldots$$

In the majority of practical problems, the Markov property does not hold. In fact the agent knows the state of the environment through a sensorial apparatus that returns limited and noisy information about the current state of the environment. For example a camera of a robot returns information about a limited part of the environment surrounding the robot, with a limited definition, etc. In the scenario used in this thesis (cf. s. 6.2) the robot perceives the environment through sensors that return a feature-like pattern, i.e. a vector of real numbers **x**, in correspondence to each state s. This can be considered a quite general case, where the agent is not allowed to directly observe the state of the environment but can receive "messages" from it that contain information about its state. At each time t an observable message **x** is drawn from a finite set of messages **X** according to an unknown probability distribution $Pr[\mathbf{x}|s]$ (cf. Jaakkola et al., 1995). Notice that it is possible that different states s form S are mapped into the same message **x**. This problem, named the "perceptual aliasing problem", is particularly impairing because the "agent's internal representation confounds external world states" (Whitehead and Ballard, 1991).

An environment for which the Markov property does not hold is said to be "partially observable" or "inaccessible" (cf. also s. 2.4), and to generate "Partially Observable Markov Decision Problems" (POMDP). When reinforcement learning methods are applied to a partially observable Markov decision problem they may still work, and their performance degrades gracefully as the degree of "non-Markovianness" increases, but this is not guaranteed (Singh et al., 1994). Some solutions have been proposed to deal with these problems. The majority of these solutions have attempted to combine some forms of estimate of states with learning. The internal representation of the state is built by combining current sensor readings with the *memory* of past internal representations and readings. For example to

this end recurrent neural networks have been used (e.g. Lin and Mitchell, 1992) or probability distributions over underlying states (e.g. Sondik, 1978; McCallum, 1993). A second approach has attempted to use "active perception", i.e. actions directed to gather further information to disambiguate the states (e.g. Whitehead and Ballard, 1990). Another approach has proposed a system that uses the current sensorial information only, and searches between stochastic policies rather than between deterministic policies. This approach has been suggested by the result that for some partially observable Markov decision problems, memory-less stochastic policies are significantly better than any memory-less deterministic policy (Singh et al., 1994; Jaakkola et al., 1995). All these solutions are currently under investigation and each have several drawbacks, so partially observable Markov decision problems are still a fully open chapter of reinforcement learning research.

The scenario used to test the algorithms proposed in the thesis and introduced in s. 6.2 involves a partially observable environment. In s. 6.4.3 some negative consequences that this environment produces on reinforcement learning are shown. These problems have been tolerated and not directly tackled in this research, because the focus was on different issues, i.e. planning with reinforcement learning and neural networks.

13.2.3 Reinforcement Learning

Reinforcement learning methods (Barto et al., 1983; Kaelbling et al., 1996; Sutton and Barto, 1998) attempt to find a policy that solves Markov decision problems in two phases (usually carried out in parallel, as we shall see). In the first phase they build up the evaluations of the states, or the state-action pairs. We have seen in s. 13.2.1 the definitions of the evaluations in these two cases. It is worth stressing that once these evaluations are built, they form a gradient field over the state space. This gradient field has high evaluations for states that are "close" (in terms of number of actions that need to be executed) to the states with relevant positive rewards, and low evaluations for states far from them. For example in the case of the stochastic path-finding problems, the level of evaluations increases going toward the goal. Similarly, the evaluations of the gradient field are low (and negative) for states that are close to states with relevant negative rewards. The second phase exploits this gradient field to build up the policy that leads to states with positive rewards and away from states with negative rewards, as quickly as possible.

13.2.4 Approximating the State or State-Action Evaluations

Updating the Estimates of $V^{\pi}[s]$. If we assume we have a policy π that leads to exploration of the different regions of the problem space (for example, for now, suppose we have a random walk policy), it is possible to progressively find more accurate estimates of the evaluation function $V^{\pi}[s]$. This can be done by using an iterative approximation rule based on the Bellman equation Eq. 13.9. In particular when a state is visited, the following updating rule can be applied to the estimate $V'^{\pi}[s]$ of $V^{\pi}[s]$:

$$V'^{\pi}[s_t] \leftarrow V'^{\pi}[s_t] + \eta\,((r_{t+1} + \gamma\, V'^{\pi}[s_{t+1}]) - V'^{\pi}[s_t]) \qquad \text{Eq. 13.14}$$

This rule is called "TD(0)" (Sutton and Barto, 1998, p. 134), where TD stands for "Temporal Difference". "0" indicates that the updating rule considers two succeeding steps only. In this research only this case is considered, and the so called "TD(λ) rule" (cf. Sutton and Barto, 1998, p. 163-191) is not investigated (cf. s. 6.4.2 for the reasons of this).

The value e_t, defined as:

$$e_t = (r_{t+1} + \gamma V'^{\pi}[s_{t+1}]) - V'^{\pi}[s_t] \qquad \text{Eq. 13.15}$$

is called "TD-error" (Temporal-Difference error). The TD-error represents the difference between two estimates of the true evaluation $V^{\pi}[s_t]$. The first estimate, $(r_{t+1} + \gamma V'^{\pi}[s_{t+1}])$, is expressed at time t+1. The second estimate, $V'^{\pi}[s_t]$, is expressed at time t. The fact that this rule updates the estimate $V'^{\pi}[s_t]$ on the basis of another estimate $(V'^{\pi}[s_{t+1}])$, makes the TD(0) rule a "bootstrapping" method. How is it possible that notwithstanding this bootstrapping process, $V'^{\pi}[s_t]$ converges to the true evaluations? The explanation is that the estimate $(r_{t+1} + \gamma V'^{\pi}[s_{t+1}])$ is more accurate than the estimate $V'^{\pi}[s_t]$ because:

- r_{t+1} is directly experienced, and not guessed as in $V'^{\pi}[s_t]$
- $V'^{\pi}[s_{t+1}]$ is temporally closer to future rewards than $V'^{\pi}[s_t]$, so it is more accurate.

Notice that this rule attempts to compute accurate evaluations of the states according to the *current policy* π, independently of its quality. S. 13.2.5 and 13.2.6 will show how this policy can be improved on the basis of the evaluations. Notice also that this rule updates $V'^{\pi}[s_t]$ on the basis of a *sampling* over the possible resulting states to which the execution of an action might lead. When the evaluation of state s is updated several times the frequencies of visiting of the state s' that follow s reflect the transition probabilities of the Bellman equation (cf. Eq. 13.9).

If each state is visited an infinite number of times, and if some other weak conditions hold, the approximations $V'^{\pi}[s_t]$ converges to the true evaluations $V^{\pi}[s_t]$ (cf. Sutton and Barto, 1998, p. 141). Intuitively what happens during the updating of the evaluations is that the rewards received at the goal states are propagated backward towards the preceding states, then the (discounted) evaluations of these states are propagated backward to the states that preceded them, and so on. Notice that the discount factor implies that the evaluations associated with the states decrease exponentially for states progressively more distant from the states with positive rewards. This means that the evaluations, as initially mentioned, form a gradient field over the states, decreasing for states farther from the goal.

One possible way to implement the algorithm that approximates the V^{π} evaluations is to have a look-up table that stores the V'^{π} in correspondence of the entries given by the states s. This is called "tabular reinforcement learning".

Updating the Estimates of $Q^{\pi}[s, a]$. The same reasoning holds for the updating of the Q evaluations. Assuming a policy π that leads to repeated exploration of different state-action pairs, the following updating rule can be used to iteratively improve the estimates $Q'^{\pi}[s, a]$ of $Q^{\pi}[s, a]$ related to that policy:

$$Q'^{\pi}[s_t, a_t] \leftarrow Q'^{\pi}[s_t, a_t] + \eta ((r_{t+1} + \gamma Q'^{\pi}[s_{t+1}, a_{t+1}]) - Q'^{\pi}[s_t, a_t]) \qquad \text{Eq. 13.16}$$

This rule is called "Sarsa" (Sutton and Barto, 1998, p. 145). If each state-action pair is visited an infinite number of times, and if some other weak conditions hold, the $Q'^{\pi}[s, a]$ estimates converge to $Q^{\pi}[s, a]$ (cf. Sutton and Barto, 1998, p. 145). In the simplest case the $Q'^{\pi}[s, a]$ are stored in a two-dimensional look-up table with s and a as entries ("tabular reinforcement learning").

Sarsa is slightly different from the most popular reinforcement learning algorithm, called "Q-learning" (Watkins, 1989; Barto and Sutton, 1998, p. 148). Q-learning is implemented with the following rule:

$$Q'^*[s_t, a_t] \leftarrow Q'^*[s_t, a_t] + \eta\,((r_{t+1} + \gamma \max_{a'}[Q'^*[s_{t+1}, a']]) - Q'^*[s_t, a_t]) \qquad \text{Eq. 13.17}$$

This rule attempts to directly approximate the optimal evaluations Q* (cf. optimal Bellman Eq. 13.12). The estimates $Q'^*[s_t, a_t]$ converge to the optimal evaluations $Q^*[s_t, a_t]$ under the only condition that each state-action pair is visited an infinite number of times (Watkins, 1989).

Notice that in Q-learning the estimates Q'* converge to the evaluations Q* corresponding to the optimal policy (e.g. the policy that moves to the goal state following the most direct "route") *independently of the policy that is being followed*. For this reason it is called an "off-policy" method. This differs from TD(0) and Sarsa where the estimates V'^{π} and Q'^{π} converge to the V^{π} and Q^{π} related to the *policy π followed at the moment*. These are called "on-policy" methods.

13.2.5 Searching the Policy with the Q'* and Q'^{π} evaluations

How do we build a policy on the basis of the evaluations? The analysis of this problem starts with the easiest case of the Q evaluations and "Q-learning" (cf. Watkins, 1989; Sutton and Barto, 1998, pp. 148-149). Suppose we have a certain policy, e.g. a random walk. The updating rule of Eq. 13.17 leads to progressively more accurate estimates Q'* of Q*. At this point, if we are in a state s and we want to implement a policy that is better than the random walk, we can "ascend" the gradient field of evaluations toward higher evaluations, even if they are approximate, by selecting the actions as follows:

$$a_t = \text{argmax}_{a \in A}[Q'^*[s_t, a]] \qquad \text{Eq. 13.18}$$

This policy is called the "greedy-policy". Notice that we can improve the Q'* evaluations and follow the greedy policy in parallel. The improvement of the evaluations automatically brings an improved policy.

It is important to notice that there is a trade-off between the need to exploit the knowledge incorporated in the evaluations and the need to explore different state-action pairs to improve the global evaluations themselves. The greedy policy does not guarantee enough exploration because at each state it always selects the action with the maximum Q'*. One popular way to improve exploration is to select the best action only with a certain probability $1-\varepsilon$, say equal to 0.95, and to select an action among the other actions with probability $\varepsilon = 0.05$. The selection among these actions is done with a uniform probability distribution. The resulting policy is called "ε-greedy policy". It has been demonstrated (Watkins, 1989) that this policy converges to the optimal policy π^*, and the Q'* converges to Q*, if ε progressively converges to 0 (i.e. to the greedy policy).

A commonly used variant of the ε-greedy policy, is the "soft-max policy". This takes into consideration the fact that the probability of choosing an action should be correlated with the level of Q, instead of being, say, either 0.95 or 0.01. According to the soft-max function (or Boltzmann distribution) the probability Pr[.] that a given action a becomes the winning action a_{win} given the current state s, is:

$$\Pr[a = a_{win}] = \exp[Q'^*[s, a]] / \Sigma_{a' \in A}[\exp[Q'^*[s, a']]] \qquad \text{Eq. 13.19}$$

The case of Sarsa is similar. The greedy policy selects the action with maximum Q'^{π}:

$$a_t = \text{argmax}_{a \in A}[Q'^{\pi}[s_t, a]] \qquad \text{Eq. 13.20}$$

Also in this case an ε-greedy policy or a soft-max policy can be followed to allow exploration. When one of these policies is used in parallel with the updating of the Q'^{π} evaluations, the policy converges to the optimal policy $\pi*$ and the Q'^{π} estimates converge to the optimal evaluations Q* if each state-action pair is visited an infinite number of times and if the policy converges to a greedy policy (Sutton and Barto, 1998, p. 146).

13.2.6 Actor-Critic Model

Notice that an agent that learns on the basis of Q-learning or Sarsa does not need to have a data structure to store the action probabilities of the policy. In fact if Q'* or Q'^{π} are stored in a suitable data structure, the probabilities of actions can be computed on the fly on the basis of the Q'* or Q'^{π} values through the ε-greedy method or the soft-max method. This is what is done in the majority of reinforcement learning applications. A different type of reinforcement learning methods, called "actor-critic methods", is based on the evaluations V'^{π} of Eq. 13.14 (Barto et al., 1983; Sutton and Barto, 1998, pp. 151-153). These methods are particularly relevant because they are at the core of all the architectures and algorithms presented in this thesis.

Actor-critic methods are based on two data structures, one that stores the evaluations, called "critic", and one that stores the policy, called "actor". The term critic is often used to name the data structure storing the evaluations plus the process that computes the error e_t defined later in Eq. 13.21. In the thesis for clarity the term "evaluator" is used to refer to the data structure that stores the evaluations while the term "TD-critic" is used to refer to the process that computes e_t. The actor, the data structure storing the policy, can assume the form of a look-up table that stores the probabilities of the state-action pairs, and has the states s and the actions a as entries ("tabular reinforcement learning"). Alternatively it can store the "action merits" of the state-action pairs, that here are indicated with m[s, a]. An "action merit" is a value that summarises the contribution of that state-action pair to the achievement of the long-term reward. The probabilities used to select the actions are calculated on the basis of the merits (considered as pseudo-probabilities) using a method such as the soft-max method.

It has just been said that the evaluator stores the approximate evaluations V'^{π}. After an action a_t is executed the evaluator evaluates the new state s_{t+1} to determine if it is better or worse than the previous state s_t. This is done by comparing the estimate $V'^{\pi}[s_{t+1}]$ of the new state, and the estimate $V'^{\pi}[s_t]$ of the old state. The comparison has to take into account the fact that the evaluations are expressed in different times, and that a reward could be received when passing from s_t to s_{t+1}. The proper formulation of this comparison is represented by the TD-error e_t of Eq. 13.15, repeated here for convenience:

$$e_t = (r_{t+1} + \gamma V'^{\pi}[s_{t+1}]) - V'^{\pi}[s_t] \qquad \text{Eq. 13.21}$$

This error can be used to correct the evaluations of the evaluator, with the updating rule of Eq. 13.14, here expressed in terms of the TD-error:

$$V'^{\pi}[s_t] \leftarrow V'^{\pi}[s_t] + \eta ((r_{t+1} + \gamma V'^{\pi}[s_{t+1}]) - V'^{\pi}[s_t]) = V'^{\pi}[s_t] + \eta e_t \qquad \text{Eq. 13.22}$$

The interesting thing is that the TD-error is also suitable for updating the merit m[s, a] of the action a_t (but only this one) selected by the actor:

$$m[s, a] \leftarrow m[s, a] + \zeta e_t \qquad \text{Eq. 13.23}$$

where ζ is a learning parameter. The reason why the TD-error is suitable for updating the merit of the action selected is that if the estimate $V'^{\pi}[s_t]$ has converged to the true value $V^{\pi}[s_t]$, it represents the average of the sum of the future discounted rewards *over the actions* that can be selected at s_t, sampled on the basis of their current probabilities. If an action a_t is selected and executed at s_t and we have that $(r_{t+1} + \gamma V'^{\pi}[s_{t+1}]) > V'^{\pi}[s_t]$, this means that the action selected has led to a state s_{t+1} that has a discounted evaluation that is higher than the average discounted evaluation of the states that are usually reached from s_t. In this case the updating rule of Eq. 13.23 suitably increments the merit of the action a_t selected (and hence its probability). On the contrary if we have that $(r_{t+1} + \gamma V'^{\pi}[s_{t+1}]) < V'^{\pi}[s_t]$ this means that the action selected has led to a state that has a discounted evaluation that lower than the average evaluation of the states that are usually reached from s_t. In this case the updating rule of Eq. 13.23 suitably decreases the merit of the action a_t selected (and hence its probability).

The implementation of the actor-critic algorithms implies that the evaluations and the policy are improved in parallel while the agent is acting. The evaluations become more accurate for the states given the current policy, while the current policy is improved toward the optimal policy on the basis of the current evaluations. The parallel updating of evaluations and policy is called "policy iteration".

It has been mentioned that the actor-critic methods are at the core of all the architectures and algorithms presented in this thesis. Why have they been preferred to the more simple and popular Q-learning? There are three reasons for this choice:

- Reinforcement learning methods can be applied with success to real problems only if an approximation method is employed (cf. s. 13.2.8). Sutton et al. (2000) have shown that a version of actor critic algorithm with arbitrary differentiable function approximation converges to a locally optimal policy. In contrast so far the strategies based on Q evaluations have proven theoretically intractable for similar results.
- Stochastic policies can be better than deterministic policies to deal with partially observable Markov decision processes (cf. s. 13.2.8).
- The author is particularly interested in the actor-critic methods because it has been shown that they have an interesting biological plausibility (cf. Sutton and Barto, 1990; Houk et al., 1994; Baldassarre, 2001b and 2002).

13.2.7 Macro-actions and Options

In the last few years, reinforcement learning research has started to investigate the concept of "macro-actions" or "options" within the Markov decision processes framework (Sutton et al., 1998). To make long-term decisions, an agent needs to predict the consequences of the possible courses of action at multiple levels of temporal abstraction. Consider a traveller deciding to undertake a journey to a distant city. The traveller has to decide to go by fly or to drive. Each of these steps involves prediction and decision. After a decision is taken, smallest actions have to be decided. For example calling a taxi may involve finding a telephone, dialling each digit, and so on down to the individual muscle contractions to push the buttons.

A macro-action consists of three components. An "input set":

$$I \subseteq S \qquad \text{Eq. 13.24}$$

a "policy":

$$\pi: S \times A \rightarrow [0, 1] \qquad \text{Eq. 13.25}$$

and a “termination condition”:

$$\beta: S \rightarrow [0, 1] \qquad \text{Eq. 13.26}$$

A macro-action is available at state s only if $s \in I$. The input set restricts the range of application of the option in a potentially useful way. In particular it limits the range over which the option's policy need to be defined. If the option is taken then actions are selected according to π (in this section π is used for the policy of option and μ for the global policy). The option terminates stochastically according to β. When the option terminates, the agent selects another option.

Planning with options requires a model of their consequences. A semi-Markov decision model (SMDM) can be used for this purpose. The adjective “semi-” indicates the fact that within this model the single options are treated as a whole. At this level the Markov assumption holds. At the level of the policy of the single option, the Markov assumption does not hold.

The “reward” part of the model of an option o for any state s, is:

$$r^o_s = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{k-1} r_{t+k} \mid \varepsilon[o, s, t]] \qquad \text{Eq. 13.27}$$

where t+k is the time where o terminates, $\varepsilon[o, s, t]$ is the event that in state s and time t the macro-action o is chosen. The state-prediction part of the model is:

$$p^o_{ss'} = \Sigma^{\infty}_{k=1} [\gamma^k \Pr[s_{t+k} = s' \mid \varepsilon[o, s, t]]] \qquad \text{Eq. 13.28}$$

$p^o_{ss'}$ is a combination of the likelihood that s' is the state in which o terminates, weighted with a measure of how delayed that outcomes is relative to γ.

If we define μ the Markov policy that selects for options in correspondence of a given state, then we can define the value of a state for this option policy as:

$$V^{\mu}[s] = E[r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k} + \gamma^k V^{\mu}[s_{t+k}] \mid \varepsilon[\mu, s, t]] = \qquad \text{Eq. 13.29}$$
$$= \Sigma_{o \in O} [\mu[s, o] (r^o_s + \Sigma_{s'}[p^o_{ss'} V^{\mu}[s']])]$$

where k is the duration of the first option selected by μ. Notice that the discount coefficient is incorporated into the transition probabilities (cf. Eq. 13.28). A similar equation can be written for $Q^{\mu}[s, o]$:

$$Q^{\mu}[s, o] = E[r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k} + \gamma^k V^{\mu}[s_{t+k}] \mid \varepsilon[o, s, t]] = \qquad \text{Eq. 13.30}$$
$$= E[r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k} + \gamma^k \Sigma_{o' \in O} [\mu[s_{t+k}, o'] Q^{\mu}[s_{t+k}, o']] \mid \varepsilon[o, s, t]] =$$
$$= r^o_s + \Sigma_{s'}[p^o_{ss'} \Sigma_{o' \in O} [\mu[s', o'] Q^{\mu}[s', o']]$$

From these equations it is possible to infer the optimal equation that corresponds to the best policy μ^* as we have done in the case of simple actions. It is also possible to define an iterative algorithm to approximate the evaluations of the states:

$$V^{\mu}[s] \leftarrow V^{\mu}[s] + \eta((r^o_s + \gamma^k V^{\mu}[s_{t+k}]) - V^{\mu}[s]) \qquad \text{Eq. 13.31}$$

Notice that for these sample backups the discount factors have to be made explicit since there are no transition probabilities.

For the case of the Q values the iterative algorithm is:

$$Q^{\mu}[s, o] \leftarrow Q^{\mu}[s, o] + \eta((r^{o}_{s} + \gamma^{k} Q^{\mu}[s_{t+k}, o_{t+k}]) - Q^{\mu}[s, o]) \qquad \text{Eq. 13.32}$$

The theory of options is quite new. Few applications and few results have been obtained on its basis (e.g. McGovern et al., 1998; Sutton et al., 1999). Notwithstanding this, the idea of options that it offers is very general, so it is useful as a solid foundation for developing different algorithms for abstract reinforcement learning and abstract planning based on reinforcement learning. Chapter 11 shows how the theory of options has been useful for this research to develop a simple kind of abstract planning.

13.2.8 Function Approximation and Reinforcement Learning

So far it has been assumed that the reinforcement learning methods presented are implemented with states represented as whole discrete entities and look-up tables ("tabular reinforcement learning"). If the number of states is big, this approach is not feasible. It would not be possible to have a data structure to store all possible state evaluations and all possible state-action pairs (space complexity). It would also take too long to accumulate experience about all such states (time complexity).

An example of this problem is a robot endowed with several sensors. Each sensor gives partial information about the state of the world, so that many sensors are needed. Together the sensors return a vector of numbers ("state variables"). If b is the (average) number of states of a sensor, and n is the total number of the agent's sensors, the number of states that the robot can perceive is about b^n. For example the simulated robot used in this research is endowed with a simple one-dimension binary retina with 50 pixels. This implies 2^{50} different possible input configurations. The *exponential* increase for each state variable ("dimension") added, makes it impossible to treat each single state of the problem individually.

The only solution to this difficulty is to use "function approximation methods" (Sutton, 1996; Sutton and Barto, 1998, p. 193). These methods are capable of "generalising", i.e. they can extend the experience accumulated for some states to states described by *similar state variables* (cf. also s. 4.4.1 on this issue).

Some examples of function approximation methods are the following: neural networks (cf. s. 13.3.3; Sutton and Barto, 1998, p. 197-202; Samejima and Omori, 1999); CMACs (Albus, 1981); Kanerva coding (Kanerva, 1988; Sutton and Whitehead, 1993); decision-tree (Chapman and Kaelbling, 1991); explanation-based learning methods (Yee et al., 1990).

13.2.9 Dynamic Programming

Dynamic programming (Ross, 1983; Bertsekas, 1987) refers to a collection of algorithms that can be used to compute evaluations and policies given a model of the environment as the one summarised by Eq. 13.3 and Eq. 13.5. This and the following sections present the most important aspects of dynamic programming relevant for this research.

If a model of the environment is available, the updating of the state evaluations can exploit the fact that the transition probabilities and expected rewards are known. If we assume we have a particular policy $\pi[s]$, then the estimates $V'^{\pi}[s]$ *for each state* can be updated (this is called "sweep") with the following rule:

$$V'^{\pi}[s] \leftarrow \Sigma_{a \in A}[\pi[a, s] \Sigma_{s' \in S}[p^{a}_{ss'} (r^{a}_{ss'} + \gamma V'^{\pi}[s'])]] \qquad \text{Eq. 13.33}$$

This rule directly descends from the Bellman Eq. 13.9, and is called "policy evaluation". Notice that $V'^{\pi}[s]$ is updated according to *all* possible next states ("full backup"), not just one as in the case of the reinforcement learning methods ("sample backup", cf. Eq. 13.14 and Figure 13.1). If this updating rule is applied iteratively the estimates $V'^{\pi}[s]$ converge to the true values $V^{\pi}[s]$ (e.g. Sutton and Barto, 1998, p. 91).

The model of the environment can also be used to compute the greedy policy on the basis of the current evaluations ("policy improvement"). Suppose we have started with a random policy, and then we have executed several cycles of policy evaluation so that the estimates $V'^{\pi}[s]$ are now accurate. We can compute the greedy policy with respect to the new estimates $V'^{\pi}[s]$ by selecting the action a according to the following rule:

$$a = \text{argmax}_a[\Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma V'^{\pi}[s'])]] \qquad \text{Eq. 13.34}$$

The "policy improvement theorem" (e.g. Sutton and Barto, 1998, p. 95) guarantees that the evaluations of all states under the new greedy policy are better than or the same as previously.

If cycles of policy evaluation and cycles of policy improvement are alternated ("policy iteration"), the system converges to the optimal policy and optimal evaluation function if all the states are visited an infinite number of times (e.g. Sutton and Barto, 1998, p. 97).

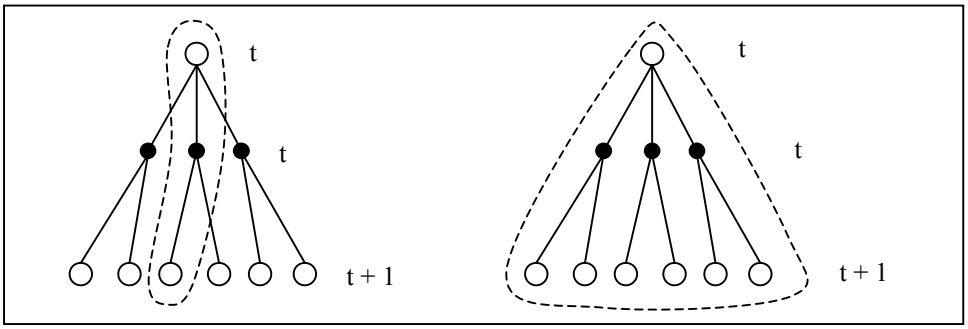


Figure 13.1: Left: sample backup typical of reinforcement learning methods. Right: full backup typical of dynamic programming methods. Empty circles represent states, full circles represent actions, and edges represent selection of actions or transitions to new states. States and actions are marked by the time step when they occur. Dotted lines have been traced around the states and actions involved in the backup.

It is not necessary to wait until the policy evaluation converges to execute a cycle of policy improvement. Policy evaluation can be stopped after just one sweep, and interleaved with cycles of policy improvements ("value iteration"). Convergence is still guaranteed. In particular the policy evaluation and the policy improvement can be combined into a single updating rule:

$$V'^{\pi}[s] \leftarrow \text{max}_a[\Sigma_{s' \in S}[p^a_{ss'} (r^a_{ss'} + \gamma V'^{\pi}[s'])]] \qquad \text{Eq. 13.35}$$

When dynamic programming is applied to a deterministic environment and is being used to estimate the optimal evaluations $V^*[s]$ off-policy, selecting an action means selecting a specific succeeding state (cf. Sutton and Barto, 1998, pp. 156-157). In these circumstances the updating rule becomes:

$$V'[s] \leftarrow \max_a[r^a_s + \gamma V'[s']] \qquad \text{Eq. 13.36}$$

It is interesting to see what happens during the succeeding sweeps of dynamic programming in this deterministic case, assuming that the initial evaluation estimates are 0 and that there is only one goal state. The evaluations will start to be updated for the states close to the goal state and then progressively for the states more distant from the it. The updated evaluations will form a sort of wave front expanding from the goal. The states reached by this wave front will immediately assume the correct value. This is equivalent to what happens with "activation diffusion planning" (cf. s. 4.5.1).

13.2.10Asynchronous Dynamic Programming

One drawback of dynamic programming value iteration is that one sweep requires the updating of *all* the estimates $V'^{\pi}[s]$. This method is called "synchronous dynamic programming". It also requires that the updates be done on the basis of the old evaluations. If the number of states is very big, this method can require a prohibitive amount of time. Luckily there are other approaches that reduce the number of states of which to update the backups. Now these approaches are reviewed.

An approach, "asynchronous dynamic programming", *focuses* the backups on few states and uses the recent values of other states to execute these backups. This approach is still guaranteed to converge if all states are still visited an infinite number of times (Bertsekas, 1995; Barto et al., 1995). Asynchronous dynamic programming is particularly important because it can be mixed with control, i.e. the execution of the actions under the policy. In doing so asynchronous dynamic programming can focus on states more frequently visited under the effect of the policy, i.e. on states more relevant for control. Notice that, given that a model of the environment is available, the execution of the actions selected by the policy can be carried out *in simulation mode*, i.e. through the model itself instead of the real experience.

13.2.11Trial-Based Real-Time Dynamic Programming and Heuristic Search

Asynchronous dynamic programming is still not fully satisfying because it still requires that the agent visit all the states an infinite number of times. "Trial-based real-time dynamic programming" (Barto et al., 1995) is another approach that allows a further focussing of the backups while still converging. Its convergence has been demonstrated for the "stochastic shortest-path problems" defined as follows (the terminology used for Markov decision problems is adopted; when it was not too limiting the problem definition has been restricted to simplify its presentation):

- The problem consists of a set of states S. Some of these states are called "absorbing states" (goal states). Any action executed at an absorbing state leads to the same absorbing state with probability 1. Any action executed at an absorbing state has reward 0. Any action executed at a non-goal state leads to a negative non-zero reward ("cost"). The discount coefficient is 1.
- The problem is divided in "trials". A trial is a finite number of time steps during which the agent can pursue the goal. The length of the trial is enough to reach a goal-state from any "initial state" (see below). The time limit imposed by trials has an important effect. It prevents getting stuck in endless cycles. The length of trials can be extended progressively to ensure that it becomes long enough to reach the goals (cf. Barto et al., 1995).

- There is a subset of "start states" from which the agent pursues the goal at the beginning of each trial.
- A subset of states named "relevant states" is defined. A relevant state is a state that can be reached by the execution of any optimal policy from any possible start state. The states that cannot be reached in this way are defined "irrelevant states".

Trial-based real-time dynamic programming executes an infinite number of trials from each start state of the stochastic shortest-path problem. It concurrently executes control and asynchronous dynamic programming updating of the evaluations of the states visited. In particular it always executes the backup of the state visited under control, and *eventually* other backups of other states. For each state visited it follows this procedure:

- Look-ahead. Compute all the possible values $\Sigma_{s' \in S}[P^a_{ss'} (r^a_{ss'} + \gamma V'^{\pi}[s'])]$ that can be obtained from the current state s by selecting each of the actions a.
- Update evaluations' estimates. Backup the current state on the basis of Eq. 13.35. Eventually execute backups for other states (e.g. by generating a short look-ahead search from the current state).
- Action selection. Follow the greedy policy with respect to the most recent evaluation estimates (ties are resolved randomly).

A theorem from Barto et al. (1995) asserts that, if applied to a stochastic shortest-path problem, trial-based real-time dynamic programming converges to the optimal evaluation function and optimal policy *on the set of relevant states* under the following conditions: (a) the initial evaluations of the goals states is 0; (b) the initial evaluations for the non-goal states are optimistic (e.g. they are 0).

Now a bridge between Markov decision processes and heuristic search can be built. This can be done very simply by noticing that the three steps of learning real time A* illustrated in s. 13.1.2 have a close correspondence with the three steps of trial-based real-time dynamic programming just illustrated, in the case this is applied to a deterministic problem. This correspondence was first demonstrated by Barto et al. (1995). Cf. s. 3.2 for some critical observations on this correspondence.

Appendix 3

13.3 Feed-Forward Architectures and Mixture of Experts Networks

13.3.1 Feed-Forward Architectures and Error Backpropagation Algorithm

The error backpropagation algorithm (Rumelhart et al., 1986) is usually used to train a feed-forward neural network with three layers (or more, see Figure 13.2).

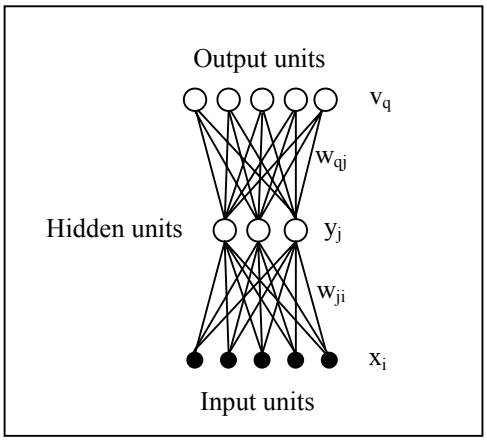


Figure 13.2: A feed-forward neural network trained with a back-propagation algorithm. Full circles are pass-through units. Empty circles are sigmoidal units or units with other types of transfer functions.

The "input layer" is made up of simple pass-through units, whose activation is denoted by x_i. The "hidden layer" and the "output layer" are made up of units whose activation is respectively:

$$y_j = f[p_j] = f[\Sigma_i[w_{ji}\ x_i]] \qquad \text{and} \qquad v_q = f[p_q] = f[\Sigma_j[w_{qj}\ y_j]] \qquad \text{Eq. 13.37}$$

where y_j is the activation of the hidden unit j, p_j is the "activation potential" of the hidden unit j, w_{ji} is the weight between the input unit i and the hidden unit j, v_q is the activation of the output unit q, p_q is the activation potential of the output unit q, and w_{qj} is the weight between the hidden unit j and the output unit q. f[.] is the "transfer function" of the units, for example a linear function or the sigmoidal function σ[.]:

$$\sigma[p] = 1 / (1 + \exp[-p]) \qquad \text{Eq. 13.38}$$

The error back-propagation algorithm updates the weights so that the network approximates a function for which some input and output patterns are known ("training set"). Let $\mathbf{x}_k$ be the input pattern (vector of real numbers) and $\mathbf{v}^d_k$ the output pattern ("teaching output") of the element k of the training set.

At the beginning the weights of the network are set randomly within a small interval. The idea of the learning algorithm is that the weights should be changed in correspondence to a given input pattern $\mathbf{x}_k$ of the training set, in order to diminish the distance between the output pattern returned by the network, $\mathbf{v}_k$, and the teaching output $\mathbf{v}^d_k$. For each training element k an error E_k is defined as follows (for simplicity the index k for the network's elements is omitted):

$$E_k = 1/2\ \Sigma_q[(v_q - v^d_q)^2] \qquad \text{Eq. 13.39}$$

where v^d_q is the teaching output for the output unit q. If a weight changes, the error changes. In order to decrease the error E_k, the backpropagation algorithm updates each weight in proportion to the error's change caused by that weight's change ("hill climbing"). This is done by updating the weights in proportion to the partial derivative of the error with respect to the weight. The Widrow-Hoff formula (Widrow and Hoff, 1960) is used to update the weights w_{qj}:

$$\Delta w_{qj} = - \eta\ (\partial E_k / \partial\ w_{qj}\ ) = - \eta\ (v_q - v^d_q)\ f'[p_q]\ y_j \qquad \text{Eq. 13.40}$$

where η is a learning rate and f'[.] is the derivative of the transfer function. If the transfer function f[.] is the sigmoidal function, then $f'[p] = \sigma'[p] = \sigma\ [p]\ (1 - \sigma[p])$. If f[.] is linear then f'[.] = 1 and the formula assumes the following form, called "delta rule":

$$\Delta w_{qj} = - \eta\ (\partial E_k / \partial w_{qj}) = - \eta\ (v_q - v^d_q)\ y_j \qquad \text{Eq. 13.41}$$

To apply the same principle to the weights w_{ji} between the input and hidden layer, the derivative $(\partial E_k / \partial w_{ji})$ of the error E_k with respect to the same weights is needed. Assuming an the error $(y_j - y^d_j)$ for the hidden units, if we use the Widrow-Hoff formula to update the weights w_{ji} we have:

$$\Delta w_{ji} = - \eta\ (y_j - y^d_j)\ f'[p_j]\ x_i \qquad \text{Eq. 13.42}$$

Given that the error $(y_j - y^d_j)$ for the hidden units is unknown, it is substituted with the derivative of the error E_k with respect to y_j:

$$\Delta w_{ji} = - \eta\ (\partial E_k / \partial y_j)\ f'[p_j]\ x_i = - \eta\ (\Sigma_q[(v_q - v^d_q)\ f'[p_q]\ w_{qj}])\ f'[p_j]\ x_i \qquad \text{Eq. 13.43}$$

13.3.2 Mixture of Experts Neural Networks

Mixture of experts neural networks (Jacobs et al., 1991; Haykin, 1998) are networks with a modular architecture based on a set of "expert networks" and a "gating network", and trained with a supervised learning algorithm. A simple example of these networks is presented in Figure 13.3. For simplicity the experts of this architecture have no hidden layer and only one output unit, but in general they can have any kind of feed-forward architecture and any number of output units. The idea at the basis of this kind of networks is that during training each expert should specialise on a sub-region of the input-output space, while the gating network should learn to decide which expert is competent for which sub-region.

The gating network has a number of output-units equal to the number of experts. The output units of the experts, v_k, and the gating network's output units, o_k, are linear:

$$v_k = \Sigma_j[w_{kj}\, y_j] \qquad o_k = \Sigma_j[z_{kj}\, y_j] \qquad \text{Eq. 13.44}$$

where y_j is the activation of the input unit j, w_{kj} is the weight of the expert k and input unit j, and z_{kj} is weight of the gating network's output unit k and input unit j. The output V of the whole network is calculated by "mixing" the output of the experts as follow:

$$V = \Sigma_k[v_k\, g_k] \qquad \text{Eq. 13.45}$$

where g_k is a transformation of the gating network's output unit k. In particular g_k is computed through the "softmax activation function" on the basis of o_k, as follows:

$$g_k = \exp[o_k]/\Sigma_f[\exp[o_f]] \qquad \text{where: } \Sigma_k\, g_k = 1 \qquad \text{Eq. 13.46}$$

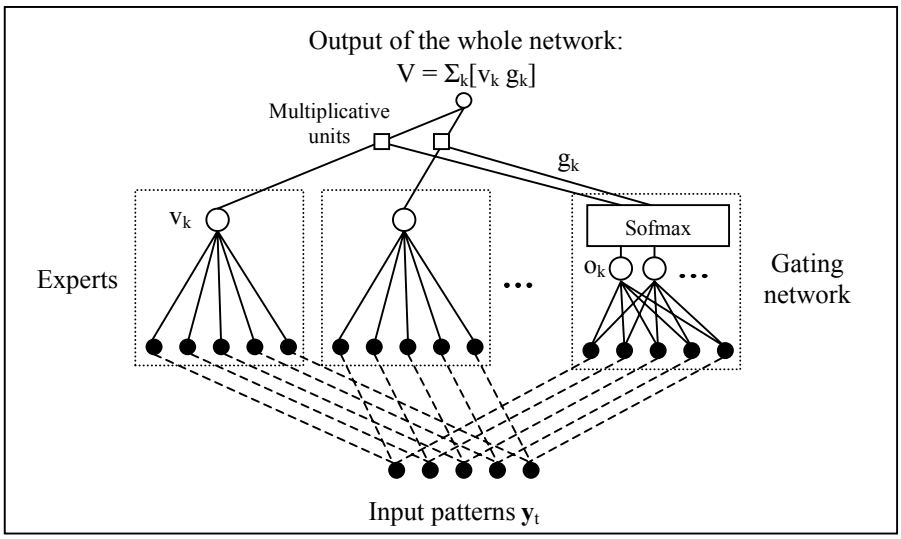


Figure 13.3: Example of architecture of a mixture of experts neural network. Full circles are pass-through units and dotted connections simply copy the signal into the downstream units (introduced for graphical reasons). Empty circles are linear units. Empty squares are multiplicative units.

In order to train the network, each expert is thought to produce a probabilistic output instead of a deterministic one. In particular it is assumed that this output has a Gaussian probability distribution centred on v_k. On the basis of this distribution, whose centre depends on the input **y** and expert's weights $\mathbf{w}_k$, it is possible to compute the *likelihood* l[.] of the desired output v^d:

$$l[v^d \mid \mathbf{w}_k, \mathbf{y}] = (1/(2\pi)^{1/2}) \exp[-1/2\, (v^d - v_k)^{\,2}] \qquad \text{Eq. 13.47}$$

The logarithmic likelihood (the logarithm simplifies computations) of v^d with respect to the distribution of the output of the *whole* network is:

$$L[v^d \mid \mathbf{w}, \mathbf{z}, \mathbf{y}] = \ln l[v^d \mid \mathbf{w}, \mathbf{z}, \mathbf{y}] = \ln[\Sigma_k[g_k\, (1/(2\pi)^{1/2}) \exp[-1/2\, (v^d - v_k)^2]]] \qquad \text{Eq. 13.48}$$

where **w** are the weights of all experts and **z** are the weights of the gating network. In this formula **g** should be interpreted as probabilities ("a-priori probabilities"). It is possible to

compute how this likelihood changes when the weights of the experts and the gating network change (gradients):

$$\partial L/\partial \mathbf{w}_k = h_k (v^d - v_k)\, \mathbf{y} \qquad \partial L/\partial \mathbf{z}_k = (h_k - g_k)\, \mathbf{y} \qquad \text{Eq. 13.49}$$

where h_k, named "a posteriori probabilities", are defined as follows:

$$h_k = (g_k \exp[-1/2\ (v^d - v_k)^2]) / (\Sigma_f [g_f \exp[-1/2\ (v^d - v_f)^2]]) \qquad \text{Eq. 13.50}$$

Similarly to what has been done for the backpropagation algorithm, the gradients can be used to iteratively adjust the weights to increase the likelihood of producing the desired output ("hill climbing"):

$$\Delta \mathbf{w}_k = \eta\, h_k (v^d - v_k)\, \mathbf{y} \qquad \Delta \mathbf{z}_k = \zeta\, (h_k - g_k)\, \mathbf{y} \qquad \text{Eq. 13.51}$$

where η and ζ are learning parameters.

13.3.3 The Generalisation Property of Neural Networks

Neural networks are capable of generalisation. This means that they can produce appropriate outputs if presented with input patterns never seen before, but similar to some patterns with which they have been trained (Hinton et al., 1986; Rolls and Treves, 1998, p. 198, p. 29). Generally speaking, the generalisation property of neural networks is caused by the fact that when the weights are updated to improve an input-output association, these changes influence other input-output associations, possibly improving similar input-output associations in terms of error.

Closely related to the generalisation capacity is the capacity to isolate "common structure" underlying different "problems" (cf. McClelland et al., 1995) and to compress information into the same weights. Here a "problem" is intended as a particular set of input-output associations to learn. "Common structure" is any correlation that may exist between the input-output associations of one problem and the input-output associations of another problem. If some of these correlations are present, it means that the whole set of associations is partially redundant and that it is possible to store it in a compressed form. This is precisely what neural networks do when they are repeatedly trained on the same input-output sets of associations (cf. Elman and Plunkett., 1997).

14 References

14.1 Candidate's Publications During the PhD Research

The research carried out during the three years of PhD study has led to the production of the published papers listed below. The papers that present simulations and results *substantially* different from the ones presented in this thesis are marked with an asterisk * or with a double asterisk **.

The papers marked with one asterisk refer to a piece of research that the author has done at the beginning of the PhD before focussing on planning. This research has investigated "cultural evolution in multi-agent systems", and has been carried out by using reinforcement learning, genetic algorithms (Mitchell, 1996) and imitation (the latter has been simulated through the error backpropagation algorithm, Rumelhart et al., 1986). The publications marked with a double asterisk have investigated the biological aspects of some models presented in the thesis. Both pieces of research have not been included in the thesis because they are too heterogeneous with respect to the topic of planning.

* Baldassarre G., Parisi D. (1999). Individual Learning, Noise and Selection in Cultural Evolution – A Study through Artificial Life Simulations. In Dautenhahn K., Nehaniv C. (eds.), *Proceedings of the Symposium on Imitation in Animals and Artifacts - AISB '99*, pp. 32-37. Brighton: The Society for the Study of Artificial Intelligence and Simulation of Behaviour.

** Baldassarre G., Parisi D. (2000). Classical and instrumental conditioning: From laboratory phenomena to integrated mechanisms for adaptation. In Meyer J-A., Berthoz A., Floreano D., Roitblat H., Wilson S.W. (eds.), *From Animals to Animats 6: Proceedings of the 6th International Conference on the Simulation of Adaptive Behaviour (SAB-2000) - Supplement Volume*, pp. 131-139. Honolulu: International Society for Adaptive Behaviour.

Baldassarre G. (2000). Needs and motivations as mechanisms of learning and control of behaviour: Interference problems with multiple tasks. In Trappl R. (ed.), *Cybernetics and Systems 2000 - Proceedings of the Fifteenth European Meeting on Cybernetics and Systems Research*, pp. 677-682. Vienna: Austrian Society for Cybernetic Studies.

* Baldassarre G. (2001a). Cultural evolution of "guiding criteria" and behaviour in a population of neural-network agents. Journal of Memetics - Evolutionary Models of Information Transmission, Vol. 4.
http://www.cpm.mmu.ac.uk/jom-emit/2001/vol4/baldassarre_g.html

** Baldassarre G. (2001b). A Modular Neural-Network Model of the Basal Ganglia's Role in Learning and Selecting Motor Behaviours. In Altmann E.M., Cleermans A., Schunn C.D., Gray W.D. (Eds.). *Proceedings of the Fourth International Conference on Cognitive Modeling (ICCM-2001)*. Pp. 37-42. Mahwah, NJ.: Lawrence Erlbaum.

Baldassarre G. (2001c). A Planning Modular Neural-Network Robot for Asynchronous Multi-Goal Navigation Tasks. In Arras K.O., Baerveldt A.-J, Balkenius C., Burgard W., Siegwart R. (eds.), *Proceedings of the 2001 Fourth European Workshop on Advanced*

Mobile Robots (EUROBOT-2001), pp. 223-230. Lund, Sweden: Lund University Cognitive Studies.

Baldassarre G. (2001d), Coarse Planning for Landmark Navigation in a Neural-Network Reinforcement Learning Robot, *IROS-2001 Proceedings of the International Conference on Intelligent Robots and Systems*, IEEE.

** Baldassarre G. (2001e), A modular neural-network model of the basal ganglia's role in learning and selecting motor behaviours, *Cognitive Systems Research*, v. 3, pp. 1-13.

** Baldassarre G. (2001f), Limiti di efficacia temporale del condizionamento operante: un modello connessionistico, In Pinna B. (ed.), *Congresso nazionale della sezione di psicologia sperimentale*, pp. 59-61, Sassari, EDES Editrice Democratica Sarda.

**Baldassarre G. (2002), A biologically plausible model of human planning based on neural networks and Dyna-PI models, In Butz M., Sigaud O., Gerard P. (eds.), *Proceedings of the Workshop on Adaptive Behaviour in Anticipatory Learning Systems – ABiALS-2002 (hold within SAB-2002)*, pp. 40-60. Wurzburg: University of Wurzburg

14.2 References

Albus J.S. (1981). *Brain, Behaviour, and Robotics*. Peterborough, NH: Byte Books.

Allen J., Hendler J., Tate A. (eds.) (1990). *Readings in Planning*. Palo Alto, CA: Morgan Kaufmann.

Ambros-Ingerson J.A., Steel S. (1988). Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-1988)*, pp. 735-740. St. Paul, Minnesota: Morgan Kaufmann.

Arkin R.C. (1989). Navigation path planning for a vision-based mobile robot. *Robotica*, vol. 7, pp. 49-63.

Arkin R.C. (1998). *Behaviour-Based Robotics*. Cambridge, MA: The MIT Press.

Barto A.G. (1994). Adaptive critics and the basal ganglia. In Houk J.C., Davis J.L., Beiser D.G. (eds.), *Models of Information Processing in the Basal Ganglia*, pp. 249-270. Cambrigde, MA: The MIT Press.

Barto A.G., Bradtke S.J., Singh S.P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, Special Volume: Computational Research on Interaction and Agency, vol. 72, pp. 81-138.

Barto A.G., Sutton R.S., Anderson C.W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 13, pp. 834-846.

Barto A.G., Sutton R.S., Watkins C.J.C.H. (1990). Learning and Sequential Decision Making. In Gabriel M., Moore J.W. (eds), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 539-602. Cambridge, MA: The MIT Press.

Bellman R.E. (1957). *Dynamic Programming*. Princeton: Princeton University Press.

Bertsekas D.P. (1995). *Dynamic Programming and Optimal Control*. Belmont, MA.: Athena Scientific.

Blanzieri E., Katenkamp P. (1996). Learning radial basis function networks on-line. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 37-45. San Francisco, CA: Morgan Kaufmann.

Boutilier C., Dearden R., Goldszmidt M. (2000). Stochastic dynamic programming with factor representations. *Artificial Intelligence*, vol. 121, pp. 49-107.

Brooks R.A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, vol 2, pp. 14-23.

Calabretta R., Nolfi S., Parisi D., Wagner G. P. (1998). Emergence of functional modularity in agents. In Pfeiffer R. (ed.), *From Animals to Animats 5: Proceedings of the 5th International Conference on the Simulation of Adaptive Behaviour (SAB-1998),* pp. 497-504. Cambridge, MA: The MIT Press.

Caruana R. (1995). Learning many related tasks at the same time with backpropagation. In Tesauro G., Touretzky D.S., Leen T.K. (eds.), *Advances in Neural Information Processing Systems 7*, vol. 7, pp. 657-664. Cambridge, MA: The MIT Press.

Chapman D., Kaelbling L.P. (1991). Input generalisation in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-1991)*, pp. 726-731. San Mateo, CA: Morgan Kaufmann.

Dearden R. (2000). *Learning and planning in structured worlds*. PhD thesis. Vancouver: Department of Computer Science, University of British Columbia.

Dearden R. (2001). Structured prioritised sweeping. *Proceedings of the Eighteenth International Conference on Machine Learning (ICML-2001)*. Pp. 82-89.

Dechter R., Pearl J. (1985). Generalized Best-First Search and the Optimality of A*. *Journal of the Association for Computing Machinery*, vol. 32 (3), pp. 505-536.

Doran J (1966). *Doubletree Searching and the Graph Traverser*. Research Memorandum EPU-R-22. Edinburgh: Department of Machine Intelligence and Perception, Edinburgh University.

Draper D., Hanks S., Weld D. (1994). Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference in Artificial Intelligent Planning Systems (AIPS)*. San Mateo: Morgan Kaufmann.

Duckett T., Nehmzow U. (1999). Exploration of unknown environments using a compass, topological map and neural networks. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*. Monterey, CA: IEEE Press.

Elman J.L., Plunkett K. (1997). *Exercises in Rethinking Innateness: A Handbook for Connectionist Simulations*. Cambridge, MA: MIT Press.

Elman, J.L. (1990). Finding structure in time. *Cognitive Science*, vol. 14, pp. 179-211.

Fikes R.E., Nilsson N.J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, vol. 2, pp. 189-208.

Fleuret F., Brunet E. (2000). DEA: An architecture for goal planning and classification. *Neural Computation*, vol. 12, pp. 1987-2008.

Fomin T., Rozgonyi T., Szepesv'ari C., Lorincz A. (1996). Self-organizing multi-resolution grid for motion planning and control. *International Journal of Neural Sciences*, vol. 7, pp. 757-776.

Gatt E. (1992). Integrating planning and reaction in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the tenth national conference on artificial intelligence (AAAI-1992)*, pp. 809-815. San Jose, CA: AAAI Press.

Ginsberg M.L. (1989). Universal planning: An (almost) universally bad idea. *AI Magazine*, vol. 10 (4), pp. 40-44.

Hampson S. (1998). Connectionist Problem Solving. In Arbib M.A. (ed.), *The Handbook of Brain Theory and Neural Networks*, pp. 756-760. Cambridge, MA: The MIT Press.

Harnad S. (1990). The symbol grounding problem. *Phisica D*, vol. 42, pp. 335-346.

Harnad S. (1993). Grounding symbols in the analog world with neural nets. *Think*, vol. 2, pp. 12-78.

Hart P.E., Nilsson N.J., Raphael B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC 4 (2), pp. 100-107.

Haykin S. (1999). *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ: Prentice Hall.

Hebb D.O. (1949). *The Organisation of Behavior*. New York: Wiley.

Hinton G.E., McClelland J.L., Rumelhart D.E. (1986). Distributed representations. In Rumelhart D.E., McClelland J.L., and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: The MIT Press.

Hopfield J.J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, vol. 79, pp. 2554-2558.

Houk J.C., Adams J.L., Barto A.G. (1994). A model of how the basal ganglia generate and use neural signals that predict reinforcement. In Houk J.C., Davis J.L., Beiser D.G. (eds.), *Models of Information Processing in the Basal Ganglia*, pp. 249-270. Cambrigde, MA: The MIT Press.

Humphrys M. (1996). Action selection methods using reinforcement learning. In Maes P., Mataric M.J., Wilson S.W. (eds.), *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour (SAB-1996)*. Cambridge, MA: The MIT Press.

Jaakkola T., Singh S.P., Jordan M.I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro G., Touretzky D.S., Leen T.K. (eds.), *Advances in Neural Information Processing Systems 7 (NIPS-1994)*, pp. 345-352. San Mateo, CA: Morgan Kaufmann.

Jacobs R.A., Jordan M.I. (1991). A competitive modular connectionist architecture. In *Advances in Neural Information Processing Systems*, vol. 3, pp. 767-773. San Mateo, CA: Morgan Kaufmann.

Jacobs R.A., Jordan M.I., Nowlan S.J., Hinton G.E. (1991). Adaptive mixtures of local experts. *Neural Computation*, vol. 3, pp. 79-87.

Jakobi N., Husbands P., Harvey I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In Moran F., Moreno A., Merelo J., Chacon P. (eds.), *Proceedings of the Third European Conference on Artificial Life*, pp. 704-720. Berlin: Springer-Verlag.

Jokoo M., Ishida T. (1999). Search algorithms for agents. In Weiss G. (ed.), *Multiagent Systems*, pp. 165-197. Cambridge, MA: The MIT Press.

Kaelbling L.P., Littman L.M., Moore A.W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research,* vol. 4, pp. 237-285.

Kanerva P. (1988). *Sparse Distributed Memory*. Cambridge, MA: The MIT Press.

Kohonen T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, vol. 43, pp. 59-69.

Korf R.E. (1985a). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, vol. 27 (1), pp. 97-109.

Korf R.E. (1985b). Iterative-deepening A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligecne (IJCAI-1985)*, pp. 1034-1036. Morgan Kaufmann.

Korf R.E. (1988). Optimal path finding algorithms. In Kanal L.N., Kumar V. (eds.), *Search in Artificial Intelligence,* pp. 223-267. Springer-Verlag: Berlin.

Korf R.E. (1990). Real-time heuristic search. *Artificial Intelligence*, vol. 42 (2-3), pp. 189-211.

Kortenkamp D., Chown E. (1992). A directional spreading-activation network for mobile robot navigation. In Meyer J.-A., Roitblat H.L., Wilson S.W. (eds.), *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB-1992),* pp. 218-224. Cambridge, MA: The MIT Press.

Kushmerick N., Hanks S., Weld D. (1994). An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1073-1078. AAAI Press.

Lee T., Nehmzow U., Hubbold R. (1998). Mobile robot simulation by means of acquired neural network models. In Zobel R., Moeller D. (eds.), *Proceedings of the Twelfth European Simulation Multiconference* (*ESM-1998).* Manchester, UK: SCS.

Lei G. (1990). A neural model with fluid properties for solving labyrinthian puzzle. *Biological Cybernetics*, vol. 64 (1), pp. 61-67.

Levenick J.R. (1991). NAPS: A connectionist implementation of cognitive maps. *Connection Science*, vol. 3, pp. 107-126.

Lin L.J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, vol. 8, pp. 293-391.

Lin L.J. (1993). Hierarchical learning of agent skills by reinforcement. In Ruspini E. H. (ed.), *IEEE - Proceedings of the International Conference on Neural Networsks*, pp. 181-186. New York, NY: IEEE Press.

Lin L.J., Mitchell T.M. (1992). Reinforcement learning with hidden states. In Meyer J-A, Roitblat H.L., Wilson S.W. (eds.), *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour (SAB-1992)*, pp. 271-280. Cambridge, MA: The MIT Press.

Lin L.-J., Mitchell T.M. (1992). *Memory approaches to reinforcement learning in non-Markovian domains*. Technical Report CMU-CS-92-138. Pittsburgh, PA: School of Computer Science, Carnegie-Mellon University.

Linaker F. (2001). From Time-Steps to Events and Back. In Arras K.O., Baerveldt A.-J, Balkenius C., Burgard W., Siegwart R. (eds.), *Proceedings of the 2001 Fourth European Workshop on Advanced Mobile Robots (EUROBOT-2001),* pp. 223-230. Lund, Sweden: Lund University Cognitive Studies.

Linden T.A. (1991). Representing software designs as partially developed plans. In Lowry M.R., McCartney R.D. (eds.), *Automating Software Design*, pp. 603-625. Cambridge, MA: The MIT Press.

Lorincz A., Polik I., Szita I. (2001). *Event-learning and robust policy heuristics*. Technical report, NIPT-ELU-14-05-2001. Budapest: Department of Information Systems, Eotvos Lorand University.

Ma Z.F. Doran J.F. (1993). *CADDIE and its Multi-agent Planner*. Unpublished Paper.

Maes P. (1989). The Dynamics of Action Selection. *Proceedings of the AAAI-1989 Spring Symposium on Limited Rationality*. AAAI Press.

Maes P. (1991). A Bottom-Up Mechanism for Action Selection in an Artificial Creature. In Wilson S. Arcady-Meyer J. (eds.), *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behaviour (SAB-1990),* pp.238-246. Cambridge, MA.: The MIT Press.

Maes Pattie (1990). Situated agents can have goals. *Journal for Robotics and Autonomous Systems*, vol. 6 (1), pp. 49-70.

Mataric M.J. (1991). Navigating with a rat brain: A neurobiologically-inspired model for robot spatial navigation. In Meyer J.A., Wilson S.W. (eds.), *From Animals to Animats:*

Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB-1990), pp. 169-175. Cambridge, MA.: The MIT Press.

McCallum R.A. (1993). Overcoming incomplete perception with utile distinction memory. In Utgoff P. (ed.), *Proceedings of the Tenth International Conference in Machine Learning*, pp. 190-196. San Mateo, CA: Morgan Kaufmann.

McClelland J.L., McNaughton B.L., O'Reilly R.C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, vol. 102, pp.419-457.

McClelland, J.L., Rumelhart, D.E., Hinton, G. E. (1986). The appeal of Parallel Distributed Processing. In Rumelhart D.E., McClelland J.L., and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: The MIT Press.

McDonald M.A.F., Hingston P. (1994). *Approximate discounted dynamic programming is unreliable*. Technical report 94/6. Department of Computer Science, The University of Western Australia.

McGovern A., Precup D., Ravindran B., Singh S., Sutton R.S. (1998). Hierarchical optimal control of MDPs. *Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems,* pp. 186-191.

McGovern A., Sutton R.S., Fagg A.H. (1997). Roles of macro-actions in accelerating reinforcement learning. *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing,* pp. 13-17.

Meyer J.-A., Berthoz A., Floreano D. (eds.) (2000). *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior (SAB 2000)*. Cambridge, MA: The MIT Press.

Meyer J.-A., Guillot A. (1990). Simulation of adaptive behaviour in animats: review and prospect. In Meyer J.-A. and Wilson S.W. (eds.), *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB-1990),* pp. 2-14. Cambridge, MA: The MIT Press.

Miglino O., Lund H.H., Nolfi S. (1995). Evolving mobile robots in simulated and real environments. *Artificial Life*, vol. 2, pp. 417-434.

Miller T.W., Glanz F.H., Kraft G.L. (1990). CMAC: An associative neural network alternative to backpropagation. In Hanson S.J., Cowan J.D., Giles C.L. (eds.), *Proceedings of IEEE*, pp. 1561-1567. San Mateo, CA: Morgan Kaufmann.

Miller T.W., Sutton R.S., Werbos P.J. (1990). *Neural networks for control*. Cambridge, MA: The MIT Press.

Mitchell M. (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Mitchell T.M (1990). Becoming increasingly reactive. *In Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-1990),* pp. 1051-1058. Boston, MA: AAAI Press.

Moore A.W., Atkeson C.G. (1993). Prioritised sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, 103-130.

Morasso P., Vercelli G., Zaccaria R. (1992). Hybrid systems for robot planning. In Aleksander I., Taylor J., (eds.), *Artificial Neural Networks 2*, pp. 691-697. Amsterdam: North-Holland/Elsevier Science Publishers.

Murphy R. (2000). *An Introduction to AI Robotics - Intelligent Robotics and Autonomous Agents*. Cambridge, MA: The MIT Press.

Neal R.M. (1995). *Stochastic feed-forward neural networks*. PhD Thesis. Toronto: Graduate Department of Computer Science, University of Toronto.

Neal R.M. (1996). *Bayesian learning for neural networks*. Berlin: Springer-Verlag.

Nehmzow U. (2001). Quantitative analysis of robot-environment interaction - On the difference between simulation and the real. In Arras K.O., Baerveldt A.-J, Balkenius C., Burgard W., Siegwart R. (eds.), *Proceedings of the 2001 Fourth European Workshop on Advanced Mobile Robots (EUROBOT-2001),* pp. 223-230. Lund, Sweden: Lund University Cognitive Studies.

Nehmzow U., Hallam J., Smithers T. (1989). Really Useful Robots. In Kanade T., Groen F.C.A., Hertzberger L.O. (eds.), *Proceedings of IAS 2 - Intelligent Autonomous Systems*, pp. 284--293. Amsterdam.

Nehmzow U., Smithers T., Hallam J. (1991). Location recognition in a mobile robot using self-organising feature maps. In Schmidt G. (ed.), *Information Processing in Autonomous Mobile Robots*. Berlin: Springer Verlag.

Nolfi S., Elman J.L., Parisi D. (1994). Learning and evolution in neural networks. *Adaptive Behavior*, vol. 3, pp. 5-28.

Nolfi S., Tani J. (1999). Extracting regularities in space and time through a cascade of prediction networks: The case of a mobile robot navigating in a structured environment. *Connection Science*, vol. 11(2), pp. 129-152.

Noreils F., Chatila R. (1995). Plan execution monitoring and control architecture for mobile robots. *IEEE Transactions on Robotics and Automation*, vol. 11, pp. 255-266.

Pohl I. (1971). Bi-directional Search. In Meltzer B., and Michie, D. (eds.), *Machine Intelligence 6*, pp. 127-140. New York: American Elsevier.

Puterman M.L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. New York: John Wiley & Sons.

Ramamurti V., Ghosh J. (1997). *Structurally adaptive modular networks for non-stationary environments*. Technical report TX 78712-1084. Austin, Texas: Department of Electrical and Computer Engineering, University of Texas.

Revel A., Gaussier P., Lepretre S. and Banquet J.P (1998). Planification Versus Sensory-Motor Conditioning: What Are the Issues? In Pfeiffer R. (ed.), *From Animals to Animats 5: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour (SAB-1998)*, pp. 271-280. Cambridge, MA: The MIT Press.

Reynolds S.I. (2002). *Experience stack reinforcement learning for off-policy control*. Technical report CSRP-02-1. Birmingham: School of Computer Science, University of Birmingham.

Rojas R. (1996). *Neural Networks - A Systematic Introduction*. Berlin: Springer-Verlag.

Rolls E., Treves A. (1998). *Neural Networks and Brain Function*. Oxford: Oxford University Press.

Ross S. (1983). *Introduction to stochastic dynamic programming*. New York, NY: Academic Press.

Rumelhart D.E., Hinton G.E., Williams R.J. (1986). Learning representations by backpropagation errors. In Rumelhart D.E., McClelland J.L., and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, pp. 318-362. Cambridge, MA: The MIT Press.

Rumelhart D.E., McClelland J.L. (1986). A distributed model of human learning and memory. In Rumelhart D.E., McClelland J.L., and the PDP Research Group (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* vol. 2, pp. 170-215. Cambridge, MA: The MIT Press.

Rumelhart D.E., McClelland J.L. and the PDP Research Group (eds.) (1986). *Parallel Distributed Processing: Explorations in the microstructure of cognition,* vol. 1. Cambridge, MA: The MIT Press.

Rummery G.A., Niranjan M. (1994). *On-line Q-Learning using connectionist systems*. Technical Report CUED/F-INFENG/TR 166. Cambridge: Engineering Department, Cambridge University.

Russell S., Norvig P. (1995). *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.

Sacerdoti E.D.(1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, vol. 5, pp. 115-135.

Sacerdoti E.D. (1977). *A Structure for Plans and Behavior*. New York: Elsevier.

Samejima K., Omori T. (1999). Adaptive internal state space construction method for reinforcement learning of a real-world agent. *Neural Networks*, vol. 12, pp. 1143-1155.

Schmajuk N.A., Blair H.T. (1993). Place learning and the dynamics of spatial navigation: A neural network approach. *Adaptive Behaviour*, Vol. 1-3, pp. 353-385.

Schmidhuber J. (1992). Learning unambiguous reduced sequence descriptions. In Moody J.E., Hanson S.J., Lippman R.P. (eds.), *Advances in Neural Information Processing Systems 4 (NIPS-1992)*, pp. 291-298. San Mateo, CA: Morgan Kaufmann.

Schmidhuber J. (1999). Artificial curiosity based on discovering novel algorithmic predictability through coevolution. In Angeline P., Michalewicz X., Schoenauer M., Yao X., Zalzala Z. (eds.), *Congress on Evolutionary Computation*, pp. 1612-1618. Piscataway, NJ: IEEE Press.

Schmidhuber J., Wahnsiedler R. (1992). Planning simple trajectories using neural subgoal generators. In Meyer J.-A., Wilson S.W. (eds.), *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB-1992)*, pp. 196-202. Cambridge, MA: The MIT Press.

Schoppers M.J. (1987). Universal plans for reactive robots in unpredictable environments. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI-1987),* pp. 1039-1046. San Mateo, CA: Morgan Kaufmann.

Schoppers M.J. (1989). In defence of reaction plans as caches. *AI Magazine*, vol. 10, pp. 51-60.

Sharkey N.E., Sharkey A.J.C. (1995). An analysis of catastrophic interference. *Connection Science,* vol. 7, pp. 301-329.

Singh S.P., Jaakkola T.S., Jordan M.I. (1994). Learning without state-estimation in partially observable Markovian decision processes. In Cohen W., Hirsh H. (eds.), *Proceedings of eleventh international conference on machine learning*, pp. 284-292. New Brunswick, NJ: Morgan Kaufmann.

Sondik E.J. (1978). The optimal control of partially observable Markov processes over the infinite horizon: Discounted case. *Operations Research*, vol. 26, pp. 282-304.

Steels L. (1994). The artificial life roots of artificial intelligence. *Artificial Life*, vol. 1, pp. 75-110.

Steels L., Brooks R. (eds.) (1995). *The Artificial Life Route to Artificial Intelligence: Building Embodied, Situated Agents*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Sun R. (2000). Symbol grounding: A new look at an old idea. *Philosophical Psychology*, vol. 13, pp. 149-172.

Sun R., Peterson T. (1998). A hybrid model for learning sequential navigation. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA-1997)*, pp. 234-239. Monterey, CA: IEEE Press.

Sutton R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceeding of the Seventh International Conference on Machine Learning*, pp. 216-224. San Mateo, Ca.: Morgan Kaufmann.

Sutton R.S. (1991). Dyna, an integrated architecure for learning, planning, and reacting. In *Working Notes of the 1991 AAAI Spring Symposium*, pp. 151- 155. AAAI Press.

Sutton R.S. (1996). Generalization in reinforcement learning: successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8 (NIPS-1996),* pp. 1038-1044. Cambridge MA: The MIT Press.

Sutton R.S., Barto A.G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Sutton R.S., Barto A.G., (1990). Time-derivative models of Pavlovian reinforcement. In Gabriel M., Moore J. (eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 497-537. Cambridge, MA: The MIT Press.

Sutton R.S., McAllester D., Singh S., Mansour Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In Solla S., Leen T, Muller K.-R. (eds.), *Advances in Neural Information Processing Systems 12 (NIPS-1999)*, pp. 1057-1063. Cambridge, MA: The MIT Press.

Sutton R.S., Precup D., Singh S. (1998). *Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales*. Technical report. Amherst, MA: Department of Computer and Information Science, University of Massachusetts.

Sutton R.S., Singh S., Precup D., Ravindran B. (1999). Improved switching among temporally abstract actions. *Advances in Neural Information Processing Systems 11 (NIPS-1998).* Cambridge, MA: The MIT Press.

Sutton R.S., Whitehead S.D. (1993). Online learning with random representations. In Utgoff P. (ed.), *Proceedings of the Tenth International Conference on Machine Learning (ICML-193)*. San Mateo, CA: Morgan Kaufmann.

Tani J. (1996). Model-Based Learning for Mobile Robot Navigation from the Dynamical Systems Perspective. *IEEE Transactions in System, Man and Cybernetics*, Part B, vol. 26 (3), pp. 421-436.

Tani J., Nolfi S. (1999). Learning to perceive the world as articulated: An approach for hierarchical learning in sensory-motor systems. *Neural Networks*, vol. 12, pp. 1131-1141.

Thrun S. (1992). *Efficient exploration in reinforcement learning*. Technical Report CMU-CS-92-102. Pittsburgh, PA: School of Computer Science, Carnegie-Mellon University.

Thrun S. B., Moller K., Linden A. (1991). Planning with an adaptive world model. In Tourtezky D. S., Lippmann R. (eds.), *Advances in Neural Information Processing Systems 3 (NIPS-1990)*, pp. 450-456. San Mateo, CA: Morgan Kaufmann.

Trullier O., Meyer J.-A. (1998). Animat Navigation Using a Cognitive Graph. In Pfeiffer R. (ed.), *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB-1998)*, pp. 213-222. Cambridge, MA: The MIT press.

Tyrrell T. (1993). *Computational Mechanisms for Action Selection*. PhD thesis. Edinburgh: Cognitive Science Department, University of Edinburgh.

Tyrrell, T. (1994). An Evaluation of Maes's Bottom-Up Mechanism for Behaviour Selection. *Adaptive Behavior,* vol. 2, n. 4, pp. 307-348.

Warren D.H.D. (1976). Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference*, pp. 344-354. AISB.

Watkins C.J.C.H. (1989). *Learning from Delayed Rewards*. PhD Thesis. Cambridge, UK: King's College, University of Cambridge.

Watkins C.J.C.H., Dayan, P. (1992). Q-learning. *Machine Learning*, vol. 8, pp. 279-292.

Whitehead S.D., Ballard D.H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, vol. 7, pp. 45-83.

Widrow B., Hoff M.E. (1960), Adaptive switching circuits, *IRE WESCON Convention Record*, Part IV, pp. 96-104.

Wiering M.A., Salustowicz R.P., Schmidhuber J. (1998). CMAC models learn to play soccer. In Niklasson L., Bod'en M., Ziemke T. (eds.), *Proceedings of the Eighth International Conference on Artificial Neural Networks (ICANN-1998)*, pp 443-448. Berlin: Springer-Verlag.

Wiering M.A., Schmidhuber J. (1998). HQ-Learning. *Adaptive Behaviour*, vol. 6, pp. 219-246.

Wuensche A. (1998). Discrete Dynamical Networks and their Attractor Basins. *Complexity International*. Vol. 6, http://www.csu.edu.au/ci/vol06/wuensche/wuensche.html.

Wyatt J. (1997). *Exploration and Inference in Learning from Reinforcement*. PhD thesis. Edinburgh: Department of Artificial Intelligence, University of Edinburgh.

Wyatt J., Hoar J., Hayes G. (1998). Design, analysis and comparison of robot learners. In Nehmzow U., Recce M., Bisset D. (eds), *Robotics and Autonomous Systems - Special Issue on Quantitative Methods in Mobile Robotics*, vol. 24 (nos 1-2), pp.17-32.

Yee R.C., Saxena S., Utgoff P.E., Barto A.C. (1990). Explaining temporal-differences to create useful concepts for evaluating states. In *Proceedings of the Eight National Conference on Artificial Intelligence (AAAI-1990)*, pp. 882-888. San Mateo, CA: Morgan Kaufmann.

Yokoo M., Ishida T. (1999). Search algorithms for agents. In Weiss G. (ed.), *Multiagent Systems*, pp. 165-199. Cambridge, MA: The MIT Press.

Yoshikawa T. (1990). *Foundations of Robotics: Analysis and Control*. Cambridge, MA: The MIT Press.

Zeller M., Sharma R., Schulten K. (1997). Motion planning of a pneumatic robot using a neural network. *IEEE Control Systems Magazine*, vol. 17, pp. 89-98.