Learning from Delayed Rewards

Christopher John Cornish Hellaby Watkins

King's College

Thesis Submitted for Ph.D.

May, 1989

Preface

As is required in the university regulations, this thesis is all my own work, and no part of it is the result of collaborative work with anybody else.

I would like to thank all the people who have given me encouragement and advice, and the many people I have had conversations with about this work. Other people's interest and encouragement has helped me enormously.

In the early stages of my research, Doug Frye introduced me to the work of Piaget, and I had a number of conversations with David Spiegelhalter on inductive inference.

I would like to thank Richard Young, my supervisor, for patiently giving me good advice over a number of years, and for spending a lot of time talking to me about ideas that led up to this work.

Alex Kacelnik has encouraged me greatly, as well as providing many fascinating examples from animal behaviour and foraging theory.

Rich Sutton gave me his technical reports, and pointed out a difficulty with Q-learning.

I have had a number of helpful conversations with Andy Barto, who has raised a number of issues and possibilities that I have not been able to discuss in the thesis. I have also had interesting conversations with, and comments from, Graeme Mitchison, and John McNamara. Andy Barto, Alex Kacelnik, Alastair Houston, and Robert Zimmer have read and made comments on early drafts of some parts of the thesis, and the final version is considerably clearer and more correct as a result.

I am most grateful for the financial support I have received from King's College. I would also like to thank Tony Weaver, my group leader at Philips Research Laboratories, for his unfailing support which has enabled me to complete this work.

Finally, I would like to dedicate this thesis to my wife Lesley, because she is the happiest to see it finished!

Chris Watkins, May 1st, 1989.

Summary

In behavioural ecology, stochastic dynamic programming may be used as a general method for calculating animals' optimal behavioural policies. But how might the animals themselves learn optimal policies from their experience? The aim of the thesis is to give a systematic analysis of possible computational methods of learning efficient behaviour.

First, it is argued that it does follow from the optimality assumption that animals should learn optimal policies, even though they may not always follow them. Next, it is argued that Markov decision processes are a general formal model of an animal's behavioural choices in its environment. The conventional methods of determining optimal policies by dynamic programming are then described. It is not plausible that animals carry out calculations of this type.

However, there is a range of alternative methods of organising the dynamic programming calculation, in ways that are plausible computational models of animal learning. In particular, there is an incremental Monte-Carlo method that enables the optimal values (or 'canonical costs') of actions to be learned directly, without any requirement for the animal to model its environment, or to remember situations and actions for more than a short period of time. A proof is given that this learning method works. Learning methods of this type are also possible for hierarchical policies. Previously suggested learning methods are reviewed, and some even simpler learning methods are presented without proof. Demonstration implementations of some of the learning methods are described.

Table of Contents

Chapter 1
Introduction

Learning to act in ways that are rewarded is a sign of intelligence. It is, for example, natural to train a dog by rewarding it when it responds appropriately to commands. That animals can learn to obtain rewards and to avoid punishments is generally accepted, and this aspect of animal intelligence has been studied extensively in experimental psychology. But it is strange that this type of learning has been largely neglected in cognitive science, and I do not know of a single paper on animal learning published in the main stream of literature on 'artificial intelligence'.

This thesis will present a general computational approach to learning from rewards and punishments, which may be applied to a wide range of situations in which animal learning has been studied, as well as to many other types of learning problem. The aim of the thesis is not to present specific computational models to explain the results of specific psychological experiments. Instead, I will present systematically a family of algorithms that could in principle be used by animals to optimise their behaviour, and which have potential applications in artificial intelligence and in adaptive control systems.

In this introduction I will discuss how animal learning has been studied, and what the requirements for a computational theory of learning are.

1. Classical and Instrumental Conditioning

I will not give any comprehensive review of the enormous literature on the experimental study of animal learning. Instead I will describe the essential aims of the experimental research, the nature of the phenomena studied, and some of the main conclusions.

There is a long history of research into conditioning and associative learning, as described by Mackintosh (1983). Animals' ability to learn has been studied by keeping them in controlled artificial environments, in which events and contingencies are under the control of the experimenter. The prototypical artificial environment is the Skinner box, in which an animal may be confronted with stimuli, such as the sound of a buzzer or the sight of an illuminated lamp, and the animal may perform responses such as pressing a lever in the case of a rat, or pecking at a light in the case of a pigeon. The animal may be automatically provided with *reinforcers*. In behavioural terms, a positive reinforcer is something that may increase the probability of a preceding response; a positive reinforcer might be a morsel of food for a hungry animal, for instance, or a sip of water for a thirsty animal. Conversely, a negative reinforcer, such as an electric shock, is something that may reduce the probabilty of a preceding response. In a typical experiment, the animal's environment may be controlled automatically for a long period of time, and the delivery of reinforcers may be made contingent upon the stimuli presented and on the responses of the animal. The events and contingencies that are specified for the artificial environment are known as the *reinforcement schedule.*

Two principal types of experimental procedure have been used: *instrumental* and *classical* conditioning schedules.

In instrumental schedules, the reinforcement that the animal receives depends on what it does. *Instrumental learning* is learning to perform actions to

obtain rewards and to avoid punishments: the animal learns to behave in a certain way *because* behaving in that way leads to positive reinforcement. The adaptive function of instrumental conditioning in animals is clear: blue tits will obtain more food in winter if they can learn to visit well stocked bird-tables.

In classical (or 'Pavlovian') experiments, the animal is exposed to sequences of events and reinforcers. The reinforcers are contingent on the events, not on the animal's own behaviour: a rat may be exposed to a light, and then given an electric shock regardless of what it does, for example. Experiments of this type are often preferred (Dickinson 1980) because the correlations between events and reinforcers may be controlled by the experimenter, whereas the animal's actions may not.

Classical conditioning experiments depend on the fact that an animal may naturally respond to certain stimuli without any previous learning: a man will withdraw his hand from a pin-prick; a dog will salivate at the sight of food. The stimulus that elicits the response is termed the *unconditioned stimulus* or US. If the animal is placed in an environment in which another stimulus—the conditioned stimulus or CS—tends to occur before the US, so that the occurrence of the CS is correlated with the occurrence of the US, then an animal may produce the response after the CS only. It is as if the animal learns to expect the US as a result of the CS, and responds in anticipation.

Whether there are in fact two types of learning in instrumental and classical conditioning is much disputed, and complex and difficult issues are involved in attempting to settle this question by experiment. However, I will be concerned not with animal experiments but with learning algorithms. I will therefore give only a brief discussion of one interpretation of the experimental evidence from animal experiments, as part of the argument in favour of the type of learning algorithm I will develop later.

Mackintosh (1983) discusses the relationship between classical and instrumental conditioning at length.

First, it might seem tempting to regard classical conditioning as a form of instrumental conditioning: might a dog not learn to salivate in anticipation of food because the dog found that if it salivated the food was more palatable? Mackintosh argues that classical conditioning cannot be explained as instrumental conditioning in this way. One of the neatest experiments is that of (Browne 1976) in which animals were initially prevented from giving any response while they observed a correlation between a stimulus and a subsequent reward. When the constraint that prevented the animals giving the response was removed, they immediately gave the response; there was no possibility of any instrumental learning because the animals had been prevented from responding.

Mackintosh also notes that, perhaps more suprisingly, much apparently instrumental conditioning may be explainable as classical conditioning. In an instrumental experiment in which animals learn to perform some action in response to a conditioned stimulus, the animal must inevitably observe a correlation between the conditioned stimulus and the reward that occurs as a result of its action. This correlation, produced by the animal itself, may give rise to a classically conditioned response: if this classically conditioned response is the same as the instrumental response, then each response will strengthen the correlation between the CS and the reward, thus strengthening the conditioning of the CS. Learning would thus be a positive feedback process: the more reliably the animal responds, the greater the correlation it observes, and the greater the correlation it observes, the more reliably the animal will respond.

But, as Mackintosh argues, not all instrumental learning may be explained in this way. A direct and conclusive argument that not all instrumental learning is explainable as classical conditioning is the common observation that animals

can learn to perform different responses in response to the same stimuli to obtain the same rewards.

Mackintosh suggests, however, that no instrumental conditioning experiment is totally free of classically conditioned effects, and vice versa. For instrumental learning to occur, an animal must produce at least one response, or sequence of responses, that results in a reward—why does the animal produce the first such response? Instrumental learning consists of attempting to repeat previous successes; the achievement of the first success requires a different explanation. One possibility is that an animal performs a totally random exploration of its environment: but this cannot be accepted as a complete explanation. A reasonable hypothesis is that classical conditioning is the expression of innate knowledge of what actions are usually appropriate when certain types of events are observed to be correlated in the environment. The roughly appropriate innate behaviour released by classical conditioning may then be fine-tuned by instrumental learning. The question of the relationship between classical and instrumental conditioning is, therefore, one aspect of a more fundamental question: what types of innate knowledge do animals have, and in what ways does this innate knowledge contribute to learning?

Conditioning theory seeks to explain animals' behaviour in detail: to explain, for example, just how the time interval between a response and a reinforcer affects the rate at which the response is learned. As a consequence of this level of detail, conditioning theory cannot readily be used to explain or to predict animal learning under more natural conditions: the relationships between stimuli, responses, and reinforcers become too complex for models of instrumental conditioning to make predictions. It is clear in a general way that instrumental conditioning could enable an animal to learn to obtain what it needs, but conditioning theory cannot in practice be used to predict the results of learning

quantitatively under most natural conditions. The difficulty is that conditioning theory has tended to be developed to explain the results of certain types of experiment, rather than to predict the effect of learning on behaviour overall.

The optimality argument as used in behavioural ecology can provide, I believe, a clear and well motivated description of what instrumental learning should ideally be. I will later consider systematically the different ways in which this type of instrumental learning may be achieved.

2. The Optimality Argument

Behavioural ecologists seek to explain animal behaviour by starting from a different direction. They argue that animals need to behave efficiently if they are to survive and breed: selective pressure should, therefore, lead to animals adopting behavioural strategies that ensure maximal reproductive success. Just as evolution has provided animals with bodies exquisitely adapted to survival in their ecological niches, should not evolution also lead to similarly exquisite adaptations of behaviour? On this view, it should be possible to explain natural animal behaviour in terms of its contribution to reproductive success. This approach to the analysis of animal behaviour is known as the *optimality argument.*

The optimality argument is controversial: Gould and Lewontin (1979) attack its uncritical use, and Stephens and Krebs (1986) give an extended discussion of when the use of the optimality argument can be justified. It is clear that there are both practical and theoretical difficulties with the optimality argument.

One difficulty that Gould and Lewontin point out is that the optimality argument must be applied to an animal and its behaviour as a whole, and not to each aspect of the animal separately. Further, optimality can only be assessed

relative to the animal's 'choice' of overall design (or 'bauplan') and mode of life. One cannot recommend a whale to become a butterfly.

A potential weakness of the optimality argument is that evolution is not a perfect optimiser. It is likely, therefore, that there are some aspects of behaviour which have no adaptive value, just as there are some parts of the body that might benefit from being re-designed. Nevertheless, there are many examples where optimality arguments can be applied convincingly to explain aspects of animals' natural behaviour.

And, of course, optimality arguments may often be difficult to provide in practice because it may be difficult to establish what the optimal behavioural strategy actually is for an animal in the wild. The difficulty may be either that it is difficult to determine what an animal's intermediate goals should be if it is to leave as many surviving descendants as possible, or else the difficulty may be that although the goals of optimal behaviour are reasonably clear, it is difficult for a behavioural ecologist to know how animals could best go about achieving them. What *is* the best way for a squirrel to look for nuts?

To apply the optimality argument to any particular example of animal behaviour is fraught with subtle difficulties, and a substantial amount of investigation of the animal's behaviour and habitat is necessary. But I am not going to do this—all I need to assume is a rather limited form of the optimality argument, which is set out below.

3. Optimality and Efficiency

The optimality argument as applied to behaviour suggests that the function of instrumental learning is to learn to behave optimally. Some aspects of behaviour are innate, other aspects are learned, but all behaviour should be optimal. But this is much too simple.

There is a basic difficulty: animals cannot *learn* how to leave as many descendants as possible. It is not possible for an animal to live its life many times and to determine from its experience the optimal strategy for perpetuating its genes. All that an animal can learn to do is to achieve certain intermediate objectives. To ensure maximal reproductive success, animals may need to achieve a variety of intermediate goals: finding food and water, finding shelter, defending territory, attracting a mate, raising offspring, avoiding predators, resting, grooming, and so on. Animals may learn to achieve these goals, but they cannot learn optimal fitness directly.

It is often possible to identify certain skills that animals need to have—one such skill that many animals need is the ability to forage so as to gain energy at the maximum possible rate. To describe an intermediate objective quantitatively, it is necessary to specify a *performance criterion*, which can be used to 'score' different possible behavioural strategies. The *maximally efficient* behavioural strategy is the one that leads to the best score according to the performance criterion. If animals can represent suitable performance criteria internally, so that they can score their current behaviour, then it becomes possible for them to learn efficient behaviour. This is the type of learning I will examine.

But an animal will not always need to achieve all its intermediate objectives with maximal efficiency. A plausible view is that, for each species of animal, there are certain critical objectives, in that the levels of performance an animal achieves in these areas strongly affect its reproductive fitness. In other areas of behaviour, provided that the animal's performance is above some minimum level of efficiency, further improvements do not greatly affect fitness. For example, a bird may have ample leisure during the autumn when food is plentiful and it has finished rearing its young, but its survival in the winter may

depend critically on its ability to forage efficiently. There is, therefore, an important distinction between *optimal* behaviour in the sense of behaviour that ensures maximal reproductive success, and behaviour that is *maximally efficient* in achieving some intermediate objective.

It is possible that some animals need to learn to optimise their behaviour overall by learning to choose to devote appropriate amounts of time to different activities; but it is likely that it is more usual that animals need to learn certain specific skills, such as how to hunt efficiently. It is unlikely that an animal will have to learn to *seek* food when it is hungry: it is more likely to need to learn how to *find* food efficiently, so that it can exercise this skill when it is hungry.

Animals may, therefore, need to learn skills that they do not always need to use. Learning of this type is to some extent incidental: an animal may learn how to forage efficiently while actually foraging somewhat inefficiently, so that the animal's true level of skill may only become evident when the animal needs to use it. Furthermore, it is usually necessary to make mistakes in order to learn: animals must necessarily behave inefficiently sometimes in order to learn how to behave efficiently when they need to. This view of learning is rather different from traditional views of reinforcement learning, as presented by, for example, Bush and Mosteller (1955).

4. Learning and the Optimality Argument

The optimality argument does predict that animals should have the ability to learn—to adapt their behaviour to the environment they find. The reason for this is that

- *The same genotype may encounter circumstances in which different behavioural strategies are optimal.*

That is, either the same individual may need to adapt its behaviour, or different individuals may encounter different circumstances. This is not an entirely trivial point: one reason why we do not learn to beat our hearts is that the design of the heart, the circulatory system, and the metabolism is encoded in the genes, so that the optimal strategy for beating the heart may be genetically coded as well. It is possible, however, that there could be a mechanism for fine-tuning the control system for the heart-beat in response to experience, because physical development is not entirely determined by the genotype.

Naturally, optimality theory predicts optimality in learning, but there are two notions of optimality in learning: *optimal learning*, and *learning of efficient strategies*. 'Optimal learning' is a process of collecting and using information during learning in an optimal manner, so that the learner makes the best possible decisions at all stages of learning: learning itself is regarded as a multi-stage decision process, and learning is optimal if the learner adopts a strategy that will yield the highest possible return from actions over the whole course of learning. 'Learning of an efficient strategy' or 'asymptotically optimal' learning (Houston et al 1987) is a much weaker notion—all that is meant is that after sufficient experience, the learner will eventually acquire the ability to follow the maximally efficient strategy.

The difference between these two notions may be made clear by considering the 'two-armed bandit' problem. In this problem, a player is faced with two levers. On each turn, the player may pull either lever A or lever B, but not both. After pulling a lever, the player receives a reward. Let us suppose that, for each lever, the rewards are generated according to a different probability distribution. Successive rewards are independent of each other, given the choice of lever. The average rewards given by the two levers are different. The reward the player obtains, therefore, depends only on the lever he pulls.

Now, suppose that the player is allowed only 10 turns; at each turn, the player may decide which lever to pull based on the rewards he has received so far in the session.

If the player knows that lever A gives a higher reward than lever B, then clearly his maximally efficient strategy is to pull lever A 10 times. But if the player is uncertain about the relative mean rewards offered by the two levers, and his aim is to maximise his total reward over n turns, then the problem becomes interesting. The point is that the player should try pulling both levers alternately at first, to determine which lever appears to give higher rewards; once the player has sampled enough from both levers, he may choose to pull one of the levers for the rest of the session. Other sampling strategies are possible.

The difference between optimal learning and learning an efficient strategy is clear for this problem. Learning an efficient strategy is learning which lever gives the higher rewards on average; a learning method learns the efficient strategy if it always eventually finds out which lever gives the higher rewards. However, a learning method is optimal for a session of length n if it results in the player obtaining the highest possible expected reward over the n turns, 'highest possible' taking into account the player's initial uncertainty about the reward distributions of the levers.

Optimal learning is the optimal use of information to inform behaviour. It is learning that is optimal when considered over the whole course of learning, taking into account both early mistakes and later successes. Optimality in this sense refers to the learning method itself, not to the final behaviour attained. In the two-armed bandit problem, for example, if only a few turns are allowed, it may be optimal for the player to perform very little initial sampling before choosing one lever to pull for the rest of the session. If the player does not

perform enough sampling, then he may easily choose the wrong lever: if many turns are allowed, therefore, the optimal strategy may be to do more sampling. Note that optimal learning does not necessarily lead to the acquisition of the maximally efficient strategy: if learning the maximally efficient skill is costly, it may not be worthwhile for the animal to learn it.

The two-armed bandit problem is perhaps the simplest learning problem which involves a trade-off between exploration of the possibilities of the environment, and exploitation of the strategies that have been discovered so far. This is a dilemma that arises in almost any instrumental learning problem. If an animal performs too much exploration, it may not spend enough time in exploiting to advantage what it has learned: conversely, if an animal is incurious and does too little exploration, it may miss discovering some alternative behaviours that would bring much higher returns, and it may spend all its time exploiting an initial mediocre strategy. This is known as the *exploration-exploitation trade-off*. During its life time, an animal must continually choose whether to explore or whether to exploit what it knows already. One prediction of optimality theory, therefore, is that an animal should make an optimal choice in the exploration-exploitation trade-off. It may happen that, in following the optimal strategy, the animal will not necessarily perform enough exploration to achieve maximally efficient performance: it may be better to be incurious and so avoid making too many mistakes during exploration.

Houston and McNamara (1988) and Mangel and Clark (1988), propose explaining natural animal learning as optimal learning in this sense. This approach is surely correct for learning in the sense of collecting and using information, but it is in practice impossible to apply for learning in the sense of the gradual improvement of skill.

Rather confusingly, 'learning' is used in the operational research and dynamic programming literature (for example, de Groot (1970) or Dreyfus and Law (1977)) to refer to the short-term collection of information for immediate use. This is the sense of 'learning' as in 'In the darkness of the room there came a slow rustling sound and then the sound of the sofa being pushed across the floor, so I learned I was dealing with a snake of remarkable size.' as opposed to the sense of learning in 'It took him three months of continuous practice to learn to ride that unicycle.' The short-term collection and use of information (learning in the first sense) is a skill that can itself be gradually improved by practice (learning in the second sense).

Krebs, Kacelnik, and Taylor (1978) performed one of the first experiments to determine whether animals could learn to collect and use information in an optimal way—indeed, one of the first experiments to determine whether animals could learn an optimal strategy, where the optimality of the strategy was determined by reference to a dynamic model. They kept birds (great tits) in an artificial environment in which they were fed in a series of short sessions. For the duration of each feeding session, the bird was presented with two feeders, one of which would yield food more readily than the other. The birds could only tell which feeder was better by trial and error, so that each session was in effect a two armed bandit problem, and successive sessions were independent problems of the same type. A reasonable strategy for a bird—and one that is very nearly optimal—is to sample from both feeders for a short time at the start of each session, and then to feed for the rest of the session exclusively from the feeder that gave out food most readily during the sampling period. Over many sessions, the birds did indeed acquire near optimal strategies of this type.

But the type of learning that I will be interested in is the improvement in performance over many feeding sessions. In this example, the birds gradually learned a maximally efficient strategy for the problem. Was this gradual learning also optimal? That is a very difficult question to answer, for two reasons.

The first reason is that it is exceedingly difficult to devise demonstrably optimal learning strategies for any but the simplest of formal problems. Even for the two-armed bandit problem, finding the optimal strategy is a formidable computation. There is a straightforward general method, as explained in de Groot (1970), for constructing optimal learning strategies, but the strategies and the computation become impractically complex for any but small problems, or problems for which simplifying assumptions are possible. The problem of learning the optimal strategy in repeated two-armed bandit problems is far too complex for it to be possible to determine the optimal learning strategy.

But a second and more fundamental difficulty is that an optimal learning strategy is optimal only with respect to some prior assumptions concerning the probabilities of encountering various possible environments. In an experiment similar to that of Krebs et al (1978), the optimal strategy within a feeding session depends on the distribution of yields from the feeders in different sessions. After the great tits have experienced many feeding sessions, they have enough information to 'know' the statistical distribution of the yields from each feeder, and it makes sense to ask whether they can acquire the optimal strategy for the distribution of yields that they have experienced. But to ask whether the birds' learning is optimal over the whole experiment is a different matter: the optimal strategy from the birds' point of view is depends on the birds' prior expectations, and we have no means of knowing what these expectations are or what they should optimally be.

In other words, to show that some particular learning method is optimal, it is necessary to specify a probability distribution over the environments that the animal may encounter, as noted by McNamara and Houston (1985). In practice, this is likely to be an insuperable difficulty in providing convincing quantitative optimality explanations for any type of skill acquisition.

But although quantitative arguments on optimal learning may be difficult to provide, some qualitative explanations involving optimal learning are common sense. Animals that are physically specialised so that they are adapted to a particular way of life, for example, should in general be less curious and exploratory than animals that are physically adapted to eat many different foods. The reason for this is that a highly specialised animal is unlikely to discover viable alternative sources of food, while an omnivore lives by adapting its behaviour to exploit whatever is most available.

I am not going to consider computational models of optimal learning, both because of the technical difficulty of constructing optimal learning methods, and because of the need to introduce explicit assumptions about a probability distribution over possible environments. In any case, optimal learning will seldom be a practical quantitative method of explaining animal learning.

Let us return to the second type of learning—*learning of efficient strategies*. By the learning of efficient strategies, I mean the acquisition of the ability to follow a strategy that is maximally efficient according to an intermediate criterion. Note that this is learning of the *ability* to follow a maximally efficient strategy: an animal with this ability need not always actually follow the efficient strategy, but it can do so if it chooses to.

An example where optimality theory would predict that an animal should learn an efficient strategy is that a prey animal should learn how to return to its burrow as fast as possible from any point in its territory. Of course, the animal

need not always return to its burrow as fast as possible—but it is vitally necessary that it should be able to do so if danger threatens. Similarly, it is not important that an animal should follow an efficient strategy in searching for food if it has just eaten, but it is advantageous for an animal to be able to follow an efficient strategy in searching for food if it needs to.

Optimal learning will require the learning of an efficient strategy if the following three conditions hold:

- *The capacity for maximally efficient performance is valuable.*
- *Exploration is cheap.*
- *The time taken to learn the behaviour is short compared to the period of time during which the behaviour will be used.*

The third condition implies that the final level of performance reached is more important than the time taken to learn it—hence optimal learning will consist of learning the efficient strategy.

Animals need to be able to survive adverse conditions that are more extreme than those they usually encounter. It is likely, therefore, that under normal circumstances most animals have some leisure for exploration; in other words, the opportunity cost of exploration is usually small. Animals may, therefore, normally perform with slightly less than maximum efficiency in order to be able to learn: maximally efficient performance is only occasionally necessary. Efficient performance may be valuable for the animal to acquire either because it is occasionally vital (as in avoiding predators), or else because it continuously ensures a small competitive advantage (as in searching for food).

Even if these assumptions are not entirely satisfied, it is still plausible that animals should learn efficient strategies. The point is that optimal learning will entail learning an efficient strategy unless learning is expensive. Learning may

be expensive if mistakes are costly: prey animals would be unwise to attempt to learn which animals were their predators by experience, for example. If animals have innate behaviours that prevent them from making disastrous mistakes, there is no reason why these behaviours should not be fine-tuned by instrumental learning. If an innately feared predator never behaves in a threatening way, for example, the prey animal may lose some of its fear, and so cease spending time and energy in avoiding the predators. Animals may have innate knowledge or behaviours that prevent them from making costly initial mistakes, and these innate behaviours may be progressively modified to become maximally efficient behaviours through instrumental learning. In other words, innate knowledge may take the pain out of learning.

In conclusion, the optimality assumption leads to the hypothesis that animals will be able to learn efficient behavioural strategies. That is, after sufficient experience in an environment, an animal should acquire the ability to exploit that environment with maximal efficiency. Most of the thesis is devoted to investigating what algorithms animals might use to learn in this way.

4.1. Learning Efficient Strategies and Conditioning

Instrumental conditioning and the learning of efficient strategies are related concepts, but they are not at all the same. The motivation for studying instrumental conditioning is that it is possible mechanism for a type of learning that could be useful to an animal in the wild. However, operant conditioning theory does not explicitly consider efficiency of strategy, and many aspects of instrumental conditioning experiments are not directly interpretable from the viewpoint of optimality theory. Conversely, many experiments that test whether animals can learn an efficient behavioural strategy are not easy to interpret as instrumental conditioning experiments.

Although they are superficially similar to instrumental conditioning experiments, experiments to test whether animals can learn maximally efficient behavioural strategies are designed quite differently. A well designed 'learning of efficiency' experiment should give animals both incentive and opportunity to learn the maximally efficient strategy.

- Animals should be placed in an artificial environment for which the experimenter can determine the maximally efficient behavioural strategy.
- The animals should be left in the artificial environment for long enough for them to have ample opportunity of optimising their strategies. The environment should not be changed during this time.
- The animals should have an adequate motivation to acquire the optimal strategy, but the incentive should not be so severe that exploration of alternative behaviours is too costly.
- Control groups should be placed in artificial environments that differ in chosen respects from the environment of the experimental group. Control groups should be given the same opportunities of optimising their behaviour as the experimental group.

Experiments designed in this way have two considerable advantages. First, it is possible to devise experiments that simulate directly certain aspects of natural conditions. Second, optimality theory can be used to make quantitative predictions about what strategy the animals will eventually learn.

Some conditioning experiments satisfy these design requirements; others do not. For example, the phenomenon known as 'extinction' in conditioning theory, in which a learned response gradually extinguishes when the stimulus is repeatedly presented without the reinforcer, is not directly interpretable in terms of optimality. This is because in a typical instrumental conditioning experiment, the purpose of testing animals under extinction is to determine the persistence

or 'strength' of the animal's expectation of a reward following the stimulus. This concept of 'strength' is difficult to interpret in terms of optimality. There is often no 'correct' behaviour during extinction: whether an animal should continue to respond for a long time or not depends entirely upon what types of regularity it should expect to find in its environment. Since the extinction condition occurs only once during the experiment, the animal is not given enough data for it to work out what it ought to do.

If, on the other hand, extinction were to occur repeatedly in the course of an experiment, the animal has the chance to learn how to react in an optimal way. Kacelnik and Cuthill (1988) report an experiment in which starlings repeatedly obtain food from a feeder. Each time it is used, the feeder will supply only a limited amount of food, so that as the birds continue to peck at the feeder they obtain food less and less often, until eventually the feeder gives out no more food at all. To obtain more food, they must then leave the feeder and hop from perch to perch in their cage until a light goes on that indicates that the feeder has been reset. In terms of conditioning theory, this experiment is (roughly) a sequence of repeated extinctions of reinforcement that is contingent upon pecking at the feeder: however, because the birds have the opportunity to accumulate sufficient experience over many days, they have the necessary information to find an optimal strategy for choosing when to stop pecking the feeder.

I do not want at all to suggest that conditioning experiments are uninterpretable: they ask different questions and, perhaps, provide some more detailed answers than optimality experiments do. However, the optimality approach is both quantitative and strongly motivated, and I will argue in the rest of this thesis that it is possible to classify and to implement a range of algorithms for learning optimal behaviour.

5. Special-Purpose Learning Methods

McNamara and Houston (1980) describe how decision theory may be used to analyse the choices that animals face in some simple tasks and to calculate the efficient strategy. They point out that it is in principle possible that animals might learn by statistical estimation of probabilities, and then use decision theory to calculate their strategies, but they suggest that it is more likely that animals learn by special purpose, ad hoc methods. McNamara and Houston give two main arguments in favour of this conclusion.

First, they point out that the calculations using decision theory are quite complex even for simple problems, and that, in order to perform them, the animals would need to collect a considerable amount of information that they would not otherwise need. Their second argument is that animals do not face the problem of determining optimal strategies in general: each species of animal has evolved to face a limited range of learning problems in a limited range of environments. Animals, therefore, should only need simple, special-purpose heuristic learning methods for tuning their behaviour to the optimum. These special-purpose strategies may break down in artificial environments that are different from those in which the animals evolved.

The classic example of a heuristic, special-purpose, fallible learning method of this type is the mechanism of imprinting as described by Lorenz. In captivity, the ducklings may become imprinted on their keeper rather than on their mother. There can be no doubt that many other special-purpose learning methods exist, of exactly the type that McNamara and Houston describe.

But I do not think that McNamara and Houston's arguments are convincing in general. Although some innate special-purpose adaptive mechanisms demonstrably exist, it is implausible that all animal learning can be described in this way. Many species such as rats or starlings are opportunists, and can learn

to invade many different habitats and to exploit novel sources of food. Animals can be trained to perform many different tasks in conditioning experiments, and different species appear to learn in broadly similar ways. Is it not more plausible that there are generally applicable learning mechanisms, common to many species, that can enable animals to learn patterns of behaviour that their ancestors never needed?

The next section presents a speculative argument that special-purpose learning methods may sometimes evolve from general learning methods applied to particular tasks. But the most convincing argument against the hypothesis of special-purpose learning methods will be to show that simple general learning methods are possible, which I will attempt to do later on.

6. Learning Optimal Strategies and Evolution

Evolution may speed up learning. If the learning of a critical skill is slow and expensive, then there will be selective pressure to increase the efficiency of learning. The efficiency of learning may be improved by providing what might be called 'innate knowledge'. By this, I do not necessarily mean knowledge in the ordinary sense of knowing how to perform a task, or of knowing facts or information. Instead, I mean by 'innate knowledge' any innate behavioural tendency, desire, aversion, or area of curiosity, or anything else that influences the course of learning. An animal that has evolved to have innate knowledge appropriate for learning some skill does not necessarily *know* anything in the normal sense of the word, but in normal circumstances it is able to *learn* that skill more quickly than another animal without this innate knowledge.

A plausible hypothesis, therefore, is that useful behaviours and skills are initially learnt by some individuals at some high cost: if that behaviour or skill is sufficiently useful, the effect of selective pressure will be to make the

learning of it quicker, less costly, and more reliable. One origin of special-purpose learning methods, therefore, may be as innate characteristics that have evolved to speed up learning by a general purpose method.

7. How Can a Learning Method be General?

A 'general learning mechanism' is an intuitively appealing idea, but it difficult to pin down the sense in which a learning mechanism can be general, because all learning must start from some innate structure. It has become a commonplace in philosophy that learning from a *tabula rasa* is necessarily impossible. Any form of learning or empirical induction consists of combining a finite amount of data from experience with some prior structure. No learning method, therefore, can be completely general, in the sense that it depends on no prior assumptions at all.

However, there is another, more restricted sense in which a learning method can be 'general'. An animal has sensory abilities that enable it to distinguish certain aspects of its surroundings, it can remember a certain amount about the recent past, and it has a certain range of desires, needs, and internal sensations that it can experience. It can perform a variety of physical actions. A behavioural strategy is a method of deciding what action to take on the basis of the surroundings, of the recent past, and of the animal's internal sensations and needs. A strategy might be viewed as a set of situation-action rules, or as a set of stimulus-response associations, where the situations or 'stimuli' consist of the appearance of the surroundings, memories of the recent past, and internal sensations and desires, and the 'responses' are the actions the animal can take. I do not wish to imply that a strategy is actually *represented* as a set of stimulus-response associations, although some strategies can be: the point is merely that a strategy is a method of choosing an action in any situation.

Learning is a process of finding better strategies. Now, a given animal will be able to distinguish a certain set of situations, and to perform a certain set of actions, and it will have the potential ability to construct a certain range of situation-action strategies. A general learning method is a method of using experience to improve the current strategy, and, ideally, to find the strategy that is the best one for the current environment, given the situations the animal can recognise and the actions the animal can perform. As will be shown, there are general methods of improving and optimising behavioural strategies in this sense.

8. Conclusion

I have argued that the optimality argument of behavioural ecology does indicate an analysis of associative instrumental learning, but the connection between the optimality argument and associative instrumental learning is indirect. Animals cannot directly learn to optimise their fitness, because they cannot live their lives many times and learn to perpetuate their genes as much as possible. Instead, animals may learn *critical skills* that improve their fitness.

By a 'critical skill', I mean a skill for which improvements in performance result directly in increases in fitness. For example, the rate at which a bird can bring food to its nest directly affects the number of chicks it can raise. In many cases, an animal need not always perform its critical skills with maximal efficiency: it is the *capacity* to perform with maximal efficiency when necessary that is valuable. A bird may need to obtain food with maximal efficiency all the time during the breeding season, but at other times it may have leisure.

One role of instrumental learning, therefore, is in acquiring the ability to perform critical skills as efficiently as possible. This learning may be to some extent *incidental*, in that performance does not always have to be maximally

efficient during learning: indeed, sub-optimal performance may be a necessary part of learning.

According to the optimality argument, if an animal has many opportunities to practise a critical skill, and if it is able to try out some alternative strategies without disaster, then the animal should ultimately acquire a capacity for maximally efficient performance.

In the next chapter, I will describe how a wide range of learning problems may be posed as problems of learning how to obtain delayed rewards. I will argue that it is plausible that animals may represent tasks subjectively in this way. After that, I will describe the established method for calculating an optimal strategy, assuming that complete knowledge of the environment is available. Then I will consider systematically what learning methods are possible. The learning methods will be presented as alternative algorithms for performing dynamic programming. After that, I will describe computer implementations of some of these learning methods.

From now on, I will speak more often about hypothetical 'agents' or 'learners' rather than about 'animals', because the discussion will not be related to specific examples of animal learning. The learning algorithms are strong candidates as computational models of some types of animal learning, but they may also have practical applications in the construction of learning machines.

Chapter 2
Learning Problems

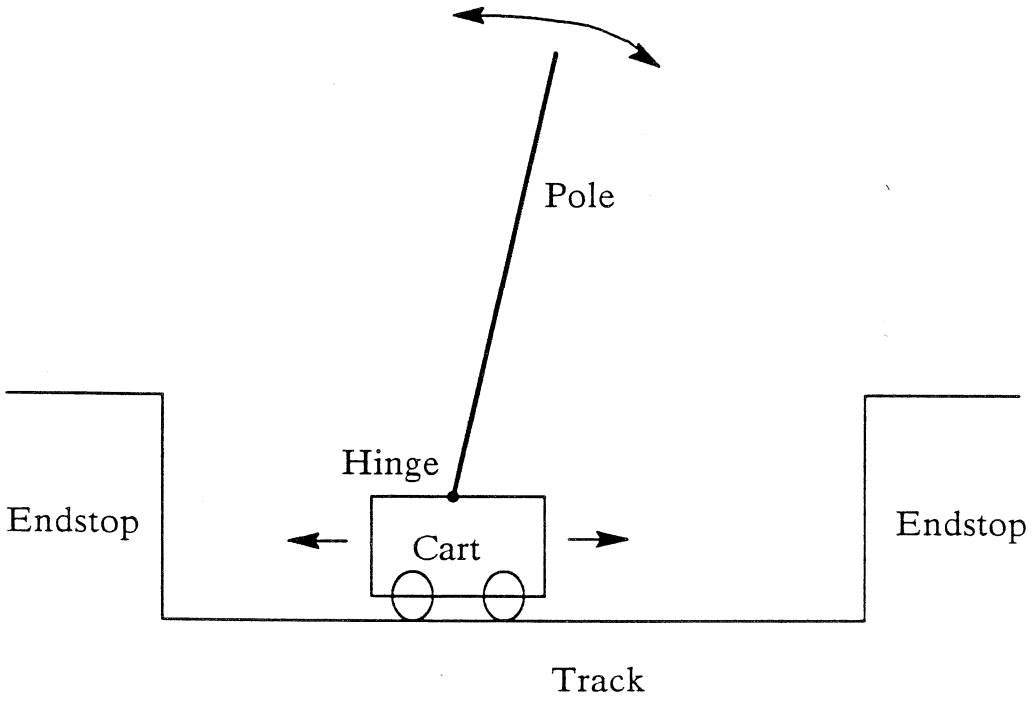
In this chapter I will describe several problems to which the learning methods are applicable, and I will indicate how the problems are related.

1. The Pole-Balancing Problem

A well-known example of a procedural learning problem, studied by Michie (1967), and Barto, Sutton, and Anderson (1983), is the 'pole-balancing' problem, illustrated overleaf.

The cart is free to roll back and forth on the track between the two end-blocks. The pole is joined to the cart by a hinge, and is free to move in the vertical plane aligned with the track. There are two possible control actions, which are to apply a constant force to the cart, pushing it either to the right or to the left. The procedural skill to be acquired is that of pushing the cart to left and right so as to keep the pole balanced more or less vertically above the cart, and to keep the cart from bumping against the ends of the track. This skill might be represented as a rule for deciding whether to push the cart to the right or to the left, the decision being made on the basis of the state of the cart-pole system.

There are several ways of posing this as a learning problem. If an 'expert' is available, who knows how to push the cart to balance the pole, then one approach would be to train an automatic system to imitate the expert's behaviour. If the learner is told which action would be correct at each time, the learning problem becomes one of constructing a mapping from states of the cart



At each time step, the controller may push the cart either to the right or to the left.

The task is to keep the pole balanced, and to keep the cart from hitting the endstops.

and pole to actions. The problem of learning a procedural skill is reduced to the problem of learning a single functional mapping from examples. The disadvantage of this 'imitate the teacher' method is that a teacher may not be available: indeed, if there is a machine teacher, there is rarely any point in having a machine learner.

A more interesting and general formulation of the learning problem is that the learner receives occasional rewards and penalties, and that the learner's aim is to find a policy that maximises the rewards it receives. The pole-balancing problem, for example, can be formulated as follows. The learner may repeatedly set up the cart and pole in any position, and it may then push the cart to and fro, trying to keep the pole balanced. The information that the learner can use consists of the sequence of states of the cart-pole system and of the actions that the learner itself performs; the learner is informed when the pole is deemed to have fallen. The falling of the pole may be thought of as a punishment or 'penalty', and the learner may be viewed as having the goal of avoiding these penalties. Note that the learner is not given the aim of 'keeping the pole nearly vertical and the cart away from the ends of the track': it is just given the aim of avoiding penalties, and it must work out for itself how to do this. The learner does not know beforehand when penalties will occur or what sort of strategy it might follow to avoid them. The learning method of Barto, Sutton, and Anderson (1983) learns under these conditions.

In the 'imitate the teacher' formulation of procedural learning, the learner is told whether each action it performs is correct; in the reward/punishment formulation, the rewards or punishments may occur several steps after the actions that caused them. For example, it may be impossible for the learner to prevent the pole from falling for some time before the pole actually falls: the final actions the learner took may have been correct in that by these actions the

falling of the pole was delayed as long as possible, and the actual mistaken actions may have occurred some time earlier.

1.1. States, Actions, and Rewards

The pole-balancing problem has a particularly clear structure. What needs to be learned is a method for deciding whether to push right or left. At any time, all the information that is needed to make this decision can be summed up in the values of four *state-variables*:

the position of the cart on the track

the velocity of the cart

the angular position of the pole relative to the cart

the angular velocity of the pole

These variables describe the physical state of the system completely for the purposes of pole-balancing: there is no point in considering any more information than this when deciding in which direction to push. If this information is available for the current time, then it is not necessary to know anything more about the past history of the process.

The space of possible combinations of values of these state variables is the *state-space* of the system—the set of possible situations that the agent may face. The purpose of learning is to find some method for deciding what action to perform in each state: the agent has mastered the skill if it can decide what to do in any situation it may face. There need not necessarily be a single prescribed action in any state—there may be some states in which either action is equally good, and a skilful agent will know that it can perform either.

The state-space may be described in alternative ways. For example, suppose that the agent cannot perceive the rates of change of the position of the cart or of the angle of the pole. It cannot, therefore, perceive the state of the

system directly. However, if it can remember previous positions and angles and the actions it has recently taken, then it can describe the state in terms of its memories of previous positions and previous actions. If the information that the agent uses to describe the current state would be sufficient to determine approximate values of all four state variables, then it is in principle sufficient information for the agent to decide what to do. In the cart and pole problem, it is easy to define one adequate state-space: many other systems of descriptions of the state are possible. The criterion for whether a method of description of state is adequate is that distinct points in the adequate state space should correspond to distinct descriptions.

2. Foraging Problems

Stephens and Krebs (1987) review a number of decision-making problems that animals face during foraging, and describe formal models of these problems that have been developed for predicting what animal behaviour should be according to the optimality argument.

One ubiquitous problem that animals face is that food is non-uniformly distributed—it occurs in 'patches': foraging consists of searching for a 'patch' of food, exploiting the patch until food becomes harder to obtain there, and then leaving to search for a new patch. While searching for a patch, the animal expends energy but obtains no food. When an animal finds a patch, the rate of intake of food becomes high initially, and then declines, as the obtainable food in the patch becomes progressively exhausted. Eventually, the animal has to make an unpleasant decision to leave the current patch and search for a new patch to exploit. This decision is 'unpleasant' in that the animal must leave behind some food to go and search for more, and the initial effect of leaving a patch is to reduce the rate of intake of food.

This dilemma is known as the *patch-leaving problem.* If the animal stays in patches too long, it will waste time searching for the last vestiges of food in exhausted patches. If, on the other hand, the animal leaves patches too soon, then it will leave behind food that it could profitably have eaten.

The animal gains energy from food, and spends energy in looking for and in acquiring it. It is reasonable to suppose that the optimal foraging strategy for an animal is to behave so as to maximise some average rate of net energy gain. A common assumption is that an animal will maximise its *long-term* average rate of energy gain; this is not the only assumption that is possible, but it is one that is frequently used by foraging theorists.

Charnov and Orians (1973), as cited by Stephens and Krebs (1986), Chapter 3, considered this and similar problems, and proposed the *marginal value theorem.* This applies in circumstances where

- At any time, an animal may be either searching for opportunities (e.g. prey, patches), or else engaged in consumption (e.g. eating prey, foraging in a patch).
- The animal may abandon consumption at any time, and return to searching for a new opportunity. The animal may only stop searching once it has found a new opportunity.
- The results of searches on different occasions—the values of the opportunity discovered, the time taken to find the opportunities, etc—are statistically independent.
- New opportunities are not encountered during consumption.
- The rate of energy intake during consumption declines monotonically. That is, 'patch depression' is monotonic.

- The decision problem for the animal is that of when to stop consumption and return to searching. An animal has the option of not starting consumption at all if an opportunity is not sufficiently promising.

If all these assumptions are satisfied, then there is a simple optimal decision rule, which has the following form. Suppose that the best possible long-term average rate of energy gain is L. Then the optimal decision rule is to abandon consumption and return to search when the instantaneous rate of energy gain falls to L or below. This definition might appear circular but it is not: L is a well defined quantity that could in principle be found by calculating the long term average returns of all possible strategies, and then choosing the largest of the results. The marginal value theorem states that the optimal strategy consists of leaving a patch when the rate of return drops below L, and this fact can often be used as a short-cut in finding L.

This decision rule, however, requires an animal to assess the instantaneous rate of energy gain during consumption. There are some circumstances where it is reasonable for animals to be able to do this (e.g. a continuous feeder such as a caterpillar), but it also often happens that energy gain in a patch is a stochastic process, in which food comes in chunks within a patch (such as a bird eating berries in a bush). In this case, the animal cannot directly measure its instantaneous rate of energy gain, but the marginal value theorem still applies if the instantaneous *expected* rate of energy gain is used. To use a decision rule based on the expected instantaneous rate of return, the animal must estimate this on the basis of other information, such as the appearance of the patch, the history of finding food during residence in the patch, and whether it has found signs of food in the patch.

McNamara (1982) has proposed a more general type of decision rule, based on the idea of *potential*. This formulation is more general than that

required for the marginal value theorem in that it is no longer necessary to assume that the rate of energy gain declines monotonically as an animal exploits a patch: the other assumptions remain the same.

The potential is rather cumbersome to define in words, but a definition runs as follows. An animal continually estimates the potential of a patch on the basis of what it currently knows about the patch. The potential is the estimated maximum achievable ratio of energy gain to residence time in the patch, the maximum being taken over all possible exploitation strategies that the animal might adopt. A bird might, for example, estimate the potential of a particular tree on the basis of the type of tree, the season, how long it has been searching in the tree, and how many berries it has found recently. The decision rule is to leave the patch if the potential drops below L, and to stay otherwise.

This is a considerably more complex analysis than Charnov and Orians' original presentation of the marginal value theorem. All that is left of the simplifying assumptions for the marginal value theorem is that the searches are statistically independent, and this assumption itself may not always be plausible. If this assumption too is dropped, a still more general method of determining optimal strategies may be used: dynamic programming.

The point is that, to calculate optimal strategies and how they depend on certain environmental variables, it is necessary to construct a simplified formal model of the foraging problem that the animal faces. It is sometimes possible to justify a very simple type of model, such as the type of model needed to apply the marginal value theorem, and the optimal strategy can then be derived by some simple statistical reasoning and a little algebra. However, these strong assumptions will often be unrealistic.

2.1. Dynamic Models of Foraging

The most general form of foraging model that it is reasonable to construct is a *dynamic model*, which can be described in the following terms. The foraging problem is described abstractly, in such a way that the foraging could in principle be simulated on a computer. At any time, the animal and the environment can be in any of a certain set of *objective states*, the objective state being the information about the state of the animal and its environment that is necessary for continuing the simulation. The objective state may contain information that would not be available to the animal, such as how much food is left in the current patch.

Let us suppose that the animal does not make decisions continuously, but that decision points occur at intervals during the simulation. The foraging problem faced by the animal is that of taking decisions as to what to do next: at each decision point, there is a certain range of alternative actions that the animal can choose between. The action the (simulated) animal chooses will affect the amount of food that it finds in the time up to the next decision point, and it will affect the objective state at the next decision point.

The book by Mangel and Clark (1988) is an extended description of this approach to the modelling of the behavioural choices that animals face.

3. Subjective Dynamic Models

One constraint on foraging is that an animal can take decisions only on the basis of the information available to it. The animal may not be able to know the objective state, but the information that the animal uses to decide what to do might be called the *subjective state*. The subjective state may consist of information about the appearance of the environment, about recent events in the past, and about the animal's internal state, and about its current

goals, if it has any. I will suppose that an animal has an internal subjective model of the foraging problem, different from, but corresponding to the objective model. The animal forms subjective descriptions of the situations it faces, the actions it takes, and of the benefits or costs of the actions it takes (the rewards).

What are the subjective rewards and costs? A behavioural ecologist may determine what an animal's short term goals ought, objectively, to be, but need the animal's subjective reward system correspond directly to the objective reward system? If animals learn to achieve subjective rewards rather than objective rewards, then all that can be deduced from the optimality argument is that the optimal strategy according to the subjective reward system should be the same as the optimal strategy according to the objective reward system. This does *not* imply that the objective and subjective rewards are the same.

A plausible example where there is a difference between subjective and objective rewards is that of an innate fear of predators. If a bird feeds in a certain spot, sees a kestrel, and escapes, it has suffered no objective penalty because it has survived. The only objective penalties from predation occur when animals get eaten, after which they can no longer learn. In order for animals to avoid predators, it is plausible that sights and sounds of predators should be subjectively undesirable experiences that the animals should seek to avoid. Within the framework of a subjective dynamic model, the sight of a predator should therefore be a subjective penalty. In the bird's natural environment, a policy of avoiding situations in which predators are seen may be a good policy for avoiding predation.

To suggest possible learning methods, I do need to assume that the animal or agent's subjective representation of the problem is sufficiently detailed that the subjective problem is that of controlling a Markov decision process. This is

a large assumption to have to make, but it is unavoidable. The only consolation is that the learning may often succeed even if the assumption does not hold. As I will not mention this assumption for the rest of the thesis, I will give two examples of how things can go wrong if the agent does not encode enough information about states for the results of its actions to be predictable.

The first example is that of finding one's way about in central London. It is easy enough to build up a mental map of the streets and the junctions, so that one can mentally plan a route; the 'states' in this case are the junctions, and the 'actions' are the decisions as to which street to turn into at each junction. Now, the street layout and even the one way system are easy enough to remember, but this information is not sufficient, because there are heavy restrictions on which way one is allowed to turn at each junction. That is, a description of which junction one is at is *not* a description of the state that is sufficient for determining what action to take—it is also necessary to specify which street one is in at the junction. If one does not succeed in remembering the turning restrictions, one cannot plan efficient routes.

As a second example of a non-Markovian subjective problem, consider an animal undergoing some conditioning experiment in which the reinforcements depend in a complicated way on the sequence of recent events and actions. If the animal does not remember enough information about the recent past to be able to distinguish aspects that are relevant to the reinforcement it will receive, then it may not be able to learn the most efficient strategy for exploiting that environment. Furthermore, the environment could be contrived so as to frustrate the animal's attempts to learn a simple strategy that made use only of the information that it could encode.

In both of these examples, the problem is that the agent does not encode enough relevant information about the situation it is in to be able to have

effective situation action rules. In both cases the remedy is clear: the agent should base its decisions on more information: it is a question of determining what information defines the state.

Autonomous learning agents will surely need to have some methods, possibly heuristic, of detecting whether their current encoding of state is adequate. I have not considered this problem, and I think that it is unlikely that there is any single general approach to it. From now on, I will assume that the agent's subjective formulation of the problem is indeed a Markov decision process.

To suppose that animals formulate subjective problems in this way is to make a *psychological* hypothesis. I think that this hypothesis is both plausible and fully consistent with long established assumptions about associative learning.

Chapter 3
Markov Decision Processes

The formal model that will be used for these and other problems is the Markov decision process; there are a number of books that treat Markov decision processes. A clear and concise account of discrete Markov processes is given in Ross (1983); other books are Bellman and Dreyfus (1962), Denardo (1975), Bertsekas (1976), Dreyfus and Law (1977), and Dynkin and Yushkevich (1976). To make this document self-contained, I will give a brief account of the main methods and results for finite-state problems here.

A *Markov decision process*, or *controlled Markov chain*, consists of four parts: a state-space **S**, a function **A** that gives the possible actions for each state, a transition function T, and a reward function R.

The state-space **S** is the set of possible states of the system to be controlled. In the case of the cart-pole system, the state-space is the set of 4-vectors of values of the position and velocity of the cart, and of the angular position and angular velocity of the pole.

In each state, the controller of the system may perform any of a set of possible actions. The set of actions possible in state x is denoted by $\mathbf{A}(x)$. In the case of the cart-pole system, every state allows the same two actions: to push right or to push left.

1. Finite Approximation to Continuous Processes

To avoid the complications of systems which have continuous state-spaces, continuous action sets, or which operate in continuous time, I will consider only finite, discrete-time Markov decision processes. That is,

- S is a finite set of discrete states
- $A(x)$ is a finite set of discrete actions for all x in S
- states are observed, actions taken, and rewards received at discrete times $1,2,3, \cdots$

Any non-pathological continuous Markov decision process may be approximated adequately for present purposes by a finite, discrete-time Markov decision process. From now on, I will discuss only finite Markov decision processes unless I specifically say otherwise.

The random variable denoting the state at time t is X_t, and the actual state at time t is x_t. The state at time t+1 depends upon the state at time t and upon the action a_t performed at time t. This dependence is described by the transition function T, so that $T(x_t , a_t) = X_{t+1}$, which is the state at time t+1. Transitions may be probabilistic, so that $T(x,a)$ may return a state sampled from a probability distribution over S.

Since there is only a finite number of states, we may define $P_{xy}(a)$ to be the probability that performing action a in state x will transform x into state y. That is

$$P_{xy}(a) = Pr(T(x,a) = y)$$

For finite systems, T is fully specified by the numbers $P_{xy}(a)$ for all x, y, and a.

Finally, at each observation, the controller receives a reward that depends upon the state and the action performed. The random variable denoting the reward at time t is R_t, and the actual reward at time t is r_t. That is, the reward

received is $R_t = R(x_t , a_t)$. In the case of the pole-balancing problem, the rewards might be defined as -1 at the time-step during which the pole falls, and 0 at all other times. Rewards may be probabiliistic: the actual reward may be sampled from a probability distribution determined by x and a.

Usually, we need not consider the reward function itself, but only its expectation, which is written

$$\rho(x,a) = \mathbf{E}[R(x,a)] \quad \text{for fixed } x \text{ and } a$$

This completes the definition of a Markov decision process.

2. The Markov Property

Note that although both transitions and rewards may be probabilistic, they depend only upon the current state and the current action: there is no further dependence on previous states, actions, or rewards. This is the *Markov property*. This property is crucial: it means that the current state of the system is all the information that is needed to decide what action to take—knowledge of the current state makes it unnecessary to know about the system's past.

It is important to note that the Markov properties for transitions and for rewards are not intrinsic properties of a real process: they are properties of the state-space of the model of the real process. Any process can be modelled as a Markov process if the state-space is made detailed enough to ensure that a description of the current state captures those aspects of the world that are relevant to predicting state-transitions and rewards.

3. Policies

A *policy* is a mapping from states to actions—in other words, a policy is a rule for deciding what to do given knowledge of the current state. A policy should be defined over the entire state-space: the policy should specify what to do in any situation.

A policy that specifies the same action each time a state is visited is termed a *stationary* policy (Ross 1983). A policy that specifies that an action be independently chosen from the same probability distribution over the possible actions each time a state is visited is termed a *stochastic* policy. During learning, the learner's behaviour will change, so that it will be neither stationary nor stochastic; however, the optimal policies that the learner seeks to construct will be stationary.

If a stochastic policy f is followed in state x_t, the probability that the next state is y is

$$Pr(\, X_{t+1} = y \,) \;=\; \sum_{a \,\in\, A(x)} Pr(\, f(x) = a \,)\, P_{xy}(a)$$

It will be convenient to write the transition probability from x to y when following policy f as

$$P_{xy}(\, f \,)$$

and, similarly, let

$$T(\, x, f \,), \quad R(\, x, f \,), \quad \rho(\, x, f \,)$$

be the next state, the reward, and the expected reward respectively when following policy f in state x.

4. Return

Broadly, the aim of the agent is to maximise the rewards it receives. The agent does not merely wish to maximise the immediate reward in the current state, but wishes to maximise the rewards it will receive over a period of future time.

There are three main methods of assessing future rewards that have been studied: total reward; average reward; and total discounted reward. I will assume that the agent seeks to maximise total discounted rewards, because this is the simplest case. The learning methods can, however, be modified for learning to maximise total rewards, or average rewards under certain conditions.

The total discounted reward from time t is defined to be

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n} + \cdots$$

where r_k is the reward received at time k and γ is a number between 0 and 1 (usually slightly less than 1). γ is termed the *discount factor.* The total discounted reward will be called the *return.* The effect of γ is to determine the present value of future rewards: if γ is set to zero, a reward at time t+1 is considered to be worth nothing at time t, and the return is the same as the immediate reward. If γ is set to be slightly less than one, then the expected return from the current state will take into account expected rewards for some time into the future. Nevertheless, for any value of γ strictly less than one, the value of future rewards will eventually become negligible.

5. Optimal Policies

The aim of the learner will be to construct a policy that is optimal in the sense that, starting from any state, following the policy yields the maximum possible expected return that can be achieved starting from that state. That is,

an optimal policy indicates the 'best' action to take in any possible situation in the sense that continuing to follow the policy will lead to the highest possible expected return.

It may not be obvious that the highest possible expected return can be achieved by following a stationary policy, or even that there is a single policy that will be optimal over all states. Nevertheless, it can be proved that for every Markov decision process as described above, there will be a stationary optimal policy, and the proof may be found in Ross (1983). The essential reason for this is that, in a *Markov* process, a description of the current state contains all information that is needed to decide what to do next; hence the same decision will always be optimal each time a state is visited.

6. The Credit-Assignment Problem

It is not immediately obvious how to compute the optimal policy, let alone how to learn it. The problem is that some judicious actions now may enable high rewards to be achieved later; each of a sequence of actions may be essential to achieving a reward, even though not all of the actions are followed by immediate rewards. Conversely, in the pole-balancing problem, the cart and pole may enter a 'doomed' state from which it is impossible to prevent the pole from eventually falling—but the pole actually falls some time later. It would be wrong to blame the decisions taken immediately before the pole fell, for these decisions may have been the best that could be taken in the circumstances. The actual mistake may have been made some time previously.

Because of this difficulty of determining which decisions were right and which were wrong, it may be difficult to decide what changes should be made to a suboptimal policy. In artificial intelligence, this problem of assigning credit or blame to one of a set of interacting decisions is known as the 'credit

assignment problem'. For efficient learning, it is necessary to have some efficient way of finding changes to a policy that improve it, because the sheer number of different possible policies in any significant problem makes a strategy of policy optimisation by trial and error hopelessly inefficient.

Chapter 4
Policy Optimisation by Dynamic Programming

Dynamic programming is a method of solving the credit-assignment problem in multi-stage decision processes. The scope of dynamic programming is often misrepresented in the computer science literature—the true variety of its applications is perhaps best explained in Bellman and Dreyfus (1962) or Dreyfus and Law (1977). The basic principle of dynamic programming is to solve the credit assignment problem by constructing an *evaluation function*, also known as a *value function* or a *return function*, on the state-space.

In discussing Markov decision processes, it is necessary to be able to refer to random quantities such as 'the state that results after starting at state x and following policy f for five time steps'. A notation for this is the following, where x is a state, f is a policy, and n is a non-negative integer:

$$\mathrm{X}(x,f,n) \quad \text{and} \quad \mathrm{R}(x,f,n)$$

are the random variables denoting the state reached and the immediate reward obtained, after starting at state x, and following policy f for n steps. Clearly,

$$\mathrm{X}(x,f,0) = x$$

and

$$\mathrm{R}(x,f,0) = \mathrm{R}(x,f)$$

I will also write

$$\mathrm{X}(x,a,1)$$

as the state that results from performing action a in state x.

1. Value Functions

In a Markov decision process, the future evolution of the process—in particular, the expected return—depends only upon the current state and on the policy that will be followed. If the process is in state x and the policy f is followed, the expected return will be written $V_f(x)$. That is,

$$V_f(x) = \mathbf{E}\,[\,R(x,f,0) \;+\; \gamma\,R(x,f,1) \;+\; \cdots \;+\; \gamma^n\,R(x,f,n) \;+\; \cdots\,]$$

Note that the value function could in principle be estimated by repeatedly simulating the process under the policy f, starting from state x, and averaging the discounted sums of the rewards that follow, the sums of rewards being taken over a sufficient period of time for γ^n to become negligible. But because value discounting is exponential, V_f also satisfies the following equation for all x:

$$V_f(x) = \rho(\,x, f\,) \;+\; \gamma\,\mathbf{E}\,[\,V_f(\,X(x,f,1)\,)\,]$$

For a finite-state problem, the evaluation function is known if its value is known for each state. The evaluation function may, therefore, be specified by the $|\mathbf{S}|$ linear equations

$$V_f(x) = \rho(x,f) \;+\; \sum_y P_{xy}(f)\,V_f(y) \qquad \text{for each } x$$

Thus in a finite-state problem, if ρ and $\mathbf{P}$ are known, the evaluation function can be calculated by solving a set of linear equations, one for each state. The calculation of the evaluation function for a policy is, therefore, straightforward but time-consuming.

2. Using the Evaluation Function in Improving a Sub-Optimal Policy

The point of constructing the evaluation function V_f for a policy f is that, once the evaluation function is known, it becomes computationally simple to improve the policy f if it is sub-optimal, or else to establish that f is in fact

optimal.

Suppose one wishes to know whether some proposed policy g will yield higher expected returns for all states than the existing policy f, for which the evaluation function V_f is known. This question might be phrased as 'Is g uniformly better than f ?' One way of determining whether g is uniformly better than f would be to compute V_g , and then to compare V_g with V_f over the entire state space. But calculating the evaluation function for a policy g is computationally expensive: to do this whole calculation for each proposed modification of the policy f would be extremely wasteful.

A simpler method of comparing f and g using V_f only is as follows. Consider the expected returns from following policy g for one step, and then following policy f thereafter. Suppose that the policy g recommends action b at state x, while policy f recommends action a. The expected return from starting at x, following policy g for one step (i.e. taking action b) and then following policy f thereafter is

$$Q_f(x,b) = \rho(x,b) + \sum_y P_{xy}(b)\ V_f(y)$$

This is much simpler to calculate than V_g , for to calculate $Q_f(x,g(x))$ it is only necessary to look one step ahead from state x, rather than calculating the whole evaluation function of g. I will call the quantity $Q_f(x,a)$ the *action-value* of action a at state x under policy f. Note that $Q_f(x,f(x)) = V_f(x)$, by definition.

To allow for the possibility that g may be a stochastic policy, I will define $Q_f(x,g)$ by

$$Q_f(x,g) = \sum_a Pr(g(x) = a)\ Q_f\ (x, a)$$

That is, $Q_f(x,g)$ is the expected return from starting at x, following policy g for one step, and then following policy f thereafter.

Action-values are useful for the following reason. Suppose that the expected return from performing one step of g and then switching to f is uniformly as good as or better than the expected return from f itself, that is

$$Q_f(x,g) \geq V_f(x)$$

for all states x. One can then argue inductively that g is uniformly as good as or better than f. Starting at any state x, it is (by assumption) better to follow g for one step and then to follow f, than it is to start off by following f. However, by the same argument, it is better to follow g for one further step from the state just reached. The same argument applies at the next state, and the next. Hence it is always better to follow g than it is to follow f. The proof is given in detail in Bellman and Dreyfus (1962), and in Ross (1983). The result is

Policy Improvement Theorem

Let f and g be policies, and let g be chosen so that

$$Q_f(x, g) \geq V_f(x) \quad \text{for all } x \in \mathbf{S}$$

Then it follows that g is uniformly better than f, i.e.

$$V_g(x) \geq V_f(x) \quad \text{for all } x \in \mathbf{S}$$

The significance of the policy improvement theorem is that it is possible to find uniformly better policies than f, if such exist, in a computationally efficient way. If the starting policy is f, then an improved policy is found by first calculating V_f, and then calculating the action-values

$$Q_f(x,a)$$

for each state x and each possible action a at x. A new policy f' is defined by choosing at each state the action with the largest action-value. That is,

$$f'(x) = a \in \mathrm{A}(x) : \max Q_f(x,a)$$

According to the policy-improvement theorem, f' is uniformly as good as or better than f. This process repeats: the evaluation function and action-values for f' may be computed, and a new policy f'' obtained, and f'' will be uniformly as good as or better than f'. With a finite Markov decision process, this process of policy improvement will terminate after a finite number of steps when the final policy f^* is found for which

$$f^*(x) = a \in \mathrm{A}(x) : \max Q_{f^*}(x,a)$$

In other words, no improvement can be found over f^* or V_{f^*} using the policy improvement theorem. Might f^* still be sub-optimal? The answer is no, according to the following theorem, proved in e.g. Bellman and Dreyfus (1962):

Optimality Theorem

Let a policy f^* have associated value function V^* and action-value function Q^*. If policy f^* cannot be further improved using the policy-improvement theorem, that is if

$$V^*(x) = \max_{a \in \mathrm{A}(x)} Q^*(x,a)$$

and

$$f^*(x) = a \text{ such that } Q^*(x,a) = V^*(x)$$

for all $x \in \mathrm{S}$, then V^* and Q^* are the unique, uniformly optimal value and action-value functions respectively, and f^* is an optimal policy. The optimal policy f^* is unique unless there are states at which there are several actions with maximal action-value, in which case any policy that recommends actions with maximal action-value according to Q^* is an optimal policy.

As a consequence of these two theorems, the following algorithm is guaranteed to find the optimal policy in a finite Markov decision problem:

f := arbitrary initial policy;

Repeat

1. calculate the evaluation function V_f

2. calculate the action-values $Q_f(x,a)$ *for all x, a*

3. update the policy for all x by

$$f(x) := a \quad \textit{such that } Q_f(x,a) = \max_{a \in A(x)} Q_f(x,a)$$

until there is no change in f at step 3.

This method of calculating an optimal policy is the *policy-improvement algorithm.* Note that the entire evaluation function has to be recalculated at each stage, which is expensive. Even though the new evaluation function may be similar to the old, there is no dramatic short cut for this calculation. There is, however, another method of finding the optimal policy that avoids the repeated calculation of the evaluation function. This method is known as *value iteration.*

2.1. Value Iteration

Value iteration is often a more efficient computational technique for finding the optimal evaluation function and policy. The principle is to solve the optimality equation directly for each of a sequence of finite-horizon problems. As the finite horizon is made more distant, the evaluation function of the finite-horizon problem converges uniformly to the evaluation function for the infinite-horizon problem.

A finite-horizon problem is a problem in which some finite number of actions are taken, which may have immediate rewards as in the infinite horizon problem, and then a final reward is given; the final reward depends only on the

final state.

Let $V^0(x)$ be the final reward for state x. V^0 is the optimal return after no stages: if there is an initial estimate of the optimal evaluation function, then this may be used as V^0, or alternatively V^0 may be an arbitrary guess. The only restriction that needs to be placed on V^0 is that it should be bounded; in finite-state problems, any V^0 is necessarily bounded. Let $V^n(x)$ be the optimal expected return achievable in n stages, starting in state x.

Once V^0 has been chosen, it is then possible to calculate V^1, V^2, $\cdots$ as follows. V^n can be calculated from V^{n-1} by

$$V^n(x) = \max_{a \in A(x)} \{\rho(x, a) + \gamma \sum_y P_{xy}(a)\, V^{n-1}(y)\}$$

The point of this procedure is that as $n \to \infty$, $| V^n - V_{f^*} | \to 0$ uniformly over all states. This proof is given in Ross (1983), and it is generalised in appendix 1 where is is used to prove the convergence of a learning method. The value-iteration algorithm is

V^0 := arbitrary bounded function over states;

i := 0;

Repeat

i := i + 1;

for each state x do

$$V^i(x) := \max_{a \in A(x)} \{\rho(x, a) + \gamma \sum_y P_{xy}(a)\, V^{i-1}(y)\}$$

until the differences between V^i and V^{i-1} are small for all x.

The method of value iteration that has just been described requires that, after V^0 has been defined, V^1 is calculated for all states, then V^2 is calculated for all states using the values of V^1, and then V^3 is calculated using the values of V^2, and so on. This is a simple and efficient way to organise the calculation

in a computer: for instance, the values of V^n and V^{n-1} may be held in two arrays, and once V^n has been computed, the values of V^{n-1} are no longer needed, and the array used to hold them may be re-used for the values of V^{n+1} as they are computed in turn. In addition to the computational simplicity, the time-horizon argument is intuitively clear.

However, in the learning processes I will consider, the learner may not be able to consider all the possible states in turn systematically, and so fill in the values for a new time-horizon. The value iteration method need not be carried out in a systematic way in which V^0, V^1, $\cdots$ are computed in sequence. Provided that the values of all states are updated often enough, the value iteration method also converges if the values of individual states are updated in an arbitrary order. As with policy iteration, the computation may be less efficient if the states are updated in arbitrary order, but it remains valid.

A different argument for the convergence of the value-iteration method runs as follows. At some intermediate stage in the calculation, let the approximate value function be U, and let the (as yet unknown) optimal value function be V. Let M be the maximal absolute difference between U and V, that is

$$M = \max_x | U(x) - V(x) |$$

If some state y is chosen, the value of $U(y)$ may be updated according to

$$U'(y) = \max_{a \in \mathbf{A}(y)} \{ \rho(y, a) + \sum_z P_{yz}(a)\, U(z) \}$$

where $U'(y)$ is the updated estimate of the value of y. It is possible to show

Local Value Improvement Theorem

If $| U(x) - V(x) | \leq M$ *for all states* x, *then for any state* y

$$| U'(y) - V(y) | \leq \gamma M$$

Proof:

By the definition of V,

$$V(y) = \max_{a \in \mathbf{A}(y)} \{ \rho(y,a) + \gamma \mathbf{E}[V(\mathrm{T}(y,a))] \}$$

$$= \rho(y,a^*) + \gamma \mathbf{E}[V(\mathrm{T}(y,a^*))]$$

for some optimal action a^*. Similarly,

$$U'(y) = \max_{a \in \mathbf{A}(y)} \{ \rho(y,a) + \gamma \mathbf{E}[U(\mathrm{T}(y,a))] \}$$

$$= \rho(y,a') + \mathbf{E}[U(\mathrm{T}(y,a'))]$$

for some action a', which is the optimal action with respect to U. Observe that

$$U'(y) \geq \rho(y,a^*) + \gamma \mathbf{E}[U(\mathrm{T}(y,a^*))]$$

$$\geq \rho(y,a^*) + \gamma \mathbf{E}[V(\mathrm{T}(y,a^*)) - M]$$

$$= V(y) - \gamma M$$

Similarly,

$$U'(y) \leq \rho(y,a') + \gamma \mathbf{E}[V(\mathrm{T}(y,a')) + M]$$

$$\leq \rho(y,a^*) + \gamma \mathbf{E}[V(\mathrm{T}(y,a^*)) + M]$$

$$= V(y) + \gamma M$$

so the proposition is proved.

Note that U' will not necessarily be uniformly more accurate than U. However, since the maximal error of U' is guaranteed to be geometrically smaller than the maximal error of U, it follows that the value iteration method is guaranteed to converge, and that the maximum error of the estimated evaluation function is guaranteed to decline geometrically, the rate of decline depending upon the discount factor γ.

Note also that the updating of the estimated value function may be done one state at a time, in any arbitrary order—it is not necessary to perform the updates systematically over the state-space as in the value-iteration algorithm. The estimated value function will converge to the true value function as the total number of updates tends to infinity provided that all states are updated infinitely many times.

The Local Improvement Theorem has an immediate corollary:

The Approximation Assessment Corollary

If U is an approximation to V, and

$$\max_x |U(x) - V(x)| = M$$

there is at least one x such that

$$|U'(x) - U(x)| \geq (1-\gamma)\, M$$

Equivalently,

$$\max_x \frac{|U'(x) - U(x)|}{(1-\gamma)} \geq M$$

This is a most useful result, for it yields an efficient method of determining whether some function U is a good approximation to the optimal evaluation function V. Given an approximate value function U, one can obtain an upper bound on $\max_x |U(x) - V(x)|$ by carrying out one pass of the value iteration algorithm and observing $\dfrac{\max_x |U'(x) - U(x)|}{(1-\gamma)}$.

2.2. Discussion

This has been a very brief account of the principle of dynamic programming applied to Markov decision problems. The main point I wish to convey is that the computation consists of three processes: computing the evaluation

function for the current policy; computing action-values with respect to the current evaluation function; and improving the current policy by choosing actions with optimal current action-values. These three procedures may be carried out repeatedly in succession, as in the policy-improvement method, or else they may be carried out concurrently, state by state, as in the value-iteration method. In either case, the policy and the evaluation function will eventually converge to optimal solutions. In this optimisation process, there are no local maxima, and uniform improvements can be found at every step. Furthermore, the minimum improvement theorem shows that convergence of either method is rapid.

These results together paint a picture of an optimisation problem that is as benign as it is possible for an optimisation problem to be; and small, finite problems are indeed benign. However, in practical problems, the state-space may be extremely large, and it may be impossible ever to examine all parts of the state-space. Although it is not necessary to examine the whole state-space to find guaranteed improvements to the current policy, it *is* necessary to examine the entire state-space to be *sure* of finding the optimal policy. The minimum improvement theorem states only that there is *some* state for which an improvement of a certain size may be found, not that such improvements can be found for any state. Hence, if it is not possible to examine the entire state space, it is not in general possible to establish whether the current policy is indeed optimal.

Chapter 5
Modes of Control of Behaviour

An animal or agent chooses what actions to perform by doing some computations on internally stored information. The type of computation done and the type of information used constitute the *mode of control of behaviour.*

Different modes of control require different learning algorithms, so it is necessary to classify possible modes of control of behaviour before considering learning algorithms.

In classifying modes of control, the first distinction to make is between modes in which the agent looks ahead and considers the future states and rewards that would result from various courses of action, and modes in which the agent decides what to do by considering only the current state. If the agent considers the effects of different courses of action, it may be said to use a 'model-based' or 'look-ahead' mode of control, and if it considers only the current state, it may be said to use a 'primitive' mode of control. Hierarchical control will be considered in chapter 9.

1. Look-Ahead

In controlling its actions by look-ahead, an agent uses an internal model of its world to simulate various courses of action mentally before it performs one of them. That is, the agent considers the likely rewards from each of a number of possible courses of action, and chooses that course of action that appears best. For example a hill-walker may need to consider how best to cross a mountain stream with stepping stones without getting his feet wet. He may do

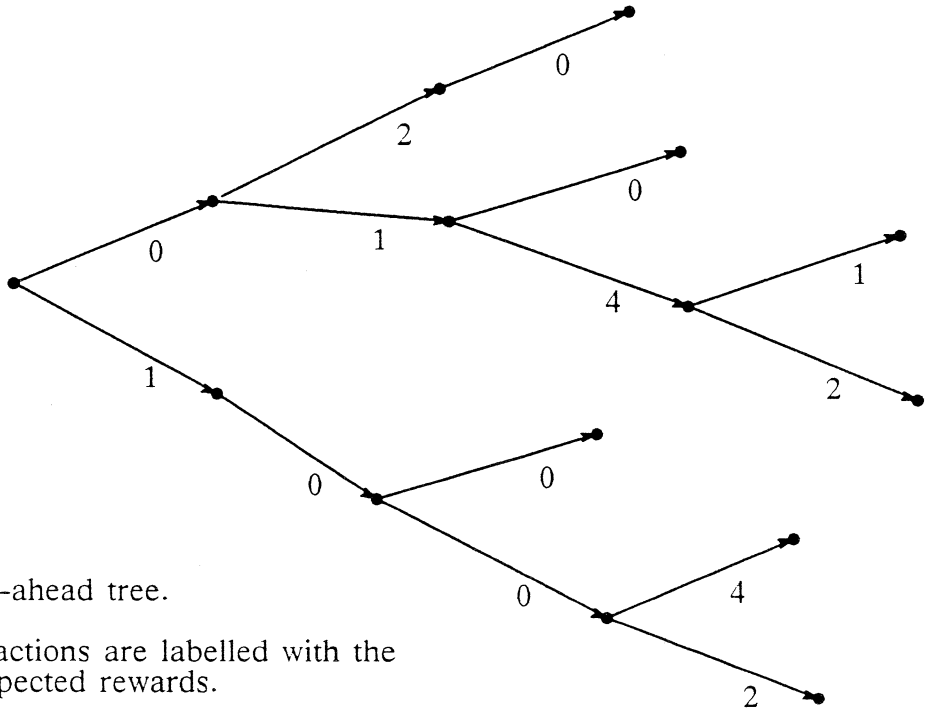
this by tracing out various possible routes from rock to rock across the stream, and for each route he should consider how likely he is to fall in, taking into account the distances between the rocks, and how slippery they appear.

The method of deciding what to do by 'look-ahead' is important because it is much used in artificial intelligence. For example, chess-playing programs decide what to do at each turn by tracing out many thousands of possible sequences of further moves, and considering the desirabilities of the positions that result.

In abstract terms, the essential abilities that an agent must have to use look-ahead control are: a transition model, a reward model, and an ability to consider states other than the current state. An agent with these abilities can in principle compute the best course of action for any finite number of time-steps into the future.

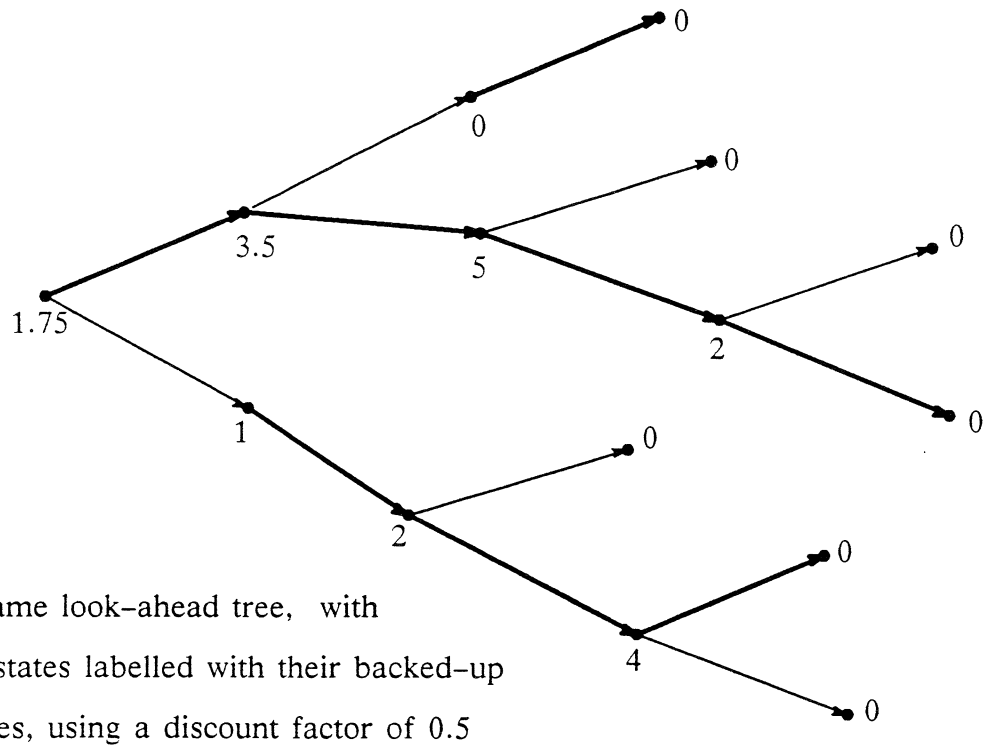
The computation for this may be laid out in the form of a tree, as illustrated as the upper tree in the diagram overleaf. Let us assume for the moment that actions have deterministic effects, so that a unique state results from applying any action. Each node represents a state, and each branch represents an action, leading from one state to another. The root of the tree—the leftmost state in the diagram—is the current state. Paths from the root through the tree represent possible sequences of actions and states leading into the future. Each action is labelled with the expected reward that would result from it.

If the tree is extended fully to a depth of n, so that it represents all possible courses of action for n time-steps into the future, then it is possible to determine the sequence of moves that will lead to the greatest expected reward over the next n time steps. The path from root to leaf in the tree that has maximal total discounted reward represents the optimal sequence of actions. This path may be found efficiently as follows. The nodes in the tree are labelled



A look-ahead tree.

The actions are labelled with the expected rewards.



The same look-ahead tree, with the states labelled with their backed-up values, using a discount factor of 0.5

The action at each state that is optimal in the look-ahead tree is marked in bold.

with their values, the value of a node being the maximal possible expected return obtainable by starting in that node and following a course of action represented in the tree. The values may be computed efficiently by starting with the leaves and working back to the root, labelling each node of the tree with its value. The node labels for the previous tree are shown on the lower tree in the diagram, assuming, for convenience, a discount factor of 0.5

The optimal path from each node—that is, the optimal path within the tree of possibilities—is drawn as a thicker line. The node values are calculated as follows. Leaf nodes are given zero value, since no actions for them are represented in the tree. If x is an interior node of the tree then the estimated value of x, written $V(x)$, is given by

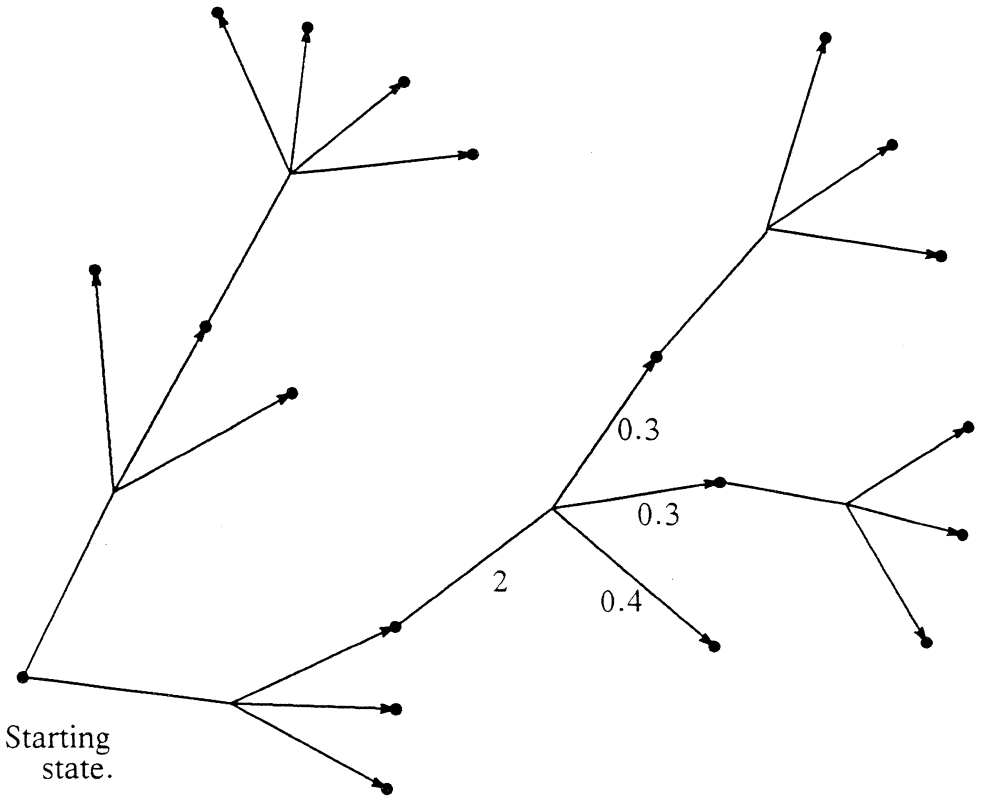
$$V(x) = \max_a \{\rho(x,a) + \gamma V(\mathrm{T}(x,a))\}$$

where $\mathrm{T}(x,a)$ ranges over all the successor nodes of x as a varies.

With this mode of control, the action the agent chooses to perform is the first action on the path with maximal expected return. Once the agent has done this, it chooses the next action in the same way as before, by extending a new tree of possibilities to the same depth as before, and then recomputing the path with maximal expected return.

This mode of control is equivalent to following the policy that is optimal over a time-horizon of n steps—the nth policy calculated in the value-iteration method of dynamic programming.

If the state-transitions are not deterministic, then the tree of possibilities becomes much larger. This is because each action may lead to any of a number of possible states, so that at each level, many more possible transitions have to be considered. An example of a (small) tree of possibilities for actions with probabilistic effects is shown in the diagram overleaf. In this case, actions are



A small stochastic look-ahead tree.

Each action may have several possible results, so that the size of the tree grows rapidly as more actions are considered.

The actions need to be labelled both with the expected reward and with the probability of each result; only one action has been labelled as an example.

represented by multi-headed arrows, because each action may lead to any one of several states. Not only must each action be labelled with the expected immediate reward—each possible transition between a state x and a successor state y must be labelled with $P_{xy}(a)$.

It is still possible to calculate the optimal course of action over the next n moves, with the difference that future actions will depend on which states are reached. With probabilistic actions, a 'course of action' itself has the form of a tree, a tree that specifies a single action for each state reached, but which branches to take account of the possibility of reaching different states.

The best course of action may be found by a procedure that is similar to that for the deterministic case. The value of a node is still defined as the maximal expected return that can be obtained by starting from that node and following a course of action represented in the tree, but $V(x)$ is given by

$$V(x) = \max_a \{\rho(x,a) + \gamma \sum_y P_{xy}(a)V(y)\}$$

That is, the agent must take into account all possible states that could result when computing the expected return from performing the action.

It is only computationally possible to extend a tree of possible courses of action to a finite depth. If the agent wishes to follow a policy that yields optimal total discounted rewards, then it must extend the tree of possibilities far enough into the future for the return from subsequent rewards to become negligible.

However, the number of nodes in the tree will in general grow exponentially with the depth of the tree. It is therefore usually impractical to consider courses of action that extend for more than a short time into the future—the number of possible sequences of state transitions becomes far too large. This is the so called 'combinatorial explosion' of artificial intelligence.

If the agent can extend the tree of possible courses of action far enough into the future, then it can compute the optimal policy and value function. However, it is just this vast calculation that dynamic programming avoids, if the state-space is small enough.

2. Look-Ahead with an Evaluation Function

The amount of computation required for look-ahead control may be greatly reduced if the value function, or even an approximate value function, is known.

If the agent can compute an evaluation function, then it may choose its actions by considering a much smaller tree of possibilities than would be needed if the naive look-ahead method of the previous section were used. Recall that in computing the values, the leaf states were assigned values of zero in the naive look-ahead method. If, instead, the leaf nodes are given estimated values, the values computed for interior nodes of the tree may be more accurate.

2.1. One-Step Look Ahead

In fact, if an accurate optimal value function is available, the agent need only look one step ahead to compute the optimal action to take. That is, the agent selects the action to perform by finding an action for which

$$\rho(x,a) + \gamma \sum_{y} P_{xy}(a) V(y)$$

is maximal. Note that a policy determined in this way from V will be optimal, because this is just the optimality condition of dynamic programming. Even if the estimated value function is not optimal, the size of tree needed to compute an adequate policy may be substantially reduced.

2.2. One Step Look Ahead

The special case of one-step look-ahead control is important because the agent only needs to consider possible actions to take in the current state: the agent needs to imagine what new states it might reach as a result of actions in the current state, and it must estimate the values of these new states, but it need not consider what actions to take in the new states.

It is possible that there are examples of agents that can consider the consequences of actions in the current state, but which cannot consider systematically what to do in subsequent states. For example, in playing chess one considers the positions that would be reached after playing various sequences of moves: the longer the sequence of moves, the more different the appearance of the board would become, and the more difficult it is to consider what further moves would be possible, and what the values of the positions are.

2.3. Discussion

Much research in artificial intelligence has been devoted to finding methods of using an evaluation function to reduce the size of the tree of possibilities that it is necessary to construct to determine the optimal action to take. The formulation of the problem is, however, usually in terms of minimising total distance to a goal state, rather than of maximising total discounted reward. Pearl (1984) is a standard reference on this topic.

In conclusion, it is only feasible for an agent to choose its actions using naive look ahead if

- The agent has accurate transition and reward models.
- The effects of actions are deterministic or nearly so.

- The number of alternative actions at each stage is small.
- The agent seeks to find an optimal course of action for only a few steps into the future.
- The agent has enough time to consider the tree of possible courses of action.

If the agent has can compute a value function or an approximate value function, the amount of computation required may be greatly reduced, and the method of look-ahead becomes more feasible. The limiting case is that of one-step look-ahead, which requires an accurate value function.

3. Primitive Modes of Control

In primitive control, the agent does not consider future states. This means that primitive control methods are suitable for simple animals in stochastic environments. Although higher animals are capable of considering the consequences of their actions, this certainly does not mean that they govern all of their behaviour using look-ahead. If actions must be chosen quickly, or if the environment is stochastic, or if the effects of actions are poorly understood or difficult to predict, then it is likely that higher animals may also use these 'primitive' control methods.

There are three basic types of primitive control: by policy, by action-values, and by value function alone. The amount of information that the agent may need to store in order to represent any of these three functions may on occasion be far smaller than the amount of information that would be needed to represent a transition or reward model. Furthermore, each type of primitive control may be learned, by methods that will be described in the following chapters.

3.1. Explicit Representation of Policy

The most direct way for the agent to choose what actions to perform is for it to store a policy f explicitly. In effect, the agent then has a 'situation-action rule': in any state x it performs the action $f(x)$.

However, if an agent only has a representation of a policy, then it is not able to compute values of states or action-values without considerable further computation. As a result, efficient unsupervised learning of a policy alone may be difficult, and some additional internal representation either of action-values or of a value function may be helpful for learning.

3.2. Explicit Representation of Action Values

In choosing actions by one-step look-ahead with a value function, the agent computes the action value for each possible action, and then chooses an action with maximal action value. That is, the action value of an action is

$$Q(x,a) = \rho(x,a) + \gamma \sum_{y} P_{xy}(a) V(y)$$

and the agent chooses an action a for which $Q(x,a)$ is maximal. If the agent were to store the values of the function Q explicitly, instead of computing them by a one step look ahead, then it could choose the action a for which the stored value of $Q(x,a)$ was maximal. This is primitive control according to stored action values.

This type of primitive control has the advantage that the agent represents the costs of choosing sub-optimal actions. If special circumstances arise, so that exceptional rewards or penalties attach to some actions, then the agent may choose sub-optimal actions with action values that are as high as possible.

Note that the agent does not need to store the actual values: any function of x and a that is monotonically increasing in $Q(x,a)$ at x will serve as well,

because the agent just chooses the action with maximal $Q(x,a)$. It is, therefore, still possible to represent a policy using an inaccurate action-value function which has the same maxima at each state as the value function.

3.3. Control Using a Value Function Alone

Selfridge (1983) describes a mode of control of behaviour that he terms *run and twiddle*. Loosely, this may be defined as

If things are improving, then carry on with the same action.

If things are getting worse, then try doing something—anything—else.

Perhaps the classic example of control of this type is that of the motion of bacteria, described by Koshland (1979). Certain bacteria are covered in motile cilia, and they can move in two ways: they may move roughly in a straight line; or they can 'tumble' in place, so that they do not change position but they do change direction. A bacterium alternates between these two types of motion. The bacterium seeks to move from areas with low concentrations of nutrients to areas with high concentrations. It does this by moving in the direction of increasing concentration of nutrients. Too small to sense changes in concentration along its length, the bacterium can nevertheless sense the time variation of concentration as it swims along in a straight line.

A bacterium will continue to swim in its straight-line mode as long as the concentration of nutrients continues to increase. However, if the concentration begins to fall, the bacterium will stop swimming in a straight line, and start tumbling in place. After a short time, it will set off again, but in a new direction that is almost uncorrelated with the direction in which it was swimming before. The result of this behaviour is that the bacteria tend to climb concentration gradients of desirable substances, and cluster around sources of nutrients.

Although this is perhaps the most elegant example of run and twiddle behaviour—and particularly impressive because it is so effective a strategy for such a simple organism—run and twiddle may also be used by higher animals. Selfridge (1983) describes the behaviour of moths following concentration gradients of pheromones to find their mates as a form of 'run and twiddle' with some additional control rules. But even humans may use run and twiddle sometimes: who has not tried a form of 'run and twiddle' while, say, trying to work the key in a difficult lock, or in trying to fly a steerable kite for the first time, when one does not know which actions affect the motion of the kite? Run and twiddle is a suitable control strategy if the agent cannot predict the effects of its actions, or, perhaps, even represent adequately the actions that it is performing.

It is necessary to distinguish between run and twiddle control by immediate rewards from control according to a value function and immediate rewards. Consider some hypothetical bacteria that control their movement in the manner described above. A bacterium's aim is to absorb as much nutrient as possible: the amount of nutrient absorbed per unit time is the 'immediate reward' for that time. If the bacterium decides whether to tumble or to continue to move in a straight line on the basis of the change in the concentration of *nutrients*, then it is controlling its actions according to immediate rewards. In contrast, consider a fly looking for rotten meat. The fly could find rotting meat by following concentration gradients of the smell of rotting meat, in a similar way to that in which the bacteria followed the concentration gradients of nutrients. The meat, however, is the reward—not the smell: the current smell may be used in the definition of the current state, on which a value function may be defined. In this case, therefore, the state might be defined as the current smell.

One method of run and twiddle control using a value function works as follows. The value of the current state is an estimate of the expected return. Suppose an action a is performed on a state x, resulting in a new state y and an immediate reward r. Let the value function be V. Then the return expected from following the control policy at x is $V(x)$. The return that results from performing a may be estimated as

$$r + \gamma V(y)$$

The action is deemed successful if the resulting return is better than expected, and it is deemed unsuccessful if return is less than expected. That is, a is successful if

$$r + \gamma V(y) \geq V(x)$$

and unsuccessful if

$$r + \gamma V(y) < V(x)$$

A control rule is 'If the last action was successful, then repeat that action, otherwise perform another action.'

This is a very simple method of control. More complex control rules are possible. The essential feature of 'run and twiddle' control is that

- *The returns estimated using the evaluation function are used in choosing subsequent actions.*

As in action-value control, run and twiddle need not necessarily use the value function itself: it may also use functions that are monotonically related to the value function. Such a function of states might be called a *desirability function.* An alternative description of these control methods is as *desirability gradient* methods, since the control strategy is a form of stochastic hill-climbing of a desirability function. Many such control methods are possible.

A published example of an artificial learning system which chooses its actions by a desirability-gradient method is that of Connell and Utgoff (1987), who describe a learning system for the cart and pole problem described in chapter 2. This problem is particularly simple in that there are only two possible actions in each state, so that it is suitable for desirability gradient control. The desirability function is not a value function. It is constructed by using ad hoc heuristic rules to identify a certain number of 'desirable' and 'undesirable' points in the state space during performance. The desirability function is constructed assigning a desirability of 1 to each desirable point, and of -1 to each undesirable point, and then smoothly interpolating between the desirable and undesirable points. The undesirable points are the states of the system after a failure has occurred; the desirable points are states after which balancing continued for at least 50 time steps, and which satisfy certain other conditions. In Connell and Utgoff's system, the controller continues to perform the same action until the desirabilities of successive states start to decline; when this happens, it switches to the other action. Their method was highly effective for the cart and pole problem, but it is of course based on an ad hoc method of constructing a desirability function.

It is possible to learn value functions for run and twiddle methods by incremental dynamic programming. I do not know of general, principled methods for learning desirability functions of other kinds. It is possible that Connell and Utgoff's method of identifying desirable and undesirable states during performance and then interpolating between them can be applied more generally.

4. Hybrid Modes of Control

A mode of control is simply a method of choosing an action on the basis of a knowledge of the current state: given the current state, one or more alternative actions are recommended for performance. There is, therefore, no reason why different modes of control may not be used as alternatives to one another. An agent may have alternative modes of control that it can use, or else it may use different modes of control in different parts of the state space.

For example, an agent may be able to predict the effects of some actions, but be unable to predict the effects of others. It may, therefore, evaluate some actions with one-step look-ahead, and it compare these values with stored action-values for the other actions.

5. Learning Faster Modes of Control

This is an appropriate point to describe one type of learning which I am *not* going to consider further. This is the learning of a fast mode of control using a slow mode of control to provide training data. This type of learning is analogous to the 'imitate the teacher' learning considered in chapter 2.

Suppose, for example, that an agent can use look-ahead to control its actions, but that this mode of control is inconveniently slow, or that it consumes valuable mental resources that could be employed on some other activity. It would, therefore, be advantageous for the agent to acquire a faster or computationally less expensive form of control.

While using the slow mode of control, the learner may collect a set of situation-action pairs that may be used as training data for learning a faster mode of control. The situation-action pairs, laboriously calculated by look-ahead, for example, may be used to learn situation-action rules inductively. Learning of this type has been studied in artificial intelligence, by Mitchell,

Utgoff, and Banerji (1983).

Another approach has been suggested by Laird, Rosenbloom, and Newell (1986) who propose 'chunking' as the general learning mechanism. 'Chunking' is the process of cacheing action sequences performed, and then treating these stored action sequences as single actions. Once an action sequence has been found by look-ahead search, and successfully applied, the entire action sequence is associated with the starting state, generalised if possible, and then stored, ready to be applied again in the same situation.

The particular method of chunking has the limitation that as more and more action-sequences are stored in this way, the search for an appropriate action sequence may become more and more lengthy, so that the actual speed of the mode of control may sometimes get worse. An abstract model of this phenomenon has recently been given by Shrager et al (1988).

Chunking appears to be *a* possible general learning mechanism—but is is not plausible to claim that it is *the* general learning mechanism.

Learning a fast control mode using by (internal) observation of the performance of a slow control mode may have considerable importance in learning many skills. In skill-learning, slow closed-loop control is gradually altered to the faster open-loop control.

6. Conclusion

There are, therefore, a number of possible modes of control of action, each based on the use of a different type of internally represented knowledge. For each control mode, there is at least one method of learning.

The possible modes of control of action can be divided according to whether or not the agent can predict state-transitions and rewards—that is, according to whether the agent can look ahead. From now on I will concentrate

on those modes of control of action in which the agent cannot predict the effects of its actions in the sense of predicting state-transitions or rewards. These modes of control are:

- by using an explicitly represented policy, and choosing the action recommended by the policy

- by using explicitly represented action-evaluations, and choosing an action with maximal action-value

- by following the gradient of a desirability function, by preferentially performing those actions that currently appear to lead to net increases in desirability.

These modes of control are in a sense more primitive than those in which the agent uses an internal model of its world to look ahead. One of the main contributions of this research is to show that it is still possible to optimise these primitive modes of control through experience, without the agent ever needing to look ahead using an internal model.

Chapter 6
Model-Based Learning Methods

Learning in which the agent uses a transition and a reward model will be termed 'model-based' learning. There are essentially two possibilities: either the learner knows the transition and reward models at the start, or else it acquires them through experience.

1. Learning with Given Transition and Reward Models

Consider a learning problem in which an agent initially knows accurate transition and reward models, but does not know the optimal value function or policy. In other words, suppose the agent has all the information it needs to calculate the optimal solution by one of the standard dynamic programming methods, but that it has not done so. This might not appear to be a learning problem at all: in principle, the agent could use its initial knowledge to calculate the optimal solution. But this may not be possible.

One reason is that the agent may not have the leisure or the computational ability to consider all states systematically in carrying out one of the conventional optimisation methods. An animal might know the layout of its territory, and, if located in any spot, it might know how to reach various nearby places: however, it might be unable to use this knowledge to plan ahead because it might be unable to consider alternative possible routes from a place different from its current location. A specific reason an agent might not be able to compute an optimal policy or course of action is that it may not be able to systematically consider alternative courses of action in states different from its

current state: the ability to use the transition and reward models might be tied to the agent's current situation.

In other problems, it is completely impractical to compute the optimal policy at all, because the state-space is too large. The classic examples of such problems are board games, in which the rules of the game are easy to state but the winning strategy is hard to find. The whole problem of learning to play a game such as solitaire, draughts, or chess is not in understanding the rules but in improving the quality of legal play.

In learning from experience, the agent will not be surprised by any rewards it receives, or by any state-transitions it observes, since it already possesses transition and reward models. The new information is in the form of the sequence of states that are visited. One method of improving an initial evaluation function is to carry out a value-iteration operation at each state visited. The value-iteration can be neatly combined with look-ahead control of action, since many of the same computations need to be performed for both.

Let the agent's approximate current value function be U, and suppose the agent controls its actions by one-step look-ahead. If the current state is x, then the action chosen by one-step look-ahead according to U will be an action a that maximises

$$\rho(x,a_i) + \gamma\, U(T(x,a_i))$$

among actions a_i possible in x; that is, an action with maximal estimated return according to U. But the value iteration at x is to set $U(x)$ to be equal to the maximum estimated return according to U, that is:

$$U'(x) \leftarrow \max_a \; \rho(x,a_i) + \gamma\, U(T(x,a))$$

So the improvement of U using value-iteration can be done as a by-product of one-step look-ahead control.

If the agent performs a multi-step look-ahead search, so that it constructs a large tree of possibilities, then it is also valid—indeed usually better—to combine many-step look-ahead control with value iteration in the same way.

This type of 'learning' is just value-iteration carried out at the states the agent happens to visit or to consider, rather than value iteration carried out systematically over the whole state space. If the agent does not repeatedly visit or consider all states, then the learning may not converge to an optimal value function because U may remain perpetually in error on states that the agent does not visit. There is also no guarantee that each value iteration will be an actual improvement of U towards the optimal value function: if U is correct for some state x but in error for the successors of x, then the value iteration may worsen U at x. But according to the local improvement theorem, $U'(x)$ cannot be in error by more than γ times the maximal error of U. If there is a subset of the state-space that the agent covers repeatedly, then the agent must develop an optimal value function for the problem restricted to that set of states.

1.0.1. Samuel's Checker-Playing Program

The classic implementation of this method of learning is Samuel's (1963, 1967) program for playing checkers. This program refined its position-evaluation heuristic during play by what was essentially value-iteration. Two learning methods were used.

One method was to cache certain board positions encountered together with their values estimated from a look-ahead search. The store of cached positions and values can be viewed as a partial function from states (board positions) to their estimated values—a partial implementation of U. If a cached position is encountered during a look-ahead search, the search need not go beyond the cached position—the cached value may be used as an estimate of

the value of the look-ahead tree from the cached position, so that the effective size of the look-ahead search is increased. Note that the values associated with the cached positions should in principle be updated periodically because the look-ahead search used to compute the cached value did not take advantage of the positions and values cached subsequently: it is unclear whether Samuel's program did this. Cacheing and the recomputation of cached values is thus a form of value iteration. The effect of the cacheing may be described either in terms of increasing the effective size of look-ahead search, or equally in terms of storing and improving an evaluation function by value iteration.

The disadvantage of cacheing in a game such as checkers is that it is impractical to cache more than a tiny fraction of all positions. The second learning method that Samuel's program used was a method of developing and improving a value function that could be applied to any position, not just to cached positions. This was a parametrised function of certain features of board positions, and the parameters could be altered by a gradient method to fit the function to revised values. The parameters were incrementally adjusted during play according to a value-iteration method.

This method of learning is similar to methods I will consider later. The danger in adjusting a parametrised value function is that in changing the parameters, the value function changes for many positions other than the current position. There is, therefore, the possibility that different adjustments will work in opposite directions and the overall quality of the value function will deteriorate. Samuel reports that he found that this method of adjustment of the evaluation function was far from reliable, and that occasional manual interventions were necessary.

A further danger of the value-iteration learning method is that in checkers, where the payoff occurs at the end of the game only, values must be adjusted

to be consistent over very long chains of moves. Although an approximate value function might be locally near-consistent according to value-iteration, it might nevertheless be largely wrong. This simply reflects the fact that value iteration is not always guaranteed to improve the value function at every stage. Samuel's formulation of checkers did not use discounted rewards; in this case, although value iteration would in principle be guaranteed to converge to the optimal value function if the iteration could be carried out repeatedly over all states, a single value-iteration, even over all states, could not even be guaranteed to reduce the maximum error of the value function.

Samuel's program did not really need to play games of checkers to learn to improve its evaluation function: in principle, the learning could have been done by performing value-iteration on an arbitrary collection of checkers positions, instead of doing it on the positions encountered during the games the program played. As far as I know, Samuel did not try the experiment of comparing learning from an arbitrary collection of positions to learning from the positions encountered during play, and it seems probable that learning from arbitrary positions would have been worse. On the other hand, if the arbitrary positions were taken from games between human expert players, the learning might have been better.

The reason why it may often be useful to perform value iteration at the states encountered during performance is that many arbitrarily generated states might never be reached in actual performance. In chess, for example, only a tiny proportion of random configurations of pieces on the board could plausibly occur as game positions. Even if a chess-player developed an ability to choose good moves from random positions, this hard-won skill might not be applicable in actual play since the type of position encountered in games between people would be qualitatively different.

In spite of the limitations I have mentioned, Samuel's program worked very well. In fact, it may still claim to be the most impressive 'learning program' produced in the field of artificial intelligence, as it achieved near-expert levels of performance at a non-trivial game. I have argued that both of the learning methods that Samuel used may be regarded as forms of value iteration applied at the states the program visits during play. In the case of checkers, this type of learning cannot be guaranteed to improve the value function, but the analysis in terms of incremental dynamic programming provides a framework for explaining both the program's successes and its limitations.

1.1. Learning with Adaptive Transition and Reward Models

A different type of learning problem arises when the learner does not possess accurate transition and reward models initially, and the learning task is both to learn transition and reward models and to optimise the value function.

A learner may have initial approximate transition and reward models, and an initial approximate value function; it may improve both its models and its value function through experience.

The problem of improving the transition and reward models is a problem of inductive inference. In a finite system, the most general way of inferring the transition and reward models is to visit all states repeatedly and to try out all possible actions repeatedly in each state; it is then possible to keep counts, for each state-action pair, of the numbers of transitions to each other state, and to record the rewards received. The relative frequency of each transition may be used to estimate its probability, and the records of rewards may be used to estimate the expected reward as a function of state and action. The construction of the transition and reward models is a problem of *system identification*, which may be described as a problem of statistical estimation complicated by the need

to visit a sufficient variety of states to obtain the necessary empirical data.

An obvious and straightforward approach to combining model estimation, action, and learning is for an agent to maintain current estimated transition and reward models that are incrementally updated according to experience. The agent also maintains a current estimated value function, and the agent uses its current estimated models and its current estimated value function to choose its actions by look-ahead. Because both the models and the value function may be in error, the problem of choosing an appropriate look-ahead method in the early stages of learning is difficult.

However, as learning goes on, the transition and reward models will become progressively more accurate, so that the learner's policy and value function approach optimality asymptotically.

1.2. Relationship to Self-Tuning Control

Many self-tuning control problems are of this type. In self-tuning control theory, it is usual to assume that the structure of a model of the process is known, and that what remains to be done is to estimate the values of a (relatively small) number of initially unknown parameters. It is taken for granted that once the parameter values of the model have been estimated, an appropriate policy to follow may be computed immediately—the computation of a policy or value function from the estimated model is not regarded as a part of the adaptive process. A common assumption is that an appropriate policy to follow given uncertain estimates of the parameter values is a policy that would be optimal if the estimated parameter values were correct; this is known as a 'certainty equivalence' assumption. If the estimation process is consistent so that the parameter estimates do eventually converge to their true values, then 'certainty equivalence control' will ultimately converge to an optimal control

policy, even if a certainty-equivalent policy is not optimal while the parameter estimates are uncertain.

This method of empirical model identification combined with certainty equivalent control is conceptually simple, but there is a snag that may sometimes arise. The problem is that the early estimates of the model parameters may be in error, and if the agent (i.e. the controller) follows a certainty-equivalent policy for the erroneous parameter estimates, then it may limit its subsequent experience, so that the wrong parameter estimates are never corrected, and performance never improves. To ensure that the learning agent does obtain sufficiently varied experience for it to be sure of estimating the parameter values correctly eventually, it may be necessary for the agent to perform experiments as part of its learning strategy.

A number of papers have been published in the control literature on self-tuning control of Markov decision processes, mainly considering the average reward criterion rather than the discounted reward criterion. Mandl (1974) proved that a certainty-equivalence approach to self-tuning control of finite Markov decision processes would converge under certain restrictive conditions. Kumar and Becker (1982) criticise Mandl's approach as requiring too restrictive conditions, and they propose a method based upon intermittent experimentation, with the intervals between experiments growing progressively longer. The experiments they suggest consist of following the certainty equivalent control policy for a randomly chosen set of parameter values until the starting state is revisited (one of their requirements is that the chain should be recurrent for all certainty equivalent policies). They prove that the average performance of their self-tuning controller taken from the start of the run will tend to the optimal possible value, although their controller will, of course, continue to behave sub-optimally during its intermittent experiments.

There are two differences between mainstream self-tuning control theory and the learning methods I have described in this section. The first, and most important, is that I regard the computation of the optimal value function as a part of the learning process, rather than as something which can be done instantaneously. The second difference is one of emphasis: most self-tuning control theory is concerned with linear systems or with non-linear systems for which it is assumed that the form of the model is known, and that the values of only a relatively small number of parameters need to be determined. In the problems I wish to consider, the uncertainty about the form of the model may be much greater. In problems with continuous state-spaces, the models will be constructed from large families of explicitly represented functions, and it will also be possible to consider other problems in which the state-space is discrete.

Chapter 7
Primitive Learning

By 'primitive' learning, I mean learning in which the agent does not have and does not estimate transition or reward models. Instead, the agent develops a policy and evaluation function directly. In principle, the agent may develop an optimal policy and value function without having to remember more than one past observation, and without being able to predict the state-transitions or the immediate rewards that result from its actions. Although such an agent has only 'primitive' abilities, it may still be able to learn complex and effective patterns of behaviour.

Primitive learning may be described as incremental dynamic programming by a Monte-Carlo method: the agent's experience—the state-transitions and the rewards that the agent observes—are used in place of transition and reward models. In primitive learning, the agent does not perform mental experiments—it cannot, for it has no internal models. The agent performs actual experiments instead.

Methods of primitive learning are described and discussed from section 3 to the end of the chapter; these are the main results of the thesis.

1. Importance of Primitive Learning

Primitive learning is important because it requires only simple computations and because the transition and reward models are often difficult to construct and to represent if the environment is complex.

Transition models, in particular, may be complex. If actions do not have deterministic effects—if performing an action may lead to any of a number of possible states—then the transition model is a mapping from state-action pairs to probability distributions over states. A large amount of information may need to be stored to represent this mapping, and a large number of experiments might need to be performed to acquire it inductively.

For a finite Markov decision process, the reward model may have $|S||A|$ parameters, and the transition model may be larger with as many as $|S|^2|A|$ parameters, where $|S|$ and $|A|$ are the numbers of possible states and actions respectively. In contrast, a value function requires only $|S|$ parameters, and even a stochastic policy requires at most $|S||A|$ parameters. Thus, as Howard (1960) remarked, the practical applicability of the conventional approaches to policy optimisation in finite Markov decision processes is severely limited by the need to represent the transition model. Even if other methods of representing the transition model are used, it will still often be the most complex data object in the learning system, since it must represent a mapping from one large set $\mathbf{S} \times \mathbf{A}$ into another large set—the set of probability distributions over **S**.

A more subtle reason why it may often be difficult to construct a suitable transition model is the following. No model can represent the world completely—in constructing any model, it is necessary to decide which aspects of the world to ignore and which to represent. However, consider an agent that is constructing a transition model with which to construct an optimal policy: how can the agent know which aspects of the world are relevant for constructing the optimal policy, and which are not? If the agent models the world in unnecessary detail then it will waste resources. On the other hand, if the agent models the world in too little detail, and so ignores some relevant aspects of the state of the world, then a policy that is optimal according to its model may

not in fact be optimal at all.

The trouble is that, in model-based learning, the agent in effect replaces the real world with an internal model, and constructs an optimal policy by performing mental experiments with its internal model. If the agent considers the agreement of its model with the world and the optimality of its policy with respect to the model separately, then it cannot determine whether its model is an adequate representation of the world for the purposes of constructing an optimal policy.

To determine what aspects of the world are relevant to the value of a course of action, the agent must experiment in the world and keep track of the returns that result from different states. The agent must actually observe the returns that its policy brings, and it must choose to distinguish states according to whether they lead to differing returns. As will become apparent, this is what is done in primitive learning.

1.1. Information Available to the Learner

The learner's task is to find an optimal policy after trying out various possible sequences of actions, and observing the rewards it receives and the changes of state that occur. Thus an episode of the learner's experience consists of a sequence of triples of states, actions, and rewards:

$$\begin{array}{ccc} x_1 & a_1 & r_1 \\ . & . & . \\ . & . & . \\ x_n & a_n & r_n \end{array}$$

The x_k are *observed* by the learner; the a_k are *chosen* by the learner; and the r_k are measures of whether the learner is achieving its goals.

For successful learning, the agent must obtain sufficient experience. In a finite-state problem, the learner must try out each possible action in each state repeatedly to be sure of finding the optimal policy.

The constraints on what experience the learner can easily obtain depend on the learning problem. For example, a solitaire player may set up the pieces in any position he chooses, and return to any previous position to study it, and he may try out many different sequences of moves from the same board position. A lion learning how to stalk and kill gazelles does not have this luxury: if the lion charges too early so that the gazelle escapes, then it must start again from the beginning, and find another gazelle and stalk it, until it faces a similar situation and it can try creeping just a little closer before starting to charge. The lion's experience is hard won.

2. Methods of Estimating Values and Action-Values

All methods of primitive learning rely upon estimating values and action values directly from experience. At all times, the agent will have a current policy: values and action values are estimated according to this policy. The first technique to discuss, therefore, is the various methods of forming estimates of expected returns. In this section of the chapter, I will develop a notation for describing a family of estimators of returns; this notation will be convenient for the concise description of learning methods in later parts of the chapter.

The simplest way to estimate expected returns is just to keep track of the sum of discounted rewards while following the policy. Let us define the *actual return* from time t as

$$\mathbf{r}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n} + \cdots$$

That is, the actual return $\mathbf{r}_t$ is the actual sum of discounted rewards obtained after time t. The optimal policy is that policy which, if followed in perpetuity,

will optimise the expected actual return from every state. For any given policy f, $V_f(x)$ is the expected value of the actual return that would be received after starting in state x and following policy f thereafter.

Because $\gamma < 1$, γ^n will approach zero as n becomes large, and because all rewards are assumed to be bounded, for each value of γ there will be some number of time-steps n after which the remaining part of the actual return will be negligible. Hence the agent may calculate an acceptably accurate value of the actual return from time t at time $t+n-1$. The *n-step truncated return* is

$$\mathbf{r}_t^{[n]} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1}$$

If the rewards are bounded, then the maximal difference between the truncated return and the actual return is bounded, and tends to zero as the number of steps before truncation is increased.

Clearly, one method of estimating the value function for a given policy f is to follow f and, for each state, to average its n-step truncated returns, for some sufficiently large n.

There are three disadvantages of this method of estimating the value function:

- The value of the truncated return is only available to the agent after a time delay of n steps. To calculate the actual return from each state visited, the agent must remember the last n states and the last n rewards.
- The truncated returns may have a high variance, so that many observations are necessary to obtain accurate estimates.
- To use the truncated return from time t to estimate the value of state x_t according to policy f, the agent must continue to follow policy f for at least n steps after time t.

These problems may be avoided by using a different class of estimators; some of these have been described by Sutton (1988), and he terms them 'temporal difference' methods. These methods rely upon the agent maintaining a current approximation to the value function. This approximate value function is denoted by U. The agent seeks to make U approximate the value function more closely, and U will change with time: U at time t is denoted by U_t.

The n-step truncated return $r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1}$ does not take into account the discounted rewards $\gamma^n r_{t+n} + \gamma^{n+1} r_{t+n+1} + \cdots$ that would be received if the agent continued following its policy f. But the sum $r_{t+n} + \gamma r_{t+n+1} + \cdots$ can be approximated by the agent as $U_{t+n}(x_{t+n})$. This can be used as a correction for the n-step truncated return. The *corrected* n-step truncated return for time t is

$$\mathbf{r}_t^{(n)} \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n U_{t+n}(x_{t+n})$$

If U were equal to V_f, then the corrected truncated returns would be unbiased estimators of V_f, since

$$\begin{aligned} V_f(x) &= \mathbf{E}\Big[R(x,f,0) + \gamma V_f(X(x,f,1)) \Big] \\ &= \mathbf{E}\Big[R(x,f,0) + \gamma R(x,f,1) + \gamma^2 V_f(X(x,f,2)) \Big] \\ &= \mathbf{E}\Big[R(x,f,0) + \gamma R(x,f,1) + \cdots \Big] \end{aligned}$$

The reason that corrected truncated returns are useful estimators is that the expected value of the corrected truncated return tends to be closer to V_f than U is. Let K be defined as the maximum absolute error of U, that is

$$K = \max_x | U(x) - V_f(x) |$$

Then

$$\max_x | \mathbf{E}[\mathbf{r}^{(n)}(x)] - V_f(x) | \leq \gamma^n K$$

This might be called the *error-reduction property* of corrected truncated returns. For $\mathbf{r}^{(1)}$, the proof is a special case of the local improvement theorem proved in the chapter on dynamic programming; for $\mathbf{r}^{(n)}$ the proof is essentially the same. Note that $\mathbf{E}[\mathbf{r}^{(n)}(x)]$ is not necessarily closer to $V_f(x)$ than $U(x)$ is for *all* x—but the maximum error of $\mathbf{E}[\mathbf{r}^{(n)}(x)]$ is less than the maximum error of U.

One is not restricted to using $\mathbf{r}^{(n)}$ for just a single value of n: it is possible to use weighted averages of $\mathbf{r}^{(n)}$ for different values of n. These weighted averages of corrected truncated returns will still have the error-reduction property in the following sense. If $\mathbf{r}^{(\mathbf{w})}$ is a weighted sum of corrected truncated returns

$$\mathbf{r}^{(\mathbf{w})} = \sum_i w_i \mathbf{r}^{(i)}$$

where the weights w_i sum to 1, then

$$\max_x |\mathbf{E}[\mathbf{r}^{(\mathbf{w})}] - V_f(x)| \leq \sum_i \gamma^i |w_i| K$$

Note that $\sum_i \gamma^i |w_i| \leq 1$ provided that all the w_i are between 0 and 1.

An important case is to use a weighted sum in which the weight of $\mathbf{r}^{(n)}$ is proportional to λ^n for some λ between 0 and 1: this weighted sum will be denoted by $\mathbf{r}^{\lambda}$. This estimator has been investigated by Sutton (1988); I will describe his work in my notation.

What Sutton terms the TD(λ) return from time t is

$$\mathbf{r}_t^{\lambda} = (1-\lambda)[\mathbf{r}_t^{(1)} + \lambda\mathbf{r}_t^{(2)} + \lambda^2\mathbf{r}_t^{(3)} + \cdots]$$

$$= r_t + \gamma\,(1-\lambda)\,U_t(\,x_{t+1}\,) +$$

$$\gamma\lambda\Big[\, r_{t+1} + \gamma\,(1-\lambda)\,U_{t+1}(\,x_{t+2}\,) +$$

$$\gamma\lambda\Big[\, r_{t+2} + \gamma\,(1-\lambda)\,U_{t+2}(\,x_{t+3}\,) +$$

$$\gamma\lambda\Big[\, r_{t+3} + \gamma\Big[\ \cdots$$

Because $\mathbf{r}^{\lambda}$ is a weighted average of corrected truncated returns, it has the error-reduction property. Note that U is time-indexed; in value estimation, U may change slightly at each time step. However, the changes in U will be small if the learning factor is small.

The expression above may be written recursively as

$$\mathbf{r}_t^{\lambda} = r_t + \gamma\,(1-\lambda)\,U_t(x_{t+1}) + \gamma\,\lambda\,\mathbf{r}_{t+1}^{\lambda}$$

The TD(0) return, with $\lambda = 0$ is just

$$\mathbf{r}_t^{0} = r_t + \gamma\,U_t(x_{t+1})$$

and if λ is set to 1, the expression for the TD(1) return is

$$\mathbf{r}_t^{1} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

which is just the actual return.

2.1. Choice of λ as a Trade-Off between Bias and Variance

All these different estimators of the expected return may be defined, but what use are they? In particular, how should one choose the values λ? If the agent follows the policy V_f for long periods of time, then should it not use the obvious method of truncated returns for some sufficiently large n? Although I

have not developed any rigorous argument, the following considerations should affect the choice of λ.

The choice of values of λ depends on a trade-off between bias and variance. If the values of U are close to those of V, then it is easy to show that the variance will be lowest for $\lambda = 0$, and highest for $\lambda = 1$. However, low variance is obtained by truncating the sequence of discounted returns and adding a value of U as a correction. If these corrections are wrong, then the estimates will be biased. The optimal choice of λ, therefore, depends upon how close the current values of U are to V_f. It seems plausible that, to estimate V_f by following f and observing rewards, the fastest method is to use $\lambda = 1$ to start with, and then reduce λ to zero as U becomes more accurate.

I have not done any quantitative analysis of this problem, but Sutton (1988) reports some computational experiments for a related problem in which λ was kept constant throughout estimation, and he found that the most rapid convergence during the time course of his experiment was obtained with intermediate values of λ.

2.2. Implementation Using Prediction Differences

Sutton (1988) defines the *prediction difference* at time t as

$$e_t = r_t + \gamma U_t(x_{t+1}) - U_t(x_t)$$

The motivation for the term 'prediction difference' is that at time t the agent might predict the return from time t to be $U_t(x_t)$; at time t+1 the agent has more information to go on in predicting the return from time t, as it has observed r_t and x_{t+1} . It may, therefore, make a new prediction of the return from time t as $r_t + \gamma U_t(x_{t+1})$. The *prediction difference* is the difference between these two predictions. If U is equal to V_f for all states, then the

expected value of all prediction differences is zero. (The individual prediction differences actually observed will not be zero if the process is random so that state transitions or rewards may vary.)

Now, as Sutton suggests, the difference between the TD(λ) return and the estimated value may be rewritten as

$$\mathbf{r}_t^\lambda - U_t(x_t) = e_t + \gamma\lambda e_{t+1} + \gamma^2\lambda^2 e_{t+2} + \gamma^3\lambda^3 e_{t+3} + \cdots$$

$$+ \sum_{n=1}^{n=\infty} (\gamma\lambda)^n [U_{t+n}(x_{t+n}) - U_{t+n-1}(x_{t+n})]$$

If the learning factor is small, so that U is adjusted slowly, then the second summation on the right hand side above will be small.

The usefulness of this way of calculating $\mathbf{r}_t^\lambda - U_t(x_t)$ in terms of prediction differences is that the agent can calculate the prediction difference for time k at time k+1—a delay of only one time-step. Furthermore, if U is close to V, the average value of the prediction differences will be small. Now, the natural way to use $\mathbf{r}^\lambda$ to improve the estimated value function U is to use an update rule

$$\begin{aligned} U_{t+1}(x_t) &= (1-\alpha)U_t(x_t) + \alpha\mathbf{r}_t^\lambda \\ &= U_t(x_t) + \alpha(\mathbf{r}_t^\lambda - U_t(x_t)) \\ &= U_t(x_t) + \alpha\left[e_t + \gamma\lambda e_{t+1} + (\gamma\lambda)^2 e_{t+2} + \cdots\right] \end{aligned}$$

Thus the update rule can be implemented by, at each time step, adding appropriate fractions of the current prediction difference to previously visited states.

One way of doing this is to maintain an 'activity trace' for each state visited (Barto et al (1983) describe this as an 'eligibility trace'—a trace of how 'eligible' the estimated value of a state is to be modified). When a state is

visited, the activity becomes high; the activity then declines gradually afterwards. The amount by which the estimated value of the state is adjusted is α times the current activity of the state times the current prediction difference. Let the 'activity' of a state x at time t be $C(x,t)$. The levels of activity of all states may be updated at each time step by the following rules:

$$C(x, t) = 0 \quad \text{if } x \text{ has never been visited.}$$

$$C(x, t) = \gamma\lambda C(x, t-1) \quad \text{if } x_t \neq x$$

$$C(x, t) = 1 + \gamma\lambda C(x, t-1) \quad \text{if } x_t = x$$

That is, 1 is added to the activity of the current state, and the activities of all states decay exponentially with a time constant of $\gamma\lambda$. This form of the algorithm may be natural to use in connectionist implementations, where the memory of a recent visit to a state might take the form of 'traces of activity' in a distributed representation of the state. This method is used in the learning algorithm in Barto, Sutton, and Anderson (1983).

The value updating rule suggested by Sutton is

$$U_{t+1}(x) = U_t(x) + \alpha\, C(x, t+1)\, e_t$$

This rule may be most suitable for connectionist implementations of incremental updating of value functions, where it may be natural to update all states at each time step, but for implementations on sequential computers, it is inefficient to update all states at each time step, and a more efficient way is to keep track of the last n states visited for sufficiently large n. The size of n required will depend on the value of λ used: the smaller the value of λ, the smaller the value of n needed. If $\lambda = 0$, then $n = 1$.

Sutton (1988) describes the use of these temporal difference estimators for estimating value functions in Markov processes with rewards. He considers a

more general formulation of the problem, in which the states of the Markov process are described by linearly independent vectors, and the value function is represented by a vector of adjustable weights. The value of any state is the inner product of the state vector with the weight vector.

Sutton also suggests that these methods can be used in procedural learning: the learning method of Barto, Sutton, and Anderson (1983) uses a connectionist-style implementation of this method of estimating expected returns. Sutton (1988) suggests widespread application of the method in behavioural learning. Sutton and Barto (1988) suggest a model of classical conditioning in terms of the animal learning to estimate the expected total discounted unconditioned stimulus that it will receive. The probability of a classical response was assumed to be proportional to this estimate. In this model, learning is continuous during experience, rather than occurring at the end of each 'trial'; this is an advantage, since the animal might not divide its flow of experience into 'trials' in the same way that the experimenter would.

2.3. Estimating Values and Action Values in a Markov Decision Process

So far, a range of methods have been introduced for estimating expected returns in a Markov process with rewards. In a decision process, in which different actions are possible at each state, there is a complication: it is only meaningful to estimate a return relative to some policy.

In the learning methods to be described, the agent seeks to estimate returns relative to an internally represented policy—its *estimation policy*. In some methods of learning the estimation policy is stochastic, in others, it is stationary. In some methods, the agent always follows its estimation policy; in other methods, the agent may deviate from its estimation policy.

If the agent always follows its estimation policy, then the estimation problem is the same as the previous problem for Markov processes with rewards. But what if the agent does *not* follow its estimation policy f but instead follows another policy g, or else does not follow a consistent policy at all? Can the agent still estimate V_f and Q_f then?

Yes it can, by using different values of λ at different time steps, with the value of λ depending on whether the current action is a policy action or not. One may define

$$\mathbf{r}_t^\Lambda = r_t + \gamma\,(1-\lambda_{t+1})\,U_t(x_{t+1}) + \gamma\,\lambda_{t+1}\,\mathbf{r}_{t+1}^\Lambda$$

$$= (1-\lambda_{t+1})\mathbf{r}_t^{(1)} + \lambda_{t+1}(1-\lambda_{t+2})\mathbf{r}_t^{(2)} + \lambda_{t+1}\lambda_{t+2}(1-\lambda_{t+3})\mathbf{r}_t^{(3)} + \cdots$$

where Λ is the sequence of values λ_1 , λ_2 , $\cdots$, and where $0 \le \lambda_t \le 1$ for all t. Λ need not be defined in advance: λ_t may depend on the state and action at time t. Since $\mathbf{r}^\Lambda$ is a weighted average of corrected truncated returns, it has the error reduction property.

The point of this definition is that by a suitable choice of λ_t the agent may estimate the returns under one policy while behaving quite differently. Let the agent divide the steps of its experience into two classes: 'policy steps' and 'experimental steps'. If the agent's estimation policy is stationary—that is, there is a unique policy action for each state—then all steps in which the agent performs the policy action are policy steps, and all other steps are experimental steps.

If the estimation policy is stochastic, so that the policy is to choose actions by sampling from a probability distribution at each state, then the question of whether a step is policy or experimental is not so clear cut. One method that the agent can use is to accept or reject steps in such a way that, for each state, the relative rates of acceptance of actions are kept approximately the same as

the action probabilities according to the policy.

Suppose the agent immediately classifies each step of its experience either as policy or as experiment. It must ensure that its estimated returns are not contaminated by the effects of experimental actions. If the agent performs an experimental action at time t, then it cannot use any of its experience after time t in an estimate of the return from an action taken before time t. To do this, the agent may define Λ by

$$\lambda_t = \begin{cases} \lambda^* & \text{if } a_t \text{ is a policy action} \\ 0 & \text{if } a_t \text{ is an experimental action} \end{cases}$$

where λ^* is a chosen value between 0 and 1. The essential point is that if step t is experimental, then λ_t must be zero. If Λ is defined in this way, then $\mathbf{r}_t^\Lambda$ will be an estimate of $Q_t(x_t, a_t)$, uncontaminated by the effects of subsequent experimental actions.

To see this, consider the recursive definition of $\mathbf{r}_t^\Lambda$ in terms of $\mathbf{r}_{t+1}^\Lambda$:

$$\mathbf{r}_t^\Lambda = r_t + \gamma (1-\lambda_{t+1}) U_t(x_{t+1}) + \gamma \lambda_{t+1} \mathbf{r}_{t+1}^\Lambda$$

If step t+1 is experimental then $\lambda_{t+1} = 0$, so that the term on the right hand side containing the subsequent return $\mathbf{r}_t^\Lambda$ is multiplied by 0. $\mathbf{r}_t^\Lambda$ is, therefore, an estimate of $Q_f(x_t, a_t)$, because after step t, it is constructed from $U_{t+1}(x_{t+1})$ and from any further policy steps from t+1 onwards. If some step $t+n$ is an experiment, $\lambda_{t+n} = 0$, and no further time steps contribute to the estimate. $\mathbf{r}^\Lambda$ has the error reduction property, therefore.

Note that $\mathbf{r}_t^\Lambda$ is an estimate of the *action-value* of x_t , a_t. The next state x_{t+1} is always taken into account in forming the estimate. But if a_t is experimental, then it is not possible to use $\mathbf{r}_t^\Lambda$ to estimate $V_f(x_t)$. Let us therefore define $\mathbf{u}_t^\Lambda$ as

$$\mathbf{u}_t^{\Lambda} = (1-\lambda_t)U_t(x_t) + \lambda_t \mathbf{r}_t^{\Lambda}$$

The agent may estimate $V_f(x_t)$ only if the action at time t— a_t — is a policy action: the agent cannot use the rewards following a non-policy step to estimate the value of the state according to the policy.

The above method of defining Λ is a special case of the more general methods used for variance reduction in Monte-Carlo estimation in Markov processes (see Rubinstein 1981, chapter 5).

3. Learning Action-Values

The problem of finding the optimal value function for a decision problem is more complex than that of merely estimating values and returns: the optimal policy is initially unknown, so that initially it is not possible to estimate the optimal value function directly. Instead, learning is a process of improving a policy and value function together.

One method of representing a policy and value function, as described in chapter 5, is to store action values $Q(x,a)$ for each state x and action a. The values of Q at time t are denoted by Q_t. From its values of Q, the agent may estimate the value of a state x as

$$U_t^Q(x) = \max_a \{ Q_t(x,a) \}$$

The superscript Q in U^Q is to indicate that U^Q is calculated from Q. Q implicitly defines a current policy f^Q which is

$$f_t^Q(x) = a \text{ such that } Q_t(x,a) = U_t^Q(x)$$

That is, the current policy is always to choose actions with maximal estimated action value.

How might the agent improve Q through its experience? The simplest method is *one-step Q-learning*, in which the values of Q are adjusted according to

$$Q_{t+1}(x_t, a_t) = (1-\alpha)Q_t(x_t, a_t) + \alpha(r_t + \gamma U_t^Q(x_{t+1}))$$

$$= (1-\alpha)Q_t(x_t, a_t) + \alpha \mathbf{r}_t^{(1)}$$

where α is a 'learning factor', a small positive number. The values of Q for all other combinations of x and a are left unchanged.

Note that the actions that the agent should take are not specified. In fact, the agent is free to take whatever actions it likes, but for it to be sure of finding the optimal action value function eventually, it must try out each action in each state many times.

Does this learning method work? It does indeed, because it is a form of value iteration, one of the conventional dynamic programming algorithms described in chapter 4. As is explained in appendix 1, one-step Q-learning can be viewed as incremental, Monte-Carlo value iteration: Q_{t+1} is estimated from U_t^Q, and U_t^Q is obtained by maximising Q_t at each state.

Appendix 1 presents a proof that this learning method does work for finite Markov decision processes. The proof also shows that the learning method will converge rapidly to the optimal action-value function. Although this is a very simple idea, it has not, as far as I know, been suggested previously. However, it must be said that finite Markov decision processes and stochastic dynamic programming have been extensively studied for use in several different fields for over thirty years, and it is unlikely that nobody has considered this Monte-Carlo method before.

3.1. Learning Action Values using General Estimates of Returns

In fact, there is a family of methods of Q-learning, which use different estimators of expected returns. The principle of all the methods is that the values of Q are updated using estimates $\mathbf{r}_t^\Lambda$ of the action values. As before, λ_t may be defined by

$$\lambda_t = \begin{cases} \lambda^* & \text{if } a_t \text{ is a policy action} \\ 0 & \text{otherwise} \end{cases}$$

With λ_t defined in this way, returns are estimated only over sequences of policy actions. A 'softer' definition of λ_t, which is used in the second demonstration program in chapter 11, is to make λ_t depend on the difference between the estimated action value of the action performed and the estimated value of the state. If $Q_t(x_t, a_t)$ is much less than $U_t^Q(x_t)$ then λ_t should be small, whereas if $Q_t(x_t, a_t)$ is nearly as large as $U_t^Q(x_t)$ then λ_t should be nearly as large as λ^*. One method of achieving this is to calculate λ_t thus:

$$\lambda_t = \exp(-\eta(U_t^Q(x_t) - Q_t(x_t, a_t))\,\lambda^*$$

where the parameter η is non-negative real number. If η is zero, then $\lambda_t = \lambda^*$ for all t, whereas if η is large then λ_t will be small if $Q_t(x_t, a_t)$ is even slightly less than $U_t^Q(x_t)$.

Note that the value of the estimated return $\mathbf{r}_t^\Lambda$ will only become available at some later time. As a result, Q cannot be updated immediately according to $\mathbf{r}_t^\Lambda$ —the update must be made later on, at time $t+T$, say:

$$Q_{t+T+1}(x_t, a_t) = (1-\alpha)Q_{t+T}(x_t, a_t) + \alpha\, \mathbf{r}_t^\Lambda$$

or else the updates may be performed incrementally using the method of prediction differences described in section 2.3 above. There is a potential problem here: Q is being incrementally updated at each time step, but it is also being used in calculating the estimates of returns $\mathbf{r}^\Lambda$ on which the updates are based.

That is, changes in Q may affect $\mathbf{r}^{\Lambda}$, which then affect the changes in Q, and so on: however, these effects are proportional to α^2, and so will be negligible for small α.

This is a family of learning methods that appear plausible—do they work? Unfortunately, I have not been able to show that they will always work, for the following reason. The difficulty is that if the values of Q differ from the optimal values Q^*, then the implicit policy f^Q may differ from the optimal policy f^*. The problem is that if Q is perturbed away from the optimal action values Q^* by only a small amount, so that the implicit policy f^Q differs from the optimal policy, the value function for f^Q may differ from the optimal value function V by a larger amount. That is, a small perturbation of Q away from the optimum may under some circumstances lead to instability. These instabilities need not necessarily occur, but I have not been able to find useful conditions under which they can be guaranteed not to occur. In computer implementations to be described later, the instabilities were not significant.

4. Learning a Policy and a Value Function

In previous work on what I have termed primitive learning methods, the policy has been represented by action-weights, or action probabilities, rather than by action-values. The values of *states* were represented explicitly, but *action-values* were not. Methods of this type have been described by Michie (1967), Widrow et al (1972), Mendel and McLaren (1972), Witten (1977), Barto, Sutton, and Anderson (1983) and Sutton (1984), Wheeler and Narendra (1986), and Anderson (1987), and also by Liepins et al (1989), and Hampson (1983).

These previous methods divide naturally into two types: learning a policy alone, and learning a policy together with a value function.

4.1. Learning a Policy Alone

The agent maintains and adjusts a representation of a policy during learning. The policy is stochastic. Different actions may be performed on different visits to the same state, so that the agent has the opportunity to compare the effects of different actions. After an action has been performed, the agent uses its subsequent experience to form an estimate of the expected return from that action that results from following the current policy. The agent then gradually increases the probabilities of those actions that lead to high estimated returns, and it reduces the probabilities of actions that lead to lower expected returns. Provided that the agent's estimates of expected returns are unbiased, this learning process can be viewed as a form of incremental, Monte-Carlo policy improvement.

Wheeler and Narendra (1986) propose an interesting method for the case of a finite Markov decision process in which the aim is to maximise long-run *average* reward, rather than expected discounted reward. Their estimate of the expected return was obtained by a recurrence method. If the state at time t is x, and action a is performed, and the next time at which the process returns to state x is $t+T$, they use

$$\frac{r_t + r_{t+1} + \cdots + r_{t+T-1}}{T}$$

as an unbiased estimate of the expected average return that would result from a policy of performing a in x, and of following the current policy elsewhere. Of course, the policy is changing slowly at all states during the estimation of expected average returns, but Wheeler and Narendra give a proof that the system can be made to converge to an optimal policy with as high a probability as desired.

They represent the policy directly as a set of action-probabilities at each state. They assume that the Markov processes for all possible policies are recurrent, and that the estimated returns lie between 0 and 1. They update the state probabilities by the L_{R-I} rule (Narendra and Thathachar (1974) or Lakshmivarahan (1981) are surveys of stochastic learning automata).

Widrow et al (1972) considered a sequential task (playing Blackjack) that always finished after a finite number of actions; when the task finished, the agent received a reward depending on the outcome. The return that was to be maximised by an optimal policy was the expected terminal reward. The terminal reward actually obtained was used as the estimate of expected terminal reward for each action taken during each bidding sequence. The action probabilities were adjusted so that actions taken became more probable if the reward was high, and the probabilities became lower if the reward was low.

Barto, Sutton, and Anderson (1983) implemented a method of this type with discounted returns for the pole-balancing problem as a method to compare with their adaptive heuristic critic algorithm. The results were disappointing. However, their formulation of the pole-balancing task had a larger number of states than either blackjack bidding or the demonstration problems used by Wheeler and Narendra—and it is possible that for some choice of very small learning parameters, the method would work after a large number of training runs. In addition, the formulation of the pole-balancing problem used was not actually a Markov process, since the cart and pole system moves deterministically in response to actions, and the state-space was partitioned into quite coarse regions.

4.2. Learning a Policy with a Value Function

In the learning methods described by Witten (1977), Barto, Sutton, and Anderson (1983) and Sutton (1984), and Anderson (1987), the agent acquires both a policy and a value function. The estimated value function is used to provide TD(λ) returns that are of lower variance than observed returns, and which are available within a shorter time. A possible drawback, however, is that these estimates may be biased if the estimated value function happens to be in error.

In Witten's method there are two concurrent adaptive processes: improvement of the policy and estimation of the value function for the current policy. At each time-step, the value function is adjusted by

$$U_{t+1}(x_t) = (1-\alpha)U_t(x_t) + \alpha(r_t + \gamma U_t(x_{t+1}))$$

Witten proposes that the policy at each state should be adjusted by an unspecified learning automaton, using $r_t + \gamma U_t(x_{t+1})$ as the reward. Witten recommends that the learning rate of the value function should be much higher than that for the policy, so that on the time scale of the policy adjustment, the mean value of U at each state can be assumed to be equal to V. He then proves that, under these conditions, there is a unique optimal value function and class of optimal policies, but he does not point out any connection with dynamic programming.

There are two differences between the adaptive heuristic critic algorithm and Witten's method: the AHC algorithm uses TD(λ) returns, rather than just TD(0) returns, and the policy is adjusted according to a *reinforcement comparison* method (Sutton (1984)). That is, the quantity that is used to adjust the policy is the *difference* between the TD(λ) estimated return from the state x and the estimated value of x. Sutton (1984) showed in a number of simulation experiments that learning automata that used the difference between the reinforcement from the environment and an estimate of the expected reinforcement

under the current policy appeared to converge considerably faster than conventional learning automata.

I have not been able to prove that either of these policy learning methods will necessarily work, nor have I been able to construct any problem for which, using stochastic approximation rather than learning automata, the methods would fail. I think it very probable that it is possible to give conditions under which these methods could be guaranteed to work, but the proof techniques used for action-value estimation cannot be applied in this case, because the action-values themselves are not represented.

The reason for the difficulty is that there are two concurrent adaptive processes—value estimation and policy improvement—and there is a possibility that these may interact during learning to prevent convergence.

4.3. Representing a Policy by a Single Action at Each State

A still simpler way to represent a policy is to store a single action for each state: the policy is then stored as a function from states to actions. In this case, the agent does not need to determine which action has the highest strength in the current state—it simply uses its stored policy to compute an appropriate action. Note that this representation of the policy may require very much less information than either the action strength or the action-value representations. This is a genuinely simpler learning method: if many actions are possible in each state, this method of representing the policy could have considerable advantages. As far as I know, the following learning method has not been suggested before.

If the possible actions at each state themselves form a vector space, the choice of action may be improved by a gradient method, in the following way. Let the current policy be f. Suppose the agent is in some state x, and it

performs the action a, different from the policy action $f(x)$. One method of adjusting the policy is according to the rule

$$f_{t+T+1}(x_t) = f_{t+T}(x_t) + \beta(\mathbf{r}_t^{\Lambda} - U_{t+T}(x_t))(a_t - f_{t+T}(x_t))$$

where β is a learning factor, a small number greater than zero. The effect of this adjustment rule is that if $\mathbf{r}_t^{\Lambda} - U_{t+T}(x_t)$ is positive, so that the estimated return from performing a_t was greater than the estimated value of x_t—in other words, if a_t was 'unexpectedly good'—then the policy action for x_t is adjusted towards a_t.

This learning rule may be extended to cover some stochastic policies. If the stochastic policy is to perform an action

$$f(x) + \zeta$$

where ζ is a (vector) random variable of zero mean, then the same adjustment rule for f may be used.

A modification of this adjustment rule is to have two learning factors β^+ and β^-. If $\mathbf{r}_t^{\Lambda} > U_t(x_t)$, so that a_t is better than expected, then β^+ is used in the adjustment rule; if the action is worse than expected, the learning factor β^- is used. Both β^+ and β^- are greater than 0, and

$$\beta^+ \gg \beta^-$$

The learning rule becomes

$$f_{t+T+1}(x) = \begin{cases} f_{t+T}(x_t) + \beta^+(\mathbf{r}_t^{\Lambda} - U_{t+T}(x_t))(\, a - f_{t+T}(x_t)\,) & \text{if } \mathbf{r}_t^{\Lambda} > U_{t+T}(x_t) \\ f_{t+T}(x_t) + \beta^-(\mathbf{r}_t^{\Lambda} - U_{t+T}(x_t))(\, a - f_{t+T}(x_t)\,) & \text{if } \mathbf{r}_t^{\Lambda} \leq U_{t+T}(x_t) \end{cases}$$

The motivation for modifying the adjustment rule in this way is that if f and U are nearly optimal, then experimental, non-policy actions will usually lead to estimated returns that are lower than $U(x)$. If β^- is large, then each experiment

will cause $f(x)$ to change: experiments will, therefore, cause random perturbations of $f(x)$ about the optimum. If β^- is made small in comparison to β^+ then the random perturbations caused by sub-optimal experiments will be smaller and more quickly corrected. This technique of asymmetric learning factors has been widely used for stochastic learning automata (Barto and Anandan 1985, and the review by Lakshmivarahan 1981). Widrow (1972) uses asymmetric learning factors for analogous reasons, and reports a greatly increased speed of convergence.

The estimated value function, U may be modified using the estimator for values $\mathbf{u}^\Lambda$:

$$U_{t+T+1}(x_t) = (1-\alpha)U_{t+T}(x_t) + \alpha\mathbf{u}_t^\Lambda$$

$$= (1-\alpha)U_{t+T}(x_t) + \alpha[(1-\lambda_t)U_{t+T}(x_t) + \lambda_t\mathbf{r}_t^\Lambda]$$

Once again, Λ may be defined so that $\mathbf{u}^\Lambda$ is an estimated return according to f.

Note that the processes of adjusting U and f are quite separate, and different estimators for the returns may be used in each.

Should this method work? One limitation to note immediately is that the policy is updated by a gradient method at each state. If, therefore, at any time there is more than one maximum in the action-value function at a state, then it is possible for the policy action at that state to converge to a sub-optimal local maximum.

Apart from this difficulty, the 'learning method' may be subject to the same instabilities as the action-strength methods of the previous section: I do not know under what conditions it can be *guaranteed* to converge to the optimal value function and policy. Later on, however, I will describe an implementation of this learning method, and, for that example, it appears to work rather well.

5. Learning a Value Function Alone

Finally, perhaps the simplest learning method of all is to learn a value functions alone, under desirability-gradient control of action. Once again, a possible learning rule is

$$U_{t+T+1}(x_t) = (1-\alpha)U_{t+T}(x_t) + \alpha \mathbf{u}_t^{\Lambda}$$

As before, this may be implemented in either the sequential or the connectionist style. For this method to be valid, $\mathbf{u}_t^{\Lambda}$ must be constructed according to the desirability gradient control policy. There is no adjustment to the policy because the agent does not have one: U itself is used in the control of action. Whether this learning process will converge will, I believe, depend on the particular desirability gradient control method used.

The reader may wonder whether learning a value function in this way is really a form of instrumental learning. For learning of this simple type, the distinction between instrumental and classical conditioning begins to break down, but a distinction can nevertheless be maintained in principle, as follows. Some simple organisms may behave according to the same control policy all the time: others may sometimes behave according to a particular desirability-gradient method, and sometimes not. For the learning to be instrumental, it must only happen while the agent is following its desirability-gradient policy. If there are temporal correlations of states and rewards that occur during times that the agent is not using its control method, then these cannot be attributed to the method of control, and so should not cause changes in the value function in instrumental learning. Hence if the agent learns from temporal correlations that occur when the agent is not following its control policy, then it is classical rather than instrumental conditioning.

This is a simple method of learning that could be used by simple organisms, in the following way. Suppose that some fly had the ability to follow certain odour gradients in the air, using a desirability-gradient method. Some odours might be innately attractive, and generally useful for the fly to follow. However, for any particular habitat, there might be some other odours that it would be useful for the fly to learn about: these secondary odours would provide indications of the presence of the innately attractive odours. This type of learning, therefore, might more conventionally be described as a form of classical conditioning of odours.

6. Restricted Experience and Meta-Stable Policies

In chapter 4 on dynamic programming, one of the conclusions was that there is only one optimal value function V, and the optimisation process will always find V. In primitive learning one of the conditions for success is that the agent should repeatedly try all possible actions in each possible state. But at intermediate stages of learning, the agent's policy may lead it to visit only a part of the state-space: this can result in a *meta-stable* policy.

A meta-stable policy is one that is sub-optimal, but which, if followed, prevents the agent from gaining the experience necessary to improve it. Meta-stable policies occur frequently in everyday life: sitting in a corner at parties, for example, is a strategy that may prevent people from learning to enjoy them.

All the previously published primitive learning methods I have cited have one aspect in common:

- *The agent always behaves according to its current estimation policy.*

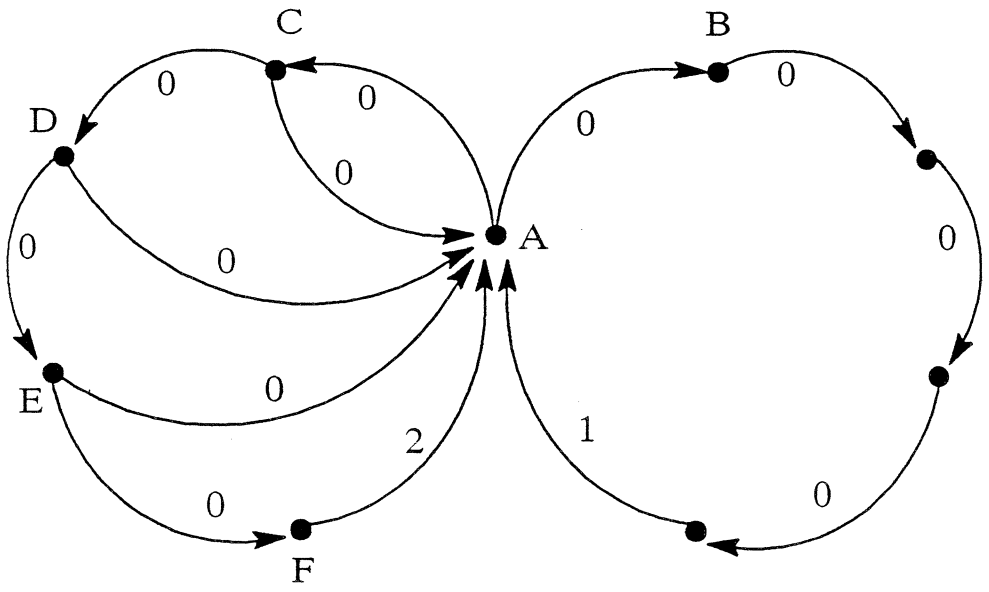
Each action the agent takes is used to adjust the estimated value function, and the policy is stochastic so that the agent necessarily experiments with different actions in following the policy. This approach of combining experiment with

value estimation might be called the 'random policy trick'—a 'trick' because it simplifies the learning algorithm, in that it is not necessary to specify any further what the agent should do. This 'trick', however, may severely limit the performance of the learning algorithm.

The random policy trick restricts the amount of experimentation that the agent can do during the later stages of learning, and it governs the nature of the experimentation at all stages. As the estimation policy approaches the optimal policy, the agent will perform fewer and fewer experimental actions, and further improvement of the policy will become slow. Worse still, at intermediate stages of learning the policy may be sub-optimal and almost deterministic in some parts of the state space. In such regions of the state-space, there will be a low level of experiment, so that changes in the policy in these regions will be slow. To see how this may happen, consider the following simple example problem, illustrated in the diagram overleaf.

The dots labelled with letters represent states, which are named A to F. The arrows between the dots represent possible actions. For example, the arrows from A to C and from A to B signify that, in state A, the agent has the choice of two actions: to move to state C or to move to state B. Whenever the agent reaches state B it receives a reward of 1, and whenever it reaches state F it receives a reward of 2. The agent wishes to find a policy that optimises expected return according to a discount factor of 0.9; otherwise, the agent receives no rewards.

In this decision process, there are two loops of states that the agent can traverse. The loop on the right, passing from state A to state B and back again, gives low level of return. The loop on the left, from A to C to D to E to F to A, yields a high reward at F. This path yields a high level of return provided that the agent goes all the way round it. The optimal policy is to follow the



The optimal policy is to follow the loop ACDEFACDEFA...

But if the initial policy at A, C, D, and E is to choose either action with equal probability, then the probability of AB will initially increase, since F is seldom reached.

If the agent must follow its estimation policy, then acquiring the optimal policy may take a very long time.

path ACDEF repeatedly. However, at states C, D, and E the agent has a choice of either continuing along the optimal path, or of returning to A with a small reward.

Consider the course of learning according to the random policy trick if the initial policy is, at each state, to choose all possible actions with equal probability. On this initial policy, the action value of going from A to B is higher than the action value of going from A to C, because if the agent goes from A to C, it is likely to return to A from either C, D, or E with no reward, whereas if it goes from A to B, then it will always receive the sure but sub-optimal return. Initially, therefore, any primitive learning system will adjust its estimation policy at A so that going to B becomes more probable than going to C. As the agent accumulates more experience, it will find, on the occasions that it reaches E, that it is much better to go to F than to go to A, and it will adjust its value estimate for E upwards. When the value estimate for E is high, the agent will find, when it visits D, that it obtains a better return from going to E than from going to A, and it will adjust its policy accordingly. In this way, the estimation policy will be adjusted towards the optimal policy and the values will be adjusted towards the optimal values starting with E, then D, then C, and finally A.

However, during a substantial initial part of this learning process, the action value at A of going to B will be higher than the action value of going to C, and the estimation policy will be adjusted to favour B over C. If the agent is following its estimation policy, it will visit C less and less often. Provided that the probability of going from A to C does not decrease too quickly, the agent will eventually obtain enough experience of the path from C to recognise its value, and, once the estimated value of C has risen above the estimated value of B, the probability of going from A to C will start to rise. Ultimately, the

estimation policy will converge to the optimal policy.

The point of this example is, however, that learning may take an indefinitely long time if the agent always follows its estimation policy because the the rate of visiting C may become very small. It is possible to make the learning indefinitely slow by increasing the number of states in the chain between C and the state F in which the agent receives the reward.

What this example demonstrates is that the speed of learning depends critically on the agent's pattern of experience during the course of learning. To return briefly to the example, consider how much more quickly the agent might learn if it were repeatedly placed in random states, so that it visited D, E, and F more frequently, and would as a result have the opportunity to improve its policy and value function at D and E.

This small, artificial example is not a contrived or exceptional case: it is a simple example of a general difficulty. In learning, the agent needs to improve its policy, to estimate expected returns, and, depending on the learning method, to construct a value function or an action-value function. In primitive learning, an agent can only improve its current policy by trying out alternative actions, and altering its policy if it finds actions that yield higher returns. Yet, in most of the learning methods, this requirement to experiment is at odds with the requirement to follow the policy. The policy may become almost deterministic while it is still sub-optimal, and further learning is then very slow.

Not only must an agent try out a sufficient variety of actions in the states that it visits: it must visit all the possible states. In justifying their learning methods, Wheeler and Narendra (1986) and Witten (1977) both need to assume that the agent will repeatedly visit all states while following *any* policy: but this assumption is restrictive, and in many problems it is simply not true. If, under some policies, the agent does not visit certain areas of the state space, then the

agent cannot improve its policy in those areas. It is then possible for the agent's policy to be optimal for the decision problem restricted to the regions of state space that it frequently visits; in this case, policy improvements could only be made in the areas that the agent does not visit. The learning system can settle into a metastable state, at a sub-optimal policy.

If the learner must always follow its estimation policy, then this problem of meta-stability will be particularly acute, because the learner's behaviour is so restricted. If the learner uses a method that allows it to make experiments, then the learner can *potentially* get the experience that it needs to improve its policy, but it need not necessarily do so.

Another, perhaps clearer, example of meta-stability might be termed the 'secret tunnel' problem in route finding. When I drive to work, I travel South across London and down to Surrey. I know the alternative routes, and in South London and North Surrey, I can follow an optimal policy. My choice of route is to some extent stochastic, but the areas I may visit during the journey are quite limited, being restricted to a narrow ellipse surrounding the optimal route. I have found this route through experience: when I first started making the journey, I tried alternative routes over a wider area, but now I have narrowed it down. There are, therefore, vast areas of England that I never visit during my journeys to work. It is possible that, if I were to travel a few miles North, I might, if I turned down some unpromising side-street, find the entrance to a secret tunnel beneath London, that would take me directly, unimpeded by traffic, to work. If such a tunnel existed, my present policy would be sub-optimal and meta-stable: if I continued to follow it, I should never find the secret tunnel. To guarantee to find the optimal route, I would have to make experiments, so that eventually, over a long period of time, I would explore every byway and eventually find the tunnel.

If such a tunnel existed, my current policy would be meta-stable in the sense that I would never find the tunnel during my current explorations near the route I now take: however, if I ever found the tunnel (or if I was led to it), I would change my current policy. Note that the optimal policy may be quite different from the meta-stable policy *even on those parts of the state space that are visited while following the meta-stable policy*: for example, if the secret tunnel existed, it would be better for me to turn back and head for the secret tunnel from up to half of my current route to work.

Meta-stability is a general phenomenon in learning by animals, machines, and people. The difficulty may be overcome to some extent by allowing the agent to perform actions inconsistent with its current policy. This allows the agent to reach parts of the state space that it might not otherwise experience, and to perform more experimental actions, and so be able to adapt its policy more quickly. However, there is no general method for obtaining suitable experience in an efficient way. As I will argue in the next chapter, the experimental strategy is an important type of prior knowledge for a learning agent.

7. Summary and Discussion

In this chapter, I have discussed a number of different plausible learning methods for Markov decision processes; but only for one of them—one-step Q-learning—have I been able to give a proof (in appendix 1) that it will converge to the optimal value function and policy. I have not been able to find proofs of convergence for the other learning methods: I believe that the methods will generally work, but I have not been able to find conditions on the behavioural policy during learning that would ensure that the methods are stable. The problem in proving that a learning method will converge is that, for the proof to be useful, the behaviour of the agent must be allowed to vary

almost arbitrarily, and some patterns of behaviour might cause instabilities that would prevent convergence.

Finally, the problem of meta-stable policies affects all of these learning methods. Unless the agent tries all actions in all states, no primitive learning method can be guaranteed to converge to an optimal solution. No local experimental strategy can succeed in eliminating this problem in general. One of the roles of prior knowledge and of advice is to induce the agent to try out useful actions, and to visit states of high value.

Chapter 8
Possible Forms of Innate Knowledge

It is notorious that it is difficult to learn something unless there is a sense in which one almost knows it already. Empirical observations need to be combined with prior knowledge, and the practical usefulness of any learning methods will be in proportion to how much use they can make of innate knowledge, and to how easily innate knowledge can be provided.

In the study of animal learning, one of the most important questions is that of what innate knowledge animals have, of what form it is encoded in, and of how it affects learning. In the introduction I argued that once members of a species rely on acquiring some valuable skill by learning, there will be selective advantage in that skill becoming innate. That is, animals that have some innate characteristics that enable them to learn the skill faster or more reliably will have an advantage over those individuals that do not learn the skill as fast or as surely. What types of innate characteristics, or 'knowledge', might individuals be provided with that would enable them to learn a behavioural strategy faster? Any candidate theory of animal learning should provide a variety of ways in which innate knowledge could be encoded and used.

1. Types of Innate Knowledge in Incremental Dynamic Programming

There are, perhaps, six types of innate 'knowledge' that may affect the speed of learning by incremental dynamic programming. They are

- physical capacities

- subjective reward systems
- methods of representation and approximation of functions
- initial policies, value functions, and models
- bounds and constraints on policies, value functions, and models
- tendencies to experiment

I will consider these types of innate characteristics in turn. 'Innate knowledge' is, perhaps, an inapposite term: 'innate characteristic that affects learning' would be better, but is longer.

1.1. Physical Capacities

Appropriate physical capacities may help learning in the same way that good tools can help carpentry. For example, if a bird has an appropriately shaped bill for opening a certain type of seed, then the precision with which the necessary actions must be performed to open a seed might be much less than would be necessary for a bird with an unsuitably shaped bill. Both birds might, if they were both skilled, be able to open the same seeds in the same time and with the expenditure of the same amount of energy: however, the bird with the well adapted beak might be able to learn the skill more quickly because the necessary policy might be simpler and require less precision to represent.

In sum, a physical adaptation may give an animal an advantage in many different ways—and one of these ways is in making a valuable strategy easier to learn. To call this a form of innate 'knowledge' is to stretch the term 'knowledge' a little far, but the effect of a more suitable bill might be the same as that of an innate cognitive ability.

1.2. Subjective Reward Systems

So far, I have usually implied that the immediate rewards that an animal receives for its actions are the primary reinforcers that directly affect its chances of survival: food, water, expenditure of energy, avoidance of danger, and so on.

However, an animal may have innate subjective reward systems to guide its learning. Certainly, to predict an animal's natural behaviour according to the optimality argument, a behavioural ecologist will wish to argue that a certain pattern of behaviour optimises an animal's eventual reproductive success: the behaviour may do this by optimising some necessary intermediate criterion, such as the rate of energy intake. However, although the animal's behaviour may be optimal or near-optimal from the behavioural ecologist's point of view, the animal itself may not be trying to optimise the behavioural ecologist's criterion. The animal may have innate 'likes and dislikes', and it may award itself subjective rewards and punishments according to this innate scheme, and it might then learn to behave so as to maximise the expected discounted sum of these subjective rewards and punishments.

If so, the animal will be learning according to a special-purpose method, in the sense of McNamara and Houston (1985): in the animals' natural environment the subjective rewards and punishments might be good guides to behaviour, but it might be possible to construct artificial environments in which the subjective reward system was misleading and would lead the animals to learn to behave sub-optimally.

As a hypothetical example of how a subjective reward system might aid learning, suppose that there were animals constructed just like the cart and pole mentioned in chapter 2. Suppose that these animals experienced pain and possible injury whenever they fell over. Then the primary reinforcers that would

motivate the animals to keep balancing are the primary punishments that occur when they fall over. However, these primary reinforcers would provide rather unhelpful information as to how to keep balancing—indeed, this is why the pole balancing problem is interesting. Although the animals could in principle learn to balance using these primary reinforcers alone, an individual that had an innate preference to be near the middle of the track, nearly vertical, and moving slowly, and which awarded itself a subjective reward whenever it was in this position, would learn to balance more quickly and with less injury than its less fastidious competitors.

An innate subjective reward system, then, can serve as an encoding for useful behavioural strategies: an encoding that is 'decrypted' by associative learning in an appropriate environment. Animals must have innate subjective reward systems that enable them to recognise the primary reinforcers—food, injury, sex—when they occur. The development of further innate subjective rewards may, therefore, also be possible.

1.3. Methods of Representation and Approximation of Functions

The aspects of the state of the environment that the agent can distinguish, the range of actions that it can encode, and the types of functional relationship that it can construct, are all innate characteristics that can affect the rate of learning. In particular, the method of approximation of functions that the agent uses in constructing its policy and value function incrementally will affect the generalisations that it makes to states that it has not previously encountered.

This is the type of prior knowledge that can be included in pattern recognition systems: what features to use, what class of function to construct to fit the observational data.

1.4. Initial Policies, Value Functions, and Models

In incremental dynamic programming, the agent's starting state will affect the amount of learning that it needs to do. All the learning algorithms I have described consist of the gradual modification of an initial strategy. Clearly, if the initial strategy happens to be optimal, then it does not have to be modified at all.

In the same way, if an agent were to construct a model of the world empirically, then the closer its initial model is to predicting events correctly, the less learning has to occur.

This type of innate 'knowledge' should be carefully distinguished from the next type, to which it might appear superficially similar.

1.5. Bounds and Constraints on Policies, Value Functions, and Models

The difference between a constraint and an initial state is that a constraint applies throughout the whole course of learning, whereas the initial state is altered during learning, and may eventually be entirely lost. A constraint, in contrast, may influence any stage, or all stages of learning.

One possible type of constraint that incremental dynamic programming can take advantage of is a constraint on the value function. For example, suppose one of the hypothetical pole-balancing animals happened to fall over, or be blown over, when it was standing still in the middle of its track. It might, erroneously, assign a low value to that state, which under the optimal policy is very safe. One type of innate knowlege that could help to prevent such a mistake and to speed up the learning is to place innate bounds on the value function: the constraint might, for example, prevent the value function in the safe states from ever falling below some limit value.

The strongest form of innate constraint on the policy is that the policy, or a part of the policy, may be innately fixed, so that the agent has innate responses to certain situations.

A crucial difference between a constraint and an initial state is that a poor initial state need not prevent an agent from attaining optimality in the end, whereas if the agent has innate constraints that are inconsistent with the optimal policy and value function then it can never reach optimality.

1.6. Tendencies to Experiment

Finally, an agent may have innate knowledge in the form of tendencies to experiment. These tendencies may be general, in the form of curiosity or the lack of it, or they may be specific.

Specific innate knowledge may be encoded as specific tendencies to experiment. These tendencies are quite a different form of innate behaviour than an innate policy: they are *tendencies* only, not constraints. These behavioural tendencies may lead the agent to perform potentially useful actions that it may then incorporate into its policy, or they may simply be actions that lead the agent to regions of state space where it will obtain useful experience.

A tendency to experiment might be very difficult to distinguish experimentally from an innate behaviour—but from the point of view of the learning algorithm, an innate behaviour is an unchangeable inheritance, while if experiments do not work the animal's policy is unaffected, and the experiments may eventually be abandoned if their estimated action value becomes too low.

The distinction between tendencies to experiment, innate constraints, and starting values is delicate but important. A tendency to experiment may affect learning at any stage, and may speed up or slow down learning, but it does not in principle affect the capacity to learn the optimal policy in the end. A starting

policy or value function may greatly affect the course of learning, but its effect will decline as time goes on. Innate constraints have a permanent effect, and may aid learning in environments to which the constraints are appropriate, but may prevent learning in inappropriate environments.

1.6.1. Particular Importance of Recommendations of Actions

Tendencies to experiment may be a particularly important type of prior knowledge for learning systems because they provide a particularly useful type of information—information as to which actions it might be profitable to perform in the current situation.

The point is that in primitive learning, an agent can only determine the consequences of an action by performing it. If many actions are possible in a particular state, the agent must revisit the state many times and try out the many different actions before it can be confident that it knows which action is best. This is a slow process if there are many possible actions, because an animal may only observe the results of one action at a time.

A tendency to experiment might be encoded in the form of recommendations for action, perhaps as an 'if-then rule':

IF the current state is of a certain type
THEN try performing a certain action
(if it has not been tried often already, and if
the estimated action value of it is not too low)

The IF part of the rule is a description that the current state should satisfy; and many types of description are possible. The description may be couched either as a direct description of the state, or it may be couched in terms of characteristics of the state that may themselves have been learned. In particular, the state-description might be one such as

IF the current state has been found to be correlated
with the imminent occurrence of event F,
THEN try performing action B.

Innate knowledge in this form might be useful because an agent cannot help but collect more information about *correlations* of events with states than it can collect about the *effects of actions* in states, because the agent must revisit a state many times before it can learn the effects of all actions, whereas on each visit it may observe subsequent events, and so improve its knowledge of correlations.

The crucial point is that it is possible for an agent to learn many different types of correlation in parallel, but the agent can only perform one action at a time. Thus prior information in the form of recommendations for action based upon correlations of events with states could be an important type of innate knowledge to provide.

2. Advice and Imitation

This discussion would not be complete without considering how an agent could make use of observations of other agents or advice from a teacher. The ability to experiment without corrupting the existing policy and value function allows an agent to try out advice, or to imitate another agent during learning. The actions advised, or the actions the agent performs while imitating another agent more skilled than itself, may be performed as experiments, so that the advice or imitation may be incorporated into the policy if it turns out a success, or it may be ignored and rejected if it fails.

This raises the possibility of automatic controllers with a manual override such that a human operator could take control and try out a strategy of his own

devising. The automatic controller could observe the actions of the human controller, and would analyse the action sequence as if it were a sequence of experiments that the automatic controller had performed itself. If the human operator managed to perform better than the automatic controller's current policy during the period of manual control, the automatic controller could use its observations to improve its internally represented policy and value function.

3. Restrictive and Advisory Innate Knowledge

In this chapter, I have described various ways in which innate 'knowledge' could be provided to affect learning by incremental dynamic programming. Such knowledge can be provided in a much greater variety of ways than is possible in simpler learning processes, such as supervised learning in pattern recognition.

There are two points that I wish to emphasise most. First, the main limitation of the primitive learning methods is likely to be the amount of experimenting that is required: the main use of prior knowledge may be to reduce the amount of experiment needed by recommending actions.

Second, behaviour need not be *either* innate *or* learned—it can be *innately learned*, in that the agent learns the behaviour with the help of innate knowledge. This innate knowledge may be either *restrictive* or *advisory*. Restrictive innate knowledge limits the range of behaviours that the agent can ever learn, while advisory knowledge may affect the rate of learning, but it does not in principle affect the range of behavioural strategies that the agent could ultimately acquire.

Chapter 9
Learning in a Hierarchy of Control

1. Limitations of Learning Methods Described So Far.

The learning methods so far described are suitable only for small problems. The methods rely for their success on visiting a sufficient variety of states and trying out sufficient actions: if the state-space or the number of possible actions is large, then the learning methods will take an impractical amount of time and experiment.

Besides the limitation on the size of the state-space, there is also a limit on the values of γ for which learning is practical. The limit is not so much because of learning time—the action replay argument of appendix 1 shows that the amount of experience necessary is proportional to the number of steps that have to be considered. Instead, the limitation is that if γ is very close to 1, then the action values must be represented to high precision, the difference between the action values of optimal and sub-optimal actions is proportional to $(1-\gamma)$.

Indeed, up until now, I have deliberately avoided the question of what value the discount factor γ should have. I have done this because there is no straightforward optimality argument. Certainly it is sometimes possible to construct an argument that an animal should seek to optimise discounted returns. Suppose that an animal forages for limited periods of time; a foraging period ends when the animal is interrupted, and interruptions occur randomly at a constant rate. Once an animal has been interrupted, it must start foraging anew, and it cannot return to the state where it left off in the previous session. In this

case, it is clear that if the animal knows that it could obtain a reward r after a further n time steps provided that the foraging session continues until then, then it should value that future reward less than a reward of the same size that could be obtained immediately, because the animal might not obtain the future reward because the foraging period might be cut off by an interruption. If the probability of continuing the session until the next time step is $\gamma < 1$, then the animal will be able to obtain a reward n steps into the future only with probability γ^n. It should, therefore, value a reward r that could be obtained n steps hence at only $\gamma^n r$.

This optimality argument for seeking to optimise discounted rewards is only sometimes relevant to foraging, and where it applies, the discount factor is a probability of interruption per *unit time*. However, the difficulty of learning is proportional to the discount factor per *decision* that the agent makes. Although an agent may be subject to random interruptions, so that a time-based discount factor is appropriate, it may still need to make far too many decisions between interruptions for learning to be feasible using the time based discount factor.

One possible response to this difficulty is to suppose that animals might seek to optimise their policies relative to a short time horizon, even though a longer time horizon (a larger discount factor) would be more appropriate for their needs. But this is not a sufficient argument.

As behaviour is analysed in more detail, the rate at which an animal appears to take decisions becomes larger. If the whole of an animal's detailed behaviour were to be described in terms of a single Markov decision process, the state-space would be enormous and the rate at which the animal would take decisions would be so great that it would not be plausible to explain learning in terms of incremental dynamic programming in the one decision process.

One way of reducing the apparent complexity of behaviour is to analyse it in terms of a hierarchy of strategies. In this chapter, I will present one method of constructing hierarchies of Markov decision processes, and I will describe how it is possible for the controller at each level of the hierarchy to learn independently of the others.

2. Hierarchical Control

In a control hierarchy, an action at the top level consists of an instruction to a lower level of control; in response to the top-level instruction, the lower level of control may carry out one or more actions, and each action at the lower level may be an instruction to a still lower level of control, and so on. In the language I have been using, once an 'action' has been chosen by the top level, it then forms part of the *state* for the lower level of control. The lower level of control chooses actions on the basis of its current state, which consists of the action specified by the top level, and other information, such as the appearance of the surroundings, recent history, and so on. An example may make this clearer.

Suppose that a navigator and a helmsman are sailing a ship together. The navigator knows the intended destination, and he is responsible for deciding how to get there. To do this, he plots possible routes on a chart, estimates expenses and times, and, by look-ahead, he chooses an initial course to take. The navigator then tells the helmsman in which direction to steer the ship. The helmsman's responsibility is to steer the ship in the direction chosen by the navigator. Suppose that in order to change the direction in which the ship is sailing, the helmsman has to perform a number of actions such as moving the rudder and adjusting the sails, and that the helmsman may have to perform several such actions over a period of time to achieve a change of direction; the

heading of the ship may drift, so that the helmsman may have to take intermittent actions to keep the ship on the specified course. The navigator is the top level controller, and the helmsman is the lower level controller.

This example can be described in terms of two interacting Markov decision processes, one at the top level, and one at the lower level. The navigator's decision process is the following:

- The state consists of the position of the ship, and the direction of the wind.
- The actions consist of sailing in a chosen direction for a certain (given) interval of time.
- There is a fixed cost during each time interval, and a large reward is received when the ship arrives at its port of destination.

The navigator makes decisions at fixed intervals of time, and each decision is a choice of a direction in which to sail. The information that is relevant to the navigator's decision is the position of the ship, and the direction of the wind, for this determines what directions are feasible, and how fast the ship will sail on each possible course. If the intervals at which the navigator makes his decisions are much longer than the time needed for the helmsman to change course, then the navigator need not take the current direction of the ship into account in deciding in which direction to sail next. One might imagine, for example, an 18th century navigator in mid-ocean, who would take his position laboriously with a sextant at noon every day, and who would then issue orders to the helmsman as to what course to steer for the next twenty four hours, after which he would retire to his cabin. If, on the other hand, the ship were sailing in a harbour, and the navigator needed to give commands at short intervals, then he would need to incorporate the current direction and speed of the ship into his description of state.

The helmsman's environment may also be described as a Markov decision process, in which

- The current state consists of the current direction of the ship, the direction of the wind, the configuration of the rudder and sails, and the navigator's current orders.
- The actions are to alter the configuration of the rudder and sails.
- The rewards are given by the navigator: the navigator punishes the helmsman when the ship is going in the wrong direction and rewards him when it is going in the right direction.

Suppose that the helmsman behaves in such a way as to maximise his expected discounted rewards. The rewards may come at every time step—if the navigator is sitting on the ship's bridge monitoring progress, for example. Even in this case, the helmsman's optimal strategy may sometimes be to take a punishment now for more rewards later on: if the helmsman is changing course, it may sometimes be more efficient to turn the ship through more than 180 degrees. Alternatively the rewards may come at longer intervals—for instance, if the navigator emerges from his cabin at random intervals and either praises or berates the helmsman according to the current heading of the ship. If both the navigator and the helmsman follow optimal policies in their respective Markov decision processes, the progress of the ship will be smooth.

As I have described this example, the navigator at the top level is using a sophisticated model-based mode of control. The helmsman might be using a more 'primitive' explicitly represented policy, or perhaps a hybrid method. But an important point is that any mode of control may be used at any level.

For example, the navigator could be completely lost and could be trying to find land by following the concentration gradient of flotsam using the search strategy of *E. Coli*. In this case the mode of control at the top level would be

more 'primitive' than that at the lower level. But in general one might expect model based methods to be used at the higher levels of control, for two reasons. First, at higher levels decisions are made at a lower rate, so that there is time for more computation for each decision. Second, at higher levels, observational data on the effects of actions is acquired more slowly, so that primitive learning methods are less suitable as they require more observational data.

The point of formulating a control hierarchy in this way is that the control problem at each level is that of controlling a Markov decision process. The control system should be designed in such a way that simultaneous optimal performance at each level results in good performance of the whole system.

The coupling between the levels is achieved by links between the decision processes, rather than by direct links between the controllers. An action in the higher process is a command for the lower process, which causes the lower controller to pursue a certain policy. This command, however, does not go directly to the lower controller—it affects the decision problem at the lower level, in two ways.

First, the higher action causes a change in the state of the lower decision process. This change in state affects the action that the lower controller takes. Second, the change in state does not have meaning by itself: the commands from the higher controller are given meaning by alterations in the reward system for the lower process.

2.1. Supervisory and Delegatory Control

It is worth discussing one distinction that marks a striking difference between natural and artificial systems: the difference between *delegatory* and *supervisory* hierarchical control. In delegatory control, the top level passes a command to the lower level, and the lower level then seeks to carry out the

command. The lower level then has the responsibility for transferring control back to the higher level: control is returned to the higher level when the lower level has successfully carried out the command, or perhaps when the lower level has decided either that it has failed, or that the command is impossible to complete. The point is that once the command has been given, the lower level is in control until it passes control back to the top level. In this sense the top-level *delegates* control to the lower level.

This type of hierarchical control is natural for sequential programming languages. It has, however, the drawback that the lower level may not correctly pass control back to the top level: the top level of a program may (all too often) wait indefinitely for an errant function call to return. This type of behaviour, so common in sequential machines, is totally uncharacteristic of animals and people.

In *supervisory* hierarchical control, the top level retains the initiative. The current top-level command is part of the state for the lower level; if the top-level changes its command, the state for the lower level changes, and the lower level may as a result choose a different action. On the ship, the navigator may keep track of the state from his point of view—which is the current position—and he may then issue new instructions when the ship reaches appropriate positions. The navigator may change his current command at any time in response to unexpected circumstances. Control does not pass to the helmsman once the navigator has given his command, and then back to the navigator only when the helmsman has finished: the navigator retains his initiative and *supervises* continually at the top level.

While learning is possible, I believe, in both types of control hierarchy, this distinction is important because in artificial intelligence hierarchical control is often tacitly assumed to be delegatory rather than supervisory. But the

concept of 'flow of control' in a computer program is one that is alien to natural systems, which appear to be inherently concurrent.

3. Learning in a Control Hierarchy

In a control hierarchy, the controller at each level may seek to learn to optimise its policy, independently of the others. If this learning is successful, the result will be that all controllers will acquire optimal policies in the Markov decision problems at their levels of control: this *optimality at each level* must be carefully distinguished from *global optimality* of the entire control system. In this section, I will consider how optimality at each level can be achieved by learning.

It might initially seem that, since each controller has its own decision process to control, learning of optimality at each level is possible with no additional constraints; but two types of difficulty can arise. Both difficulties are the result of the freedom of behaviour that learning by incremental dynamic programming allows—the learner acquires the capacity for optimal behaviour, but it need not always behave optimally. The definition of optimality at each level needs to be made more precise:

- A hierarchical control system is *optimal at each level* if, when all controllers follow their estimation policies, all of the estimation policies are optimal.

The point of this definition is that the effects of the actions of a high level controller are defined relative to the behaviour of a lower level controller. A high level action consists of changing the state and reward system of a low level controller: the effect of a particular performance of a high level action is what the low level controller actually does in response to the changed state—but if low level controllers are to be able to learn they must be free to experiment.

In terms of the ship example, the system is optimal at each level if

i) for each possible command of the navigator, the helmsman has acquired a policy that is optimal with respect to obtaining the navigator's rewards.

ii) the navigator's policy is optimal, provided that the helmsman always follows his optimal policy for the current command.

The simple vision is that each level of the control hierarchy should be a Markov decision process; if this is so, then each controller may learn to control its own decision process independently of the other controllers. But there are two potential complications which may destroy the Markov properties of the decision problems—lower controllers may experiment instead of following their orders, and higher levels may directly and arbitrarily affect the state transitions at the lower levels. Both of these problems may be solved.

3.1. Lower Controllers may Behave Sub-Optimally

If the helmsman does not follow his optimal policy for the current command, but persists in experimenting with different directions, then the navigator may be disappointed if he follows a policy that would be optimal on the assumption that the helmsman always obeyed orders. But the helmsman needs to be free to experiment if he is to be able to learn; and if the helmsman chooses a wild series of experiments so that he does not follow the same policy each time the navigator issues a command, then he may prevent the navigator from experiencing a Markov decision problem.

How can this difficulty be overcome? One approach is to stipulate that lower levels of control must always follow their orders to the best of their ability; but this would make it more difficult for the lower levels to learn optimal policies. A less severe restriction would be to allow the lower levels some limited freedom to experiment: this requirement might be formalised by allowing

the lower levels to pursue a stochastic policy with the probabilities dependent upon currently estimated action values. But this suggestion has an ad hoc flavour, and does not permit the arbitrary experimentation or advice-following that incremental dynamic programming should allow.

Another possibility is that higher levels of control should be able to command lower levels of control to follow their current estimation policies if necessary, but that they should sometimes allow the lower levels some 'time off' in which to improve their policies by experiment. In a hierarchy of several levels of control, if the *n*th controller from the top wanted to learn by experiment, then all controllers above it would be switched off, and all controllers below it would be instructed to follow orders.

A more flexible idea would be for the lower controller to pass a message saying "I am experimenting now" to the higher controller whenever it deviated from its policy by more than a certain amount. When the upper controller received such a message, it would simply not use the current stage of experience as one of its training observations. The lower controller would still have the freedom to behave as it liked—but it would have to declare its experiments to the controller above. This system has the virtue that experiments are allowed at any time during normal performance of the whole system. It would also be compatible with allowing occasional directives from higher levels to lower levels saying "Follow your optimal policies!". In this way the experience of the higher controller may be made Markovian in that the effects of its actions may be made consistent.

3.2. Higher Controllers may give Non-Markovian Commands

The second problem is more subtle. Recall that the effect of a high level action is to alter the state of the low level controller. If the low level controller is simultaneously performing actions of its own, it might 'conclude' that the change in state was caused by its own actions, and not by the high level action. Would this matter? It might indeed, because the high level controller has complete freedom of action, and it is therefore possible that it might issue commands in such a way that the effects of actions in the lower decision process appeared non-Markovian. The entire discussion of learning methods has relied heavily upon the Markov property, and if state-transitions and rewards are non-Markovian, then the learning algorithms can no longer be guaranteed to work.

A simple solution to this problem is to require that when the higher level controller initiates a new action, it should send a message to the lower controller saying "High level action now changing". While this message is in force, the lower controller should not use observations of its state-transitions, actions, and rewards as training data in optimising its policy, because the high level action changes may be arbitrary and need not be Markovian with respect to the low level state space.

Provided that high level action changes are occurring for only a small proportion of the time, this solution is valid, in that it prevents the lower level controller from corrupting its learning process with non-Markovian data.

But there is a much more interesting solution. The lower level controller seeks to protect its learning from non-Markovian data that is caused by arbitrary actions of the higher level controller.

Just suppose, for a moment, that the high level actions were in fact probabilistically chosen, and dependent only upon the current lower level state

(which includes the current higher level action), and the current lower level action. If this were so, then the higher level actions would appear to be caused by the lower level states and actions, and higher level action's effect on the state transitions and reward would appear to be caused by the lower level actions. If the lower level actions and states drove the higher level controller in this way, then the effects of the higher level actions would be Markovian with respect to the lower level process, and there would be no need to protect the lower level learning from the effects of the higher level actions.

This odd situation in which the lower level process controlled the higher level process should not actually occur in practice, but the lower level controller can observe correlations of the high level actions with its current state and its choice of action, and the lower level controller may then use the correlations to *predict* the higher level actions from the lower level state and action. If the higher level controller always chose the actions the lower level process predicted, then the lower level process would seem to control the higher level controller, and all the lower level data could be used in learning.

If the lower level controller can make strong predictions about the higher level actions, then it can, in effect, alter the definition of its Markov decision process so that the effects of lower level actions include the predicted higher level actions: to ensure that its learning data is Markovian, the lower level controller then only needs to reject those observations in which the high level action was not predicted.

The lower level controller may, therefore, learn to predict the upper controller's actions, and then optimise its policy *relative to its predictions.* I have not yet determined under what circumstances this method of learning would be stable.

4. The Sequence of Learning

Initially, neither the top-level controller—or 'master'—nor the lower level controller—or 'slave'—possesses an optimal policy. The first stage of learning is that the master may perform arbitrary actions, and the slave learns to obey some of the master's commands. During this first stage, the master may possibly not be able to learn anything, since the slave does not know how to obey the commands. Even if the master does alter its policy, the slave's policies will be changing, so that there is no guarantee that the master's initial policy changes will produce any lasting improvements.

In the second stage, the slave has learned to obey enough commands well enough for the master to start to improve its policy; as the master improves its policy, it will tend to give the commands it finds useful more often than the commands that it does not find useful. The slave will, therefore, gain more experience with the commands that the master finds useful, and its performance on these will further improve.

Ultimately, the master and slave will acquire stable or meta-stable policies. One reason why the slave's policy may be meta-stable is that it may not gain enough experience to learn to obey commands that the master did not initially find useful; the master may cease to give these commands frequently, so that the slave continues to have little experience with them. Note that the slave cannot correct this deficit in its experience by initiating its own experiments, because the *state* in which it may gain the relevant experience is that in which the *master* has given the appropriate command; hence it is partly the master's responsibility to ensure that the slave receives a sufficient variety of opportunities of experience.

Note also that the master's policy may be optimal with respect to the slave's current abilities, but that the slave's policy may be meta-stable in that it

cannot obey certain potentially useful commands. Thus, even though from the master's point of view, its policy is optimal, the masters policy may be meta-stable if the slave has not yet learned certain abilities.

As the master's policy ceases to change rapidly, the slave may, if it has the ability to do so, learn to predict the master's commands on the basis of its own state (and, perhaps, action). In doing this, the slave is altering its own decision problem, and it may adapt its policy accordingly. This changes the way in which it obeys commands, and so causes the master to adapt its policy also. I do not know under what conditions this mutual adaptation of master and slave is convergent, beneficial, or stable.

5. Discussion

In this chapter, I have presented informally a method of formulating hierarchical control problems as coupled Markov decision problems, with autonomous controllers at each level. Provided that the controllers are able to communicate in some simple ways, it is in principle possible for all the controllers to converge to optimal policies for their own decision problems. This hierarchical learning provides a mechanism for developmental self-organisation of a hierarchy of skills, using the method of incremental dynamic programming at each level. The rewards at each level may be determined both by the level above, and also by the environment: at each level of the hierarchy, therefore, the development of the skills may take account of environmental constraints.

There are also possibilities for mutual adaptation of the controllers in coupled decision problems, and there are, of course, many more possible configurations of coupled Markov decision problems besides a simple hierarchy. There are fascinating possibilities for further research here. Unfortunately I have not yet implemented any examples of these hierarchical control systems.

Chapter 10
Function Representation

1. Limitations of Finite Problems

So far, I have only discussed problems in which there are finite numbers of states and actions. This is because, in a finite problem, each action value may be adjusted separately, and the learning methods are easier to describe and to analyse. Although in principle, problems with continuous sets of states or actions may be approximated as finite problems, the finite approximation may need to contain a large number of adjustable parameters—one for each state-action pair, perhaps. At each step of learning, only one or a few of these parameters may be adjusted significantly. Learning will, therefore, be slow if there are many states.

Conventional implementations of dynamic programming use a finite-state approximation. The state-space is typically a region in a Euclidean space, and the value function is usually represented explicitly, by storing values at each of a regular grid of locations over the state space. (Action values are computed temporarily for each state, but, once the maximal action value has been determined, the other values are not usually stored.) With this method of representation, the amount of storage required is proportional to the number of points on the grid. For a high-dimensional state-space, one is faced with the choice of either having a coarse grid which may represent the function inaccurately, or else having a very large number of grid points with a correspondingly large requirement for storage, and also for calculation, since repeated calculations are necessary to obtain the value at each grid point. This is the 'curse of

dimensionality' that limits the applicability of dynamic programming in practice (Bellman and Dreyfus 1962). For learning by incremental dynamic programming, the most important constraint is the amount of experience that is needed: this corresponds roughly to the constraint on the amount of calculation in conventional implementations.

The natural and straightforward representation of a function as values at points on a grid gives, for most practical applications, a representation with far too many independently adjustable parameters. But this difficulty of representing a function on a high dimensional space may often be finessed by choosing a more compact form of representation than that of storing the value at each point on a regular grid. For example, Omohundro (1987) describes a number of representations for functions on multi-dimensional spaces in which standard computer science techniques are used to store the function economically. However, the methods of representing functions that have recently attracted enormous interest are 'artificial neural networks'.

An 'artificial neural network' is a (usually simulated) collection of small processing elements that are intended to model, or to have an analogous function to, neurons or small collections of neurons. Each element, or 'cell', is connected to many other cells; these connections may be one-way or two-way. Each connection has a numeric weight, which may be adjusted during 'learning'. (Part of) the current state of each cell consists of a numeric 'level of activation'.

The level of activation of a cell depends upon the levels of activity of the other cells from which it receives inputs, and upon the weight attached to each of these inputs. A cell's level of activation may affect other cells through its outputs. Changes in activity of some of the cells may therefore have effects that propagate through the system.

In a neural network, there are two forms of computation, short-term, and long-term. The short term computations are the changes in the levels of activation of the cells that follow a change in the inputs to the net. The long term computations consist of slow adjustment of the weights to improve the agreement of the short term computations with training examples.

In recent years, a number of new forms of computation and training methods for neural nets have been published; a number of these are described in, for example, Rumelhart and McClelland (1986); a good, concise review of various types of neural net is in Lippman (1987).. What all of these methods have in common is that they can all be viewed as devices for approximating functions; the approximation is achieved by presenting the network with *training examples*, each of which is an example of an input paired with the desired corresponding output. In the majority of the current work on 'neural networks', it is assumed that a neural network is a device that 'learns' to compute a function from training examples.

The point I want to make here is that it is widely assumed that at the level above individual neurons, the computational modules of the nervous system are collections of neurons that learn functional mappings from training examples. This assumption may be right, half right, or plain wrong—nobody can know yet. But, in primitive learning, the basic computational modules that are required are modules to learn the action value function, or to learn a value function and a policy, or a value function alone, from 'training examples' that are extracted from observations of experience. In this rather shallow sense, the learning methods I have described appear to be fully compatible with current assumptions about neural computation.

There is, however, a problem. All methods of representing functions that require less storage than is required by grids must necessarily also have fewer

adjustable parameters, for it is the adjustable parameters that must be stored. The effect of this is that each adjustment during learning will affect a whole region in state-space, and not the value stored for just one point. If an adjustment is made, for example, to a stored estimated value function as a result of an observation of an action taken from point x, the adjustment will not just alter the estimated value for point x but also the estimated values for many other points y. Which other points have their estimated values adjusted, and how, will depend on the particular approximation and adjustment method used. In incremental dynamic programming, the overall effect of the adjustments may be either to speed up or else to hinder or prevent convergence.

So the problem is: what types of parameter estimation and function representation will work with the types of incremental adjustments that are provided by the learning methods? And for what classes of Markov decision processes can each function estimation method be used? This is not so much a single problem as a field of research. I have not attempted to tackle this problem theoretically, but I have implemented a demonstration using one particular function approximation method—the 'CMAC'.

2. Function Representation Using the CMAC

The CMAC (or 'Cerebellar Model Articulation Computer') was proposed by Albus (1981), partly as a speculative model of the mode of information storage in the cerebellum, and partly as a practically useful method for the storage and incremental approximation of functions, for use in robot control. I will consider the CMAC purely as a method for the local approximation of functions, for which it is undeniably useful, rather than as a neural model, in which role its status is questionable. Considered mathematically, it is a method of approximating scalar or vector functions over a multi-dimensional space

using pre-defined step functions. The advantage of the method is computational speed and simplicity, rather than accuracy or storage economy.

A CMAC works as follows. The purpose is to maintain and adjust a representation of a function on a space: that is, for each point x in the space, the CMAC specifies a scalar or vector value $U(x)$. To store the values, the space is partitioned into rectangular regions, or *tiles*. A *tiling* of the space is a covering of the space by non-overlapping tiles, such that each point in the space is contained in exactly one tile. The tiles may be denoted t_1 , t_2 , $\cdots$ If the tiles are chosen to be rectangular, and aligned with the coordinate axes, then it is very easy to compute which tile any given point x is in.

A CMAC for a given space consists of

- a number of tilings of the space, so that each point in the space is contained in a number of overlapping tiles.
- an array $u[1]$, $\cdots$, $u[n]$ of scalar or vector values, which are adjusted incrementally during approximation.
- a hash function *hash* which maps each tile to an element of u.

Suppose there are 10 different tilings, which are:

$$t_{1,1} , t_{1,2} , t_{1,3} , \cdots$$

$$t_{2,1} , t_{2,2} , t_{2,3} , \cdots$$

$$\cdots$$

$$t_{10,1} , t_{10,2} , t_{10,3} , \cdots$$

The tilings overlap, so that each point in the space will then be in exactly ten tiles, one tile from each tiling. The method given by Albus, and the method I have used, is to choose similar tilings, but to displace them relative to each other, so that no two tilings have their tile boundaries in the same place. A

point travelling through the space will, therefore, cross tile boundaries at frequent intervals: in other words, the space is finely partitioned into small regions, with one region for each combination of overlapping tiles.

A finite number of adjustable parameters are used to store the function—they are $u[1]$, $\cdots$, $u[n]$, and their values are stored in an array. Now, each tiling is infinite in extent, so that it is not possible to store a separate parameter for each tile: the solution is that each tile is mapped to an arbitrarily chosen element in the array. This mapping is defined using a hash function, and it is fixed when the CMAC is constructed. The effect is that each array element—each adjustable parameter—is associated with one in n of the tiles, selected in a pseudo-random fashion throughout the space. To visualise this for a two dimensional space, consider some parameter $u[i]$; if the tiles mapped to $u[i]$ were coloured in, then on looking at the space the visual effect would be that a pattern of tiles would be coloured in, approximately one in n tiles being coloured, and this pattern would stretch as far as the eye could see. If the tiles mapped to some other array element were also coloured in, a second such pattern of coloured tiles would appear, and no tile would be coloured twice, since each tile is mapped to exactly one array element. Let us represent the mapping from tiles to array elements as *hash*. If the tile t_{ij} is mapped to the kth element of the array, then this is denoted by $hash(t_{ij}) = k$.

To compute the function represented by the CMAC at a given point x in the space, the following steps are carried out. (Assume that the CMAC has been constructed with 10 tilings.)

Computing the Function

1. The 10 tiles containing x are found.

   Let them be t_{1,i_1} , $\cdots$, $t_{10,i_{10}}$

2. The array elements corresponding to each of the tiles are determined.

   Let

$$k_1 = hash(t_{1,i_1})$$

$$\cdots$$

$$k_{10} = hash(t_{10,i_{10}})$$

3. The values of the array elements are then averaged, to yield the answer. That is,

$$U(x) = \frac{1}{10}\left[u[k_1] + \cdots + u[k_{10}] \right]$$

Note once more that the array u may store either scalar or vector values. That is, $U(x)$ is equal to the average of the array values associated with the tiles that contain x. This method of storing a function is an example of the method of 'coarse coding' as described by Sejnowski in Rumelhart and McClelland (1986).

The method of adjustment that Albus (1981) recommends, and which I have used, is the following. Suppose that a 'training example' is presented, consisting of a point x and a desired value v. The parameter values of the CMAC are adjusted as follows.

Incremental Approximation

1. The value $U(x)$, and the array indices k_1 , $\cdots$, k_{10} are determined exactly as in computing the function.

2. If $|U(x) - v| < \varepsilon$, where ε is a predetermined small positive constant, then no further adjustment is made, otherwise adjust each array element according to

$$u[k_1] := u[k_1] + \alpha\,(v - U(x))$$

$$\cdots$$

$$u[k_{10}] := u[k_{10}] + \alpha\,(v - U(x))$$

where α is a predetermined positive learning factor, less than 1. If the desired values v are subject to noise, then α should be small.

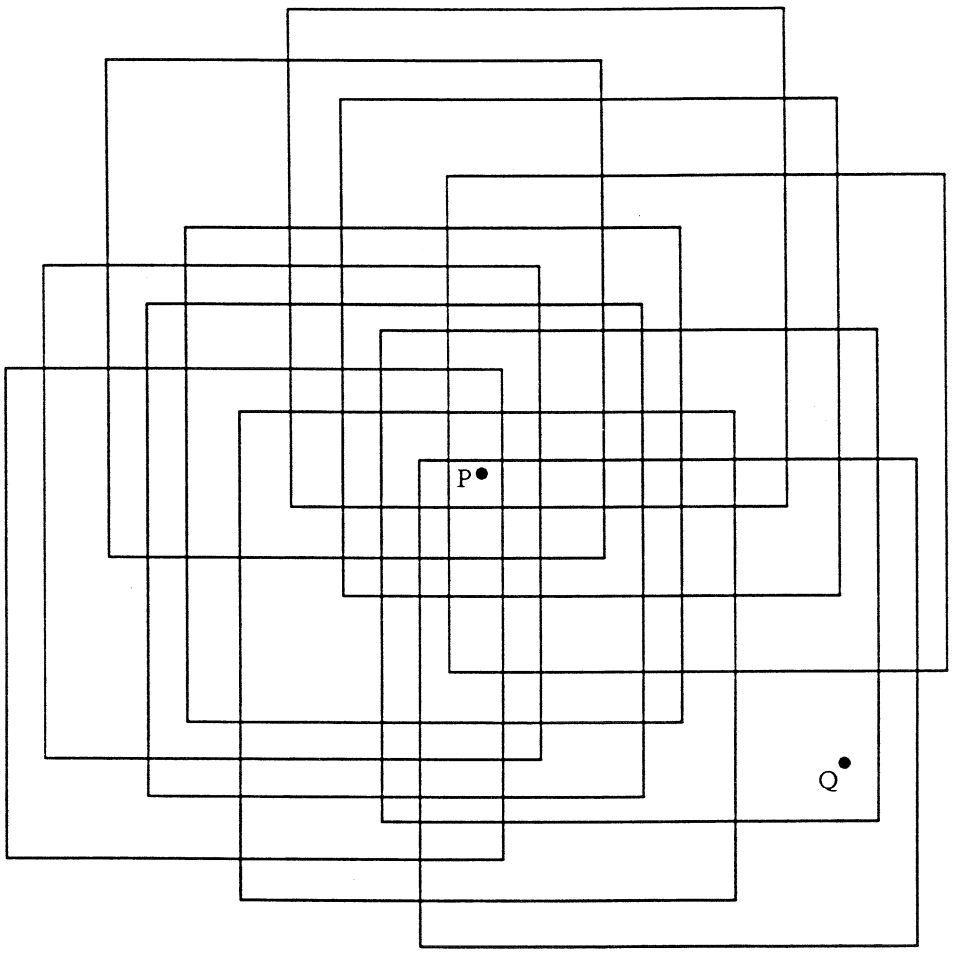
The effect of this adjustment method can be more clearly seen if we consider the set of tiles that contain a point x, as illustrated overleaf. The values associated with all these tiles are adjusted by the same amount, so that the values for points near x and contained in all 10 tiles will change by $\alpha(v-U(x))$, while the values for points further from x and contained in only one of the tiles containing x will change by an amount $\frac{\alpha}{10}(v-U(x))$.

A notation for the adjustment of a CMAC U, at a point x, towards a desired value v, with a learning factor α is

$$U' = \textbf{adjust}(\, U, x, \mathit{alpha}, \mathrm{v})$$

This notation is intended to indicate that there is a process of adjustment that will affect U in a region surrounding x; the CMAC function after the adjustment is U'.

How large must the u array be? That is, how many adjustable parameters are necessary? It will (usually) only be desired to approximate the function



This diagram shows the collection of CMAC squares that contain P.

Each CMAC square is associated with a stored value.

The value of the function at point P is the average of the values stored for the squares that contain P.

When the CMAC value is adjusted for point P, the values stored for all the squares that contain P are adjusted.

The value for point Q, which is contained in only two of the CMAC squares, will only be slightly affected by the adjustment.

over a *finite region* A of the space. The number of array elements needed depends on the size of the region A, and on the variability of the function in this region. If the array is too small, then many tiles within A will be mapped to each array element, and there may not be enough degrees of freedom to represent the function accurately over A. If, on the other hand, the array is so large that, on average, fewer than one tile in A is mapped to each array element, then there are (almost) enough degrees of freedom to adjust the value associated with each tile in A separately.

One practical advantage of the CMAC is that it is not necessary to know beforehand the region A of the space on which the approximation will be made. The training examples presented define the region over which the function is approximated.

Provided that the array u is large enough, the CMAC can be viewed as a local approximation method, in which a training example for a point x will affect the values only of other points near x. From a purely practical point of view, the great advantage of using a CMAC is that, unlike most other proposed 'artificial neural networks', it works fast and reliably.

Chapter 11
Two Demonstrations

In this chapter, I will describe two demonstration implementations of learning methods. The point of the demonstrations is first to show that the learning methods do produce improvements in performance in some simple problems, and second to demonstrate some qualitative characteristics of incremental dynamic programming. The first demonstration is of a synthetic 'route-finding' problem, and the second demonstration is analogous to a Skinner box with various simple reinforcement schedules.

The programs have been written in Pascal, and run on a Sun 3 workstation. The random number generator used is not the system supplied routine, but a subtractive generator recommended by Press et al (1986).

1. A Route-Finding Problem

The learning method described in chapter 7, section 4.3, is that of learning a value function and a policy, with policy improvement by a gradient method at each state. This method is interesting because the policy representation is economical—only one action need be stored for each state, rather than the storage of a value for each state-action pair.

The aim of the first demonstration is to display the functioning of the learning method as clearly as possible; it is not intended to model any particular real problem.

1.1. The State Space

The state space chosen is a square in the Euclidean plane, with sides of length 2, centred at the origin, and aligned with the coordinate axes. This enables the estimated value function and policy to be readily displayed: the value function is displayed as a 3D plot, and the policy is displayed by drawing the action vectors at each of a grid of points over the state space.

At any time, the current state of the agent is a point in this square.

1.2. Actions and their Effects

At any point in the square, the set of possible actions is the set of two dimensional vectors $\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}$, where Δx and Δy are arbitrary real numbers. These are the actions the agent can choose to perform. Broadly, the effect of an action $\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}$ taken at a point (x,y) is to move to a point $(x + \Delta x, y + \Delta y)$, with the proviso that the agent must always remain inside the square of the state-space.

If the agent makes a move that would take it out of the square, it is stopped at the edge. Suppose the agent is at state (x,y), and it chooses to make the move $\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}$. If it were unconstrained, it would 'land' at point $(x + \Delta x, y + \Delta y)$; for brevity, suppose this point is (x',y'). The agent cannot leave the square: its moves are curtailed by the rule

$$x' := \begin{cases} -1 & \text{if } x' < -1 \\ 1 & \text{if } x' > 1 \\ x' & \text{otherwise} \end{cases}$$

$$y' := \begin{cases} -1 & \text{if } y' < -1 \\ 1 & \text{if } y' > 1 \\ y' & \text{otherwise} \end{cases}$$

The effect of this rule can be seen in the diagram overleaf.

1.3. Rewards and Penalties

There are two sources of rewards and penalties. First, the agent is given a reward or penalty whenever it is in a specified region in the state space. In the coming example there is a 'target'—a small rectangle in the upper right hand quadrant of the state space; if the agent is in the target at time t, then it receives a large reward at time t whatever action it performs.

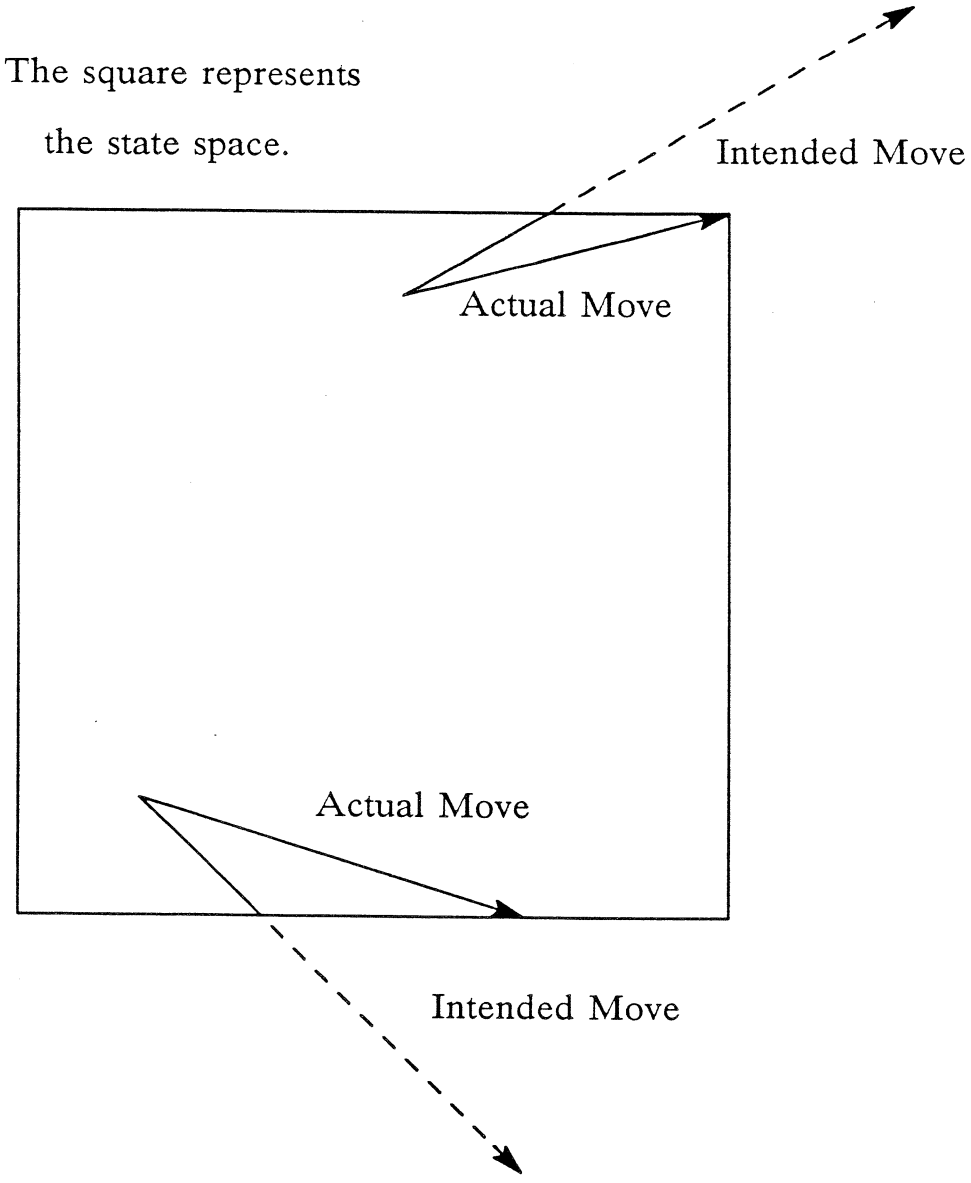
Second, each move the agent takes has an immediate cost. The cost of a move is a function of its length. If the length is d, then the cost $c(d)$ is

$$c(d) = c_1 d + c_2 d^2$$

where c_1 and c_2 are positive numbers that are kept constant for the duration of each run of the program.

The cost is a function of the length of the *intended* move, rather than of the actual distance travelled. The reason for this choice is that if an intended move leads outside the square, then the cost of a longer intended move is still greater than the cost of a shorter move in the same direction, so that the agent could reduce the cost of its action by reducing the lengths of its intended moves, even though the actual moves might all be cut off at the boundary of the state space, so that the actual distances travelled would be the same.

The cost function was chosen to be a combined quadratic and linear function of the length of the move for the following reasons. The quadratic term was included to penalise long moves, so that to reach a target a long way away, the optimal strategy would be to take a sequence of short moves rather than a single long move. If the cost of a move were simply a multiple of the square of its length, the cost of small moves would be very small, and there would be



The agent cannot leave the square: if it attempts to move outside the square, its move is curtailed, as shown.

no incentive for the agent to reduce the size of small moves on grounds of cost. To give the agent an incentive to reduce the size of even small moves, part of the cost was made proportional to the length of the move.

2. Demonstration of Action-Gradient Learning

The learning problem is set up as follows. The target is a small rectangle, placed in the 'top right hand corner' of the state space, and it is drawn as a small rectangle in the policy diagrams. If the agent performs any action when it is in the target,

- the agent receives a large reward, and
- the action the agent takes does not have its normal effect: instead, the effect is that the agent moves to a randomly chosen position in the state space. The landing position is chosen from a uniform distribution over the whole state space.

The point of re-positioning the agent in a randomly chosen position in the state space is to ensure that it gets experience of all parts of the state space.

2.1. The Learning Algorithm

Both the policy f and the value function U are represented using CMACs, as described in chapter 10. These functions are, therefore, defined at all points in the state space. It is not possible to assign new values or actions to states individually; instead, the CMACs must be adjusted according to the method described in chapter 10.

All cells in the CMAC initially contain zero for the value function and the zero vector for the policy, so that the initial policy of the agent is the zero vector throughout the state space, and the initial estimated value is also everywhere zero.

Actions are chosen by calculating the policy action for the current position, and then adding a randomly generated *deviation vector*. That is, the action chosen at time t is

$$\mathbf{a}_t = f_t(\mathbf{x}_t) + \mathbf{b}_t$$

where $f_t(\mathbf{x}_t)$ is the policy action, and $\mathbf{b}_t$ is a randomly generated vector of uniformly distributed direction. The mean length of the deviation vectors $\mathbf{b}_t$ is denoted by b. The value of b is supplied to the program at the start of a run. The lengths of the deviation vectors $\mathbf{b}_t$ are independent, and exponentially distributed, so that the probability density for $\mathbf{b}$ having length l is $\frac{1}{b} e^{-bl}$, for $l > 0$. The point of adding the deviation vectors is that in order to improve its policy, the agent must perform actions that are different from those recommended by the policy: this mechanism for adding deviation vectors to policy actions is just one simple way of achieving this. The exponential distribution was chosen simply because it is convenient, but I have also implemented the method with deviation vectors of constant length and random direction, and the behaviour of the learning system is similar. There is no reason to suppose that other distributions should not also give qualitatively similar behaviour, provided that the frequency of large deviations is not too great.

Estimated returns are computed as follows. The value of λ_t depends upon the length of $\mathbf{b}_t$, which is the random vector added to the policy action. That is,

$$\lambda_t = \exp(-\eta \, |\mathbf{b}_t|)$$

where η is the *rejection factor*, as described in chapter 7, section 3.1. η is kept constant throughout a learning run. The effect of η is to control the extent to which large deviations are rejected in estimating returns. If η is zero, then λ_t is always equal to 1, and all sequences of actions count equally in estimating

returns.

The estimate of return used in policy-adaptation is $\mathbf{r}(t)$, and the estimate of return used in value-adaptation is $\mathbf{u}(t)$. $\mathbf{r}(t)$ and $\mathbf{u}(t)$ are defined in a way that is similar, but not identical, to $\mathbf{r}_t^\Lambda$ and $\mathbf{u}_t^\Lambda$. $\mathbf{r}(t)$ is an estimate of the action value of x_t , a_t and $\mathbf{u}(t)$ is an estimate of the value of x_t. The difference between them is that if a_t deviates greatly from the policy action, λ_t will be close to zero, and $\mathbf{u}(t)$ will be close to the existing estimated value of x_t , so it will not cause any change in the estimated value function U. The definitions are:

$$\begin{aligned}\mathbf{r}(t) = r_t + (1-\lambda_{t+1})\,\gamma\, U_{t+M}(x_{t+1}) \\ + \lambda_{t+1}\,\gamma\, r_{t+1} + \lambda_{t+1}(1-\lambda_{t+2})\gamma^2 U_{t+M}(x_{t+2}) + \cdots \\ + \lambda_{t+1}\cdots\lambda_{t+M-1}\,\gamma^{M-1}\, r_{t+M-1} + \lambda_{t+1}\cdots\lambda_{t+M-1}\gamma^M U_{t+M}(x_{t+M})\end{aligned}$$

and the definition of $\mathbf{u}(t)$ is

$$\begin{aligned}\mathbf{u}(t) = (1-\lambda_t)U_{t+N}(x_t) + \\ \lambda_t r_t + \lambda_t(1-\lambda_{t+1})\gamma U_{t+N}(x_{t+1}) + \\ + \lambda_t\lambda_{t+1}\gamma r_{t+1} + \lambda_t\lambda_{t+1}(1-\lambda_{t+2})\gamma^2 U_{t+N}(x_{t+2}) + \cdots \\ + \lambda_t\cdots\lambda_{t+N-1}\gamma^{N-1} + \lambda_t\cdots\lambda_{t+N-1}\gamma^N U_{t+N}(x_{t+N})\end{aligned}$$

where M is the 'learning period' for $\mathbf{r}$ and N is the 'learning period' for $\mathbf{u}$. The point of these 'learning periods' is that they put an upper limit on the number of previous states it is necessary for the agent to store. The estimates $\mathbf{r}(t)$ and $\mathbf{u}(t)$ can be computed at times $t+M$ and $t+N$ respectively. A second difference between these estimators and the estimators $\mathbf{r}_t^\Lambda$ and $\mathbf{u}_t^\Lambda$ described in chapter 7 is that $\mathbf{r}(t)$ and $\mathbf{u}(t)$ use U_{t+M} and U_{t+N} respectively throughout. This difference between the two methods of calculation was caused by an oversight in the coding of the program; I do not believe that this difference in the method of calculation makes a significant difference to the results.

The learning rules are, in the notation of chapter 10, page 144:

$$U_{t+N+1} = \textbf{adjust}(\, U_{t+N} \,,\, x_t \,,\, \alpha \,,\, \mathbf{u}(t) \,)$$

and

$$f_{t+M+1} = \textbf{adjust}(\, f_{t+M} \,,\, x_t \,,\, \beta(\mathbf{r}(t) - U_{t+M}(x_t)) \,,\, a_t \,)$$

where $\beta = \beta^+$ if $\mathbf{r}(t) \geq U_{t+M}(x_t)$, and $\beta = \beta^-$ if $\mathbf{r}(t) < U_{t+M}(x_t)$.

The values of η, α, β, M, and N, and of all the other parameters mentioned below, may be supplied to the program at the start of each run.

3. Results

The behaviour of the program is consistent and reliable over a wide range of parameter values. I will show results from one run, and comment on them in some detail; results obtained using other parameter values and other random seeds are qualitatively similar. In informal experimentation with this and other learning problems, I have found that the changes in the parameters have the following qualitative effects, if the other parameters are kept the same:

- If the size of the CMAC patches is increased, learning is faster, but the final result becomes worse, since the representation is coarser.
- If the discount factor is made too close to 1, the policy may become unstable.
- Using a low policy learning factor for negative prediction differences improves the final policy.
- If the learning factors for the policy are made too large, the policy becomes unstable.
- η seems to have surprisingly little effect in this problem.

The parameter values used in the run for which results are presented were as follows:

Learning Parameters

Discount Factor	0.9
Learning Factors	
α	0.5
β^+	0.1
β^-	0.025
Learning Periods	
for value function (N)	2 time steps
for actions (M)	1 time step
Other Parameters	
Rejection Factor η	5.0
Mean Action Perturbation b	0.1

CMAC Parameters

Size of CMAC squares	0.2
Initial value	0 everywhere
Initial action	**0** everywhere
Size of CMAC table	40,000

Note that the returns used for adjusting actions are computed over only one time step: the reason for this is that the adjustment of the actions is a gradient descent process for finding a maximum, whereas the adjustment of the values is

a process of finding a mean value. The adjustment of actions, therefore, is more sensitive to noise in the estimated returns than is the value adjustment.

The run for which results are presented was selected arbitrarily, and the learning method will work over a wide range of parameter values. The main limitation is that if the learning factor for actions is too large, or if the discount factor is chosen to be too close to 1, the process becomes unstable.

The CMAC table may appear large for such a simple problem: in fact, the table may be reduced to little more than 1% of that size without undue degradation of the results.

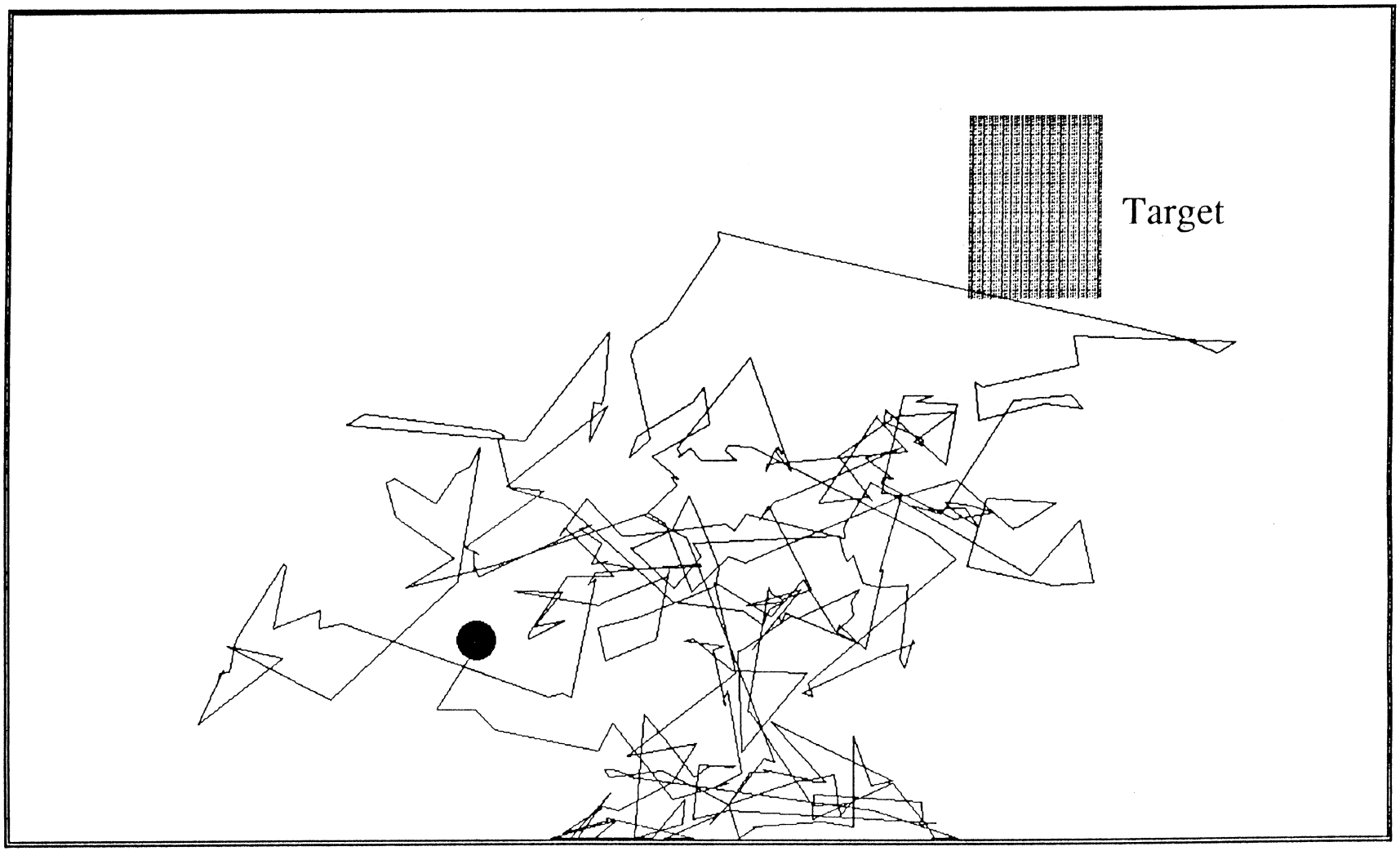
The costs and rewards are as follows

Costs and Rewards

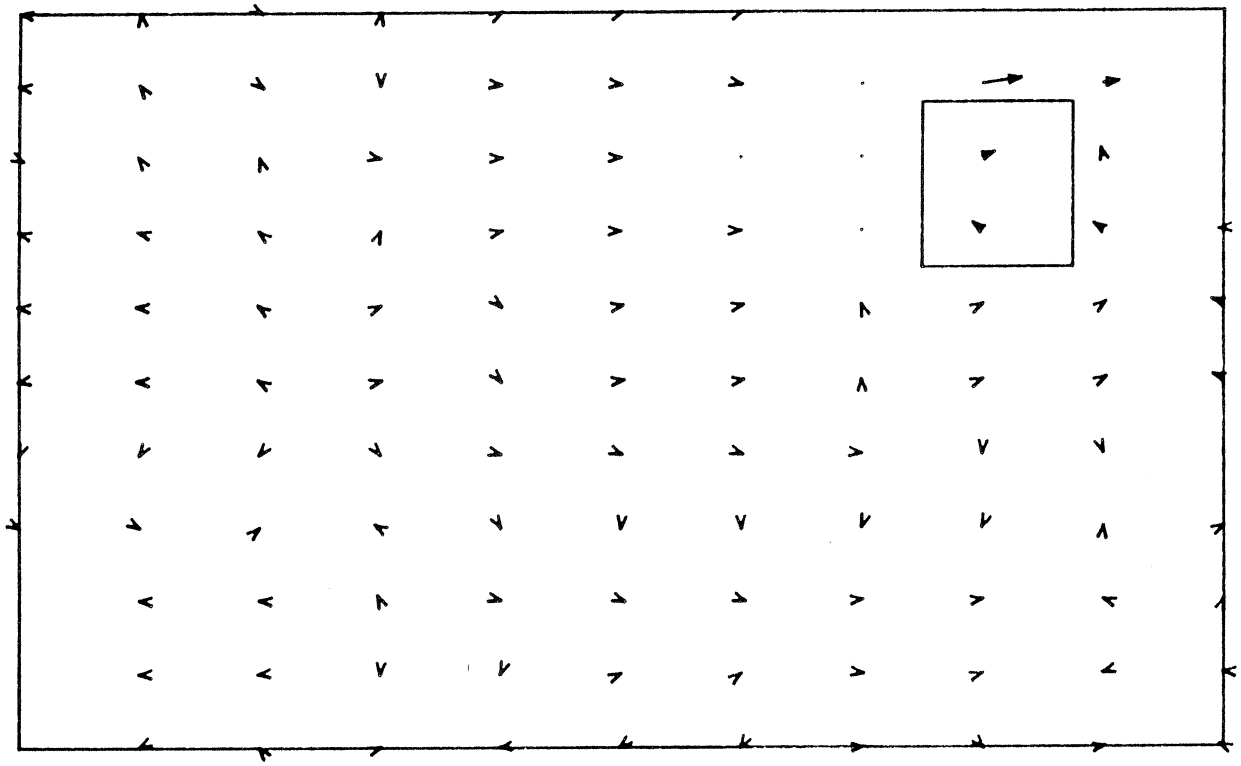
Reward on landing in target	10.0
Move cost as a function of length	$-0.3l - 3l^2$

At the start, the policy is to stay still everywhere, and the estimated value function is everywhere 0. The agent's initial performance, therefore, takes the form of a random walk in the state space, each step being just the deviation vector $\mathbf{b}_t$. The program displays the path of the agent in the state space with animated graphics. A printout of the screen showing an example of the early random performance is overleaf. It is followed by plots of the policy and value function after 3, 10, 100, and 1000 successes, a 'success' being what happens when the agent lands in the target. In the screen printout, the stippled rectangle in the top right hand corner of the graphics area is the target; the black disc is the starting position of the agent on that trial, and the sequence of jumps that the agent has made is shown as a line. Note that the agent may jump over the target without landing in it.

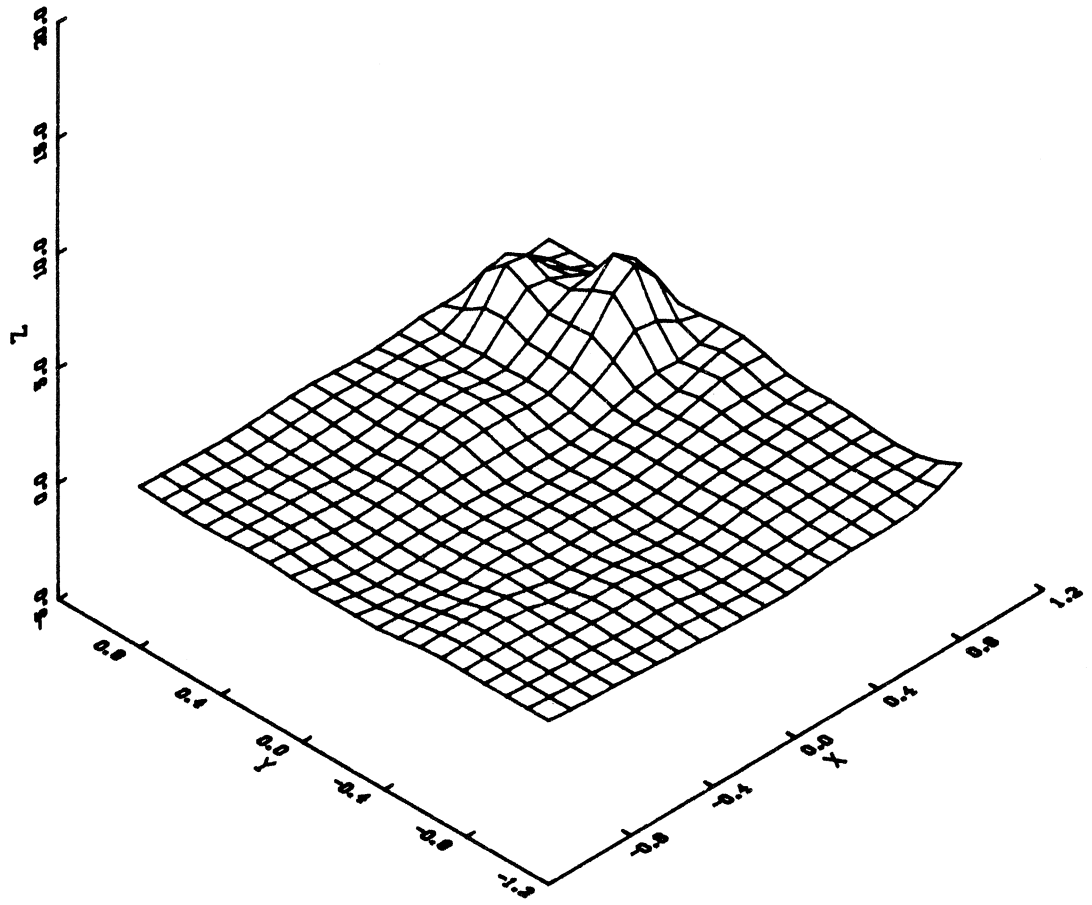
Initial Random Performance



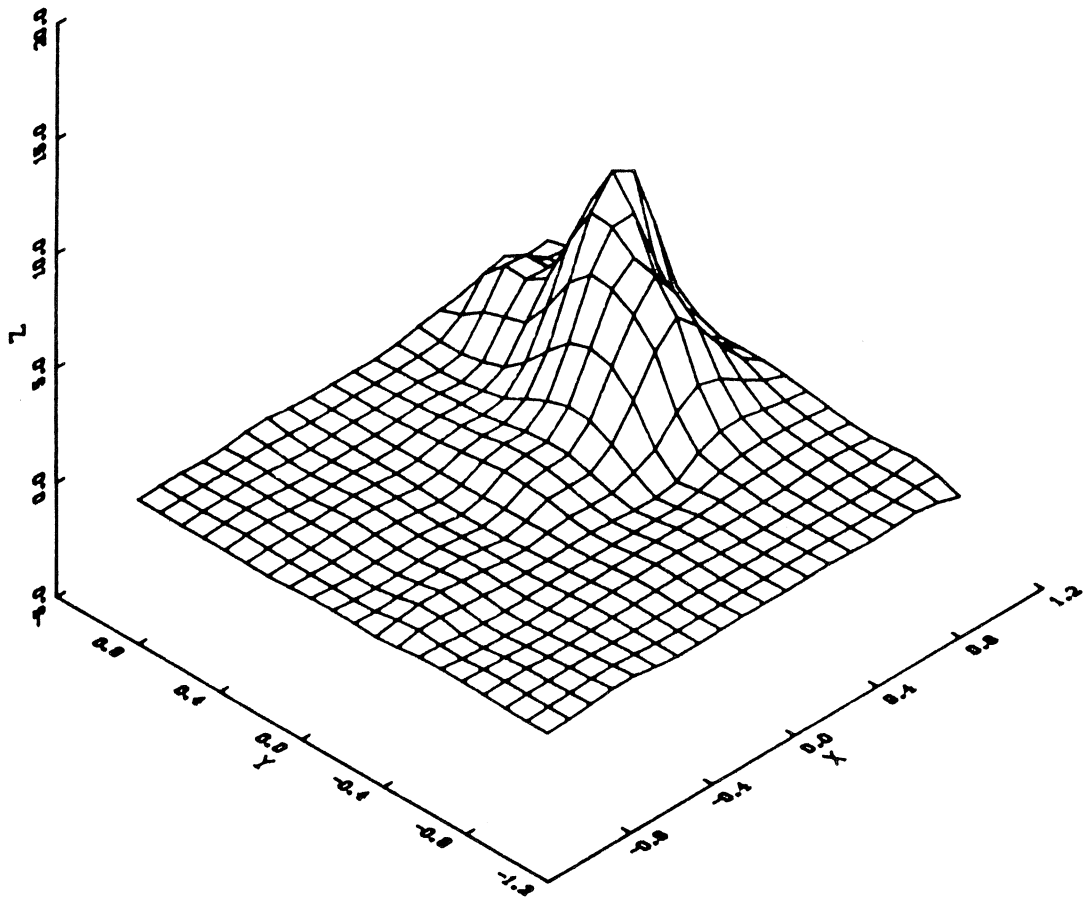
Policy after 3 Successes



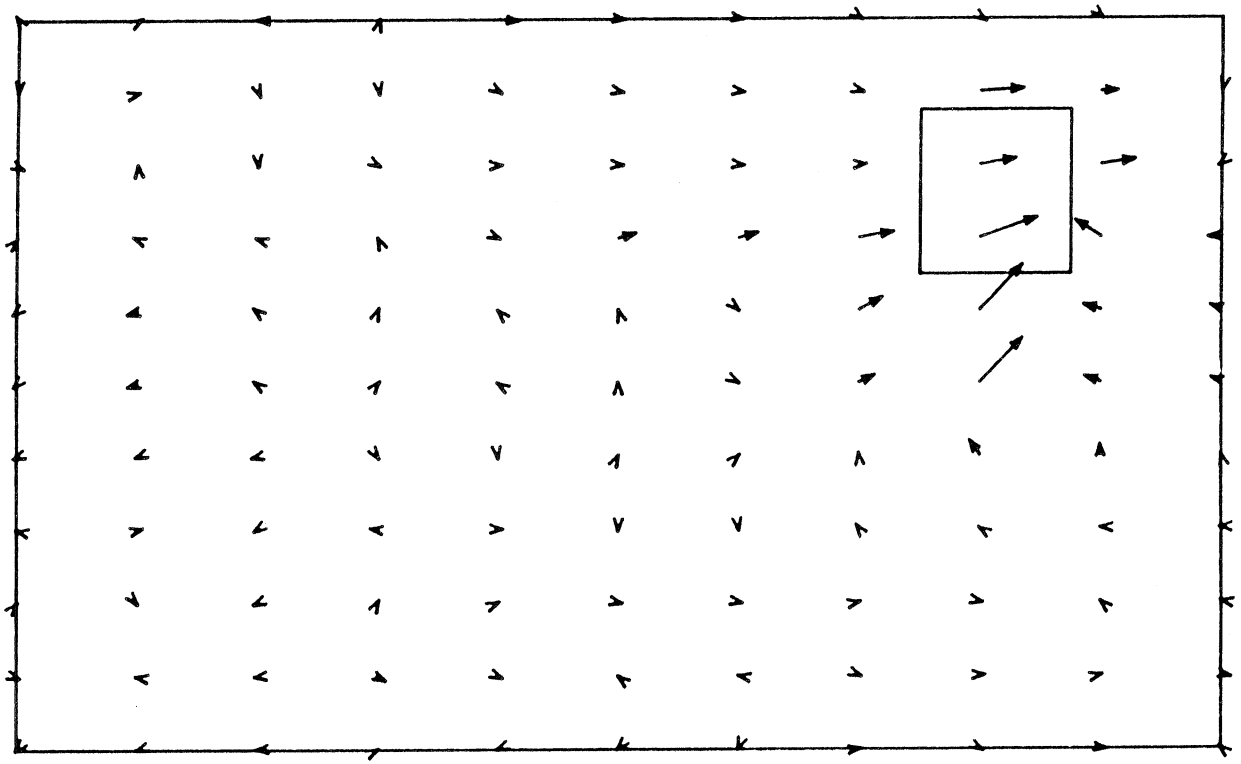
Value Function after 3 Successes



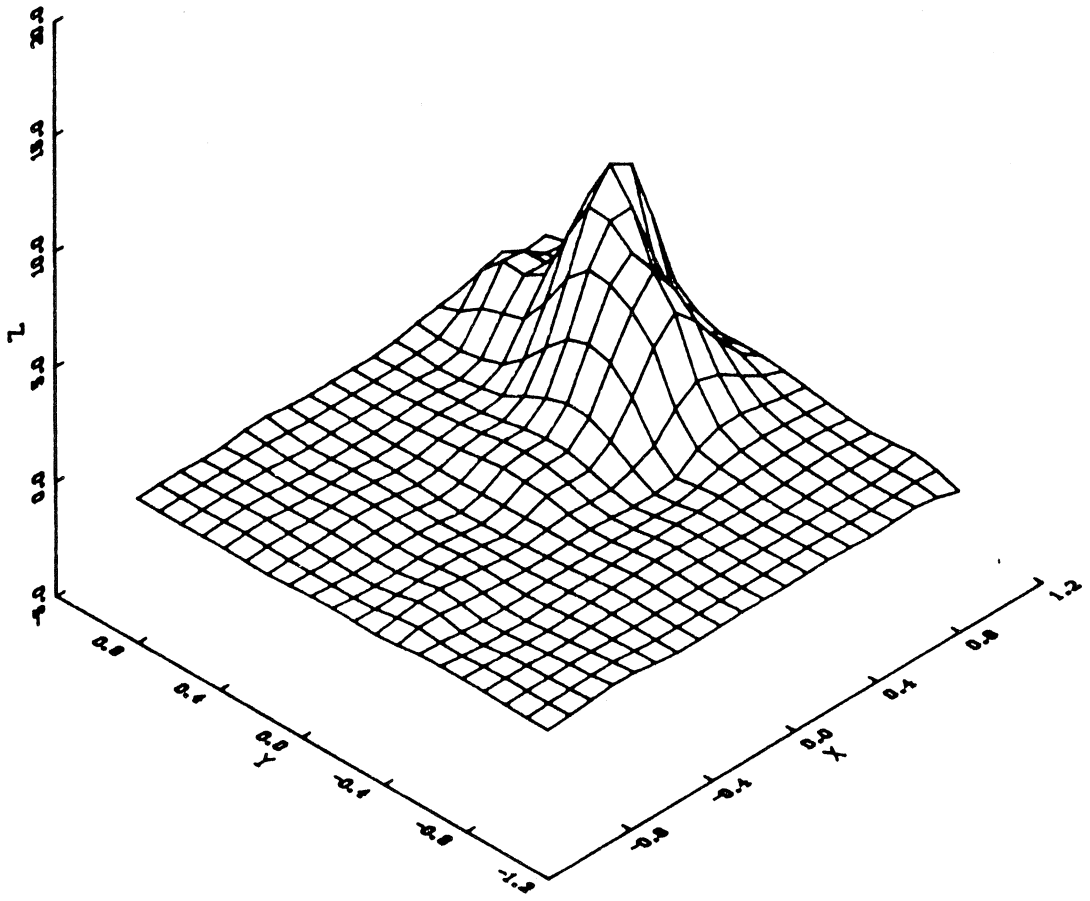
Value Function after 10 Successes



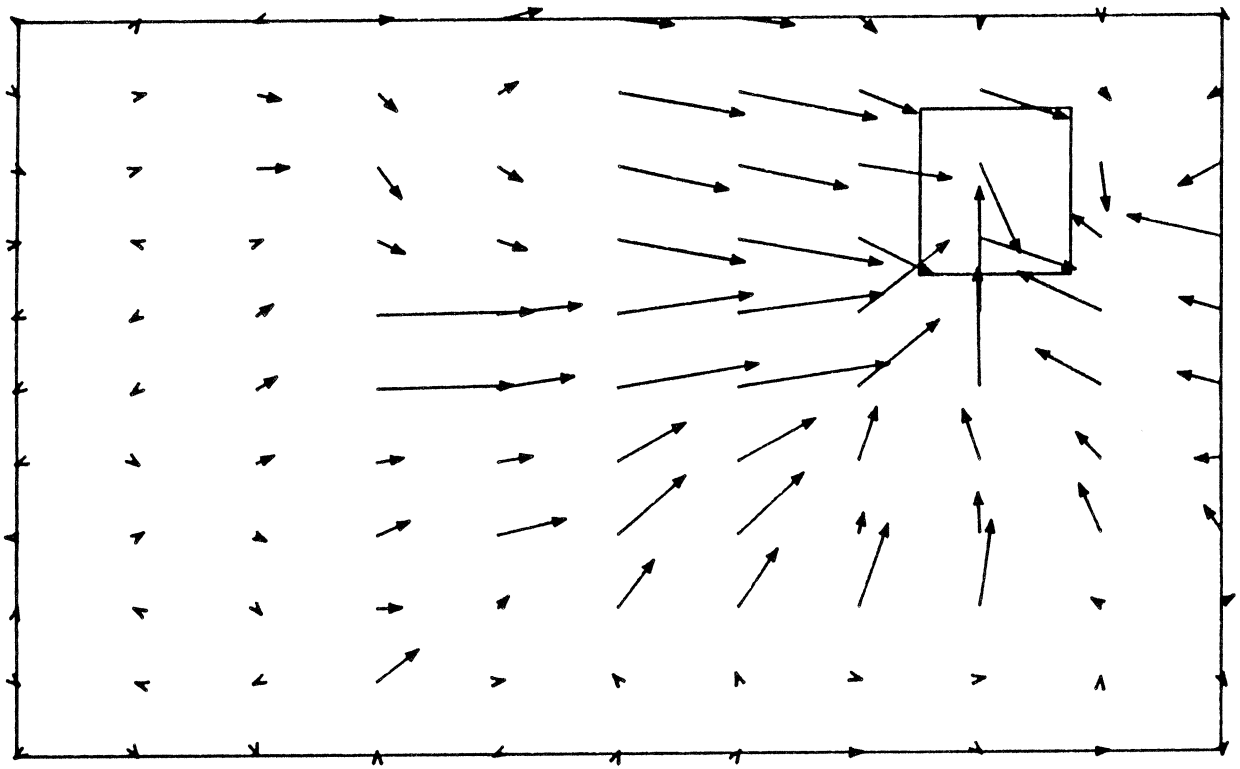
Policy after 10 Successes



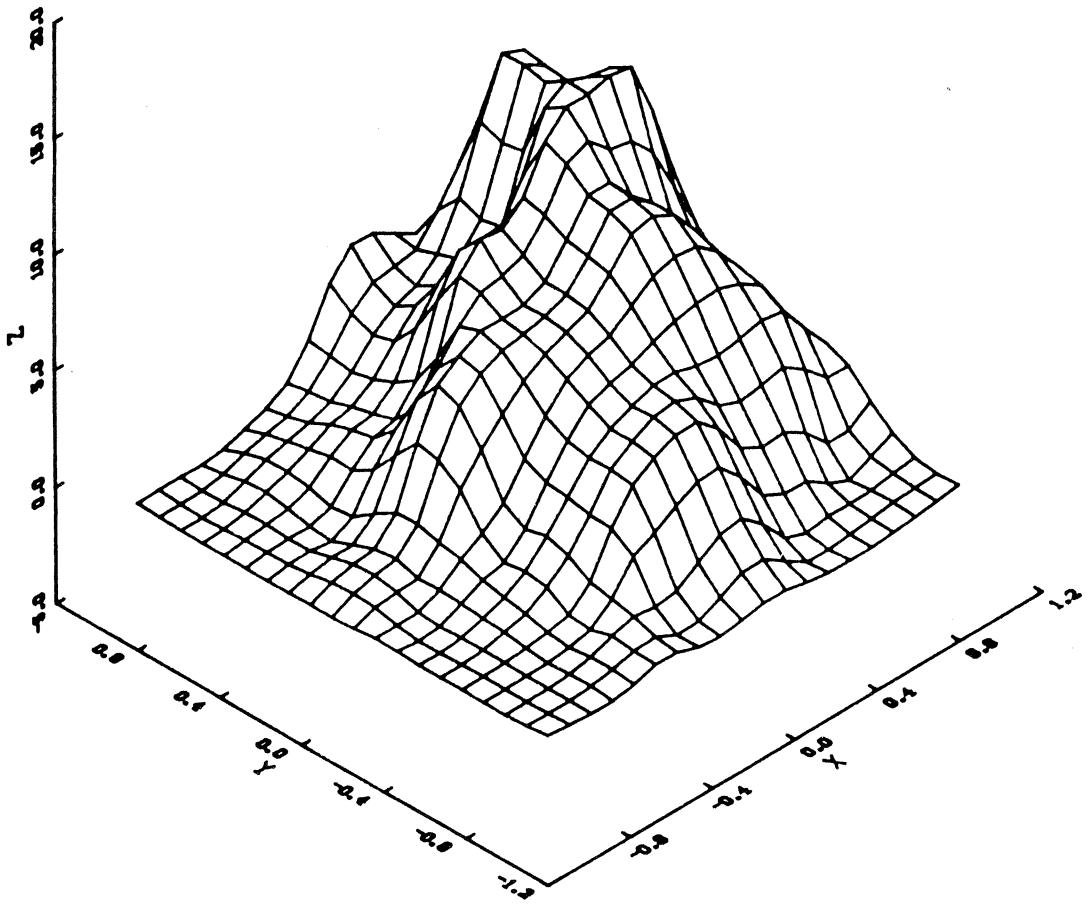
Value Function after 10 Successes



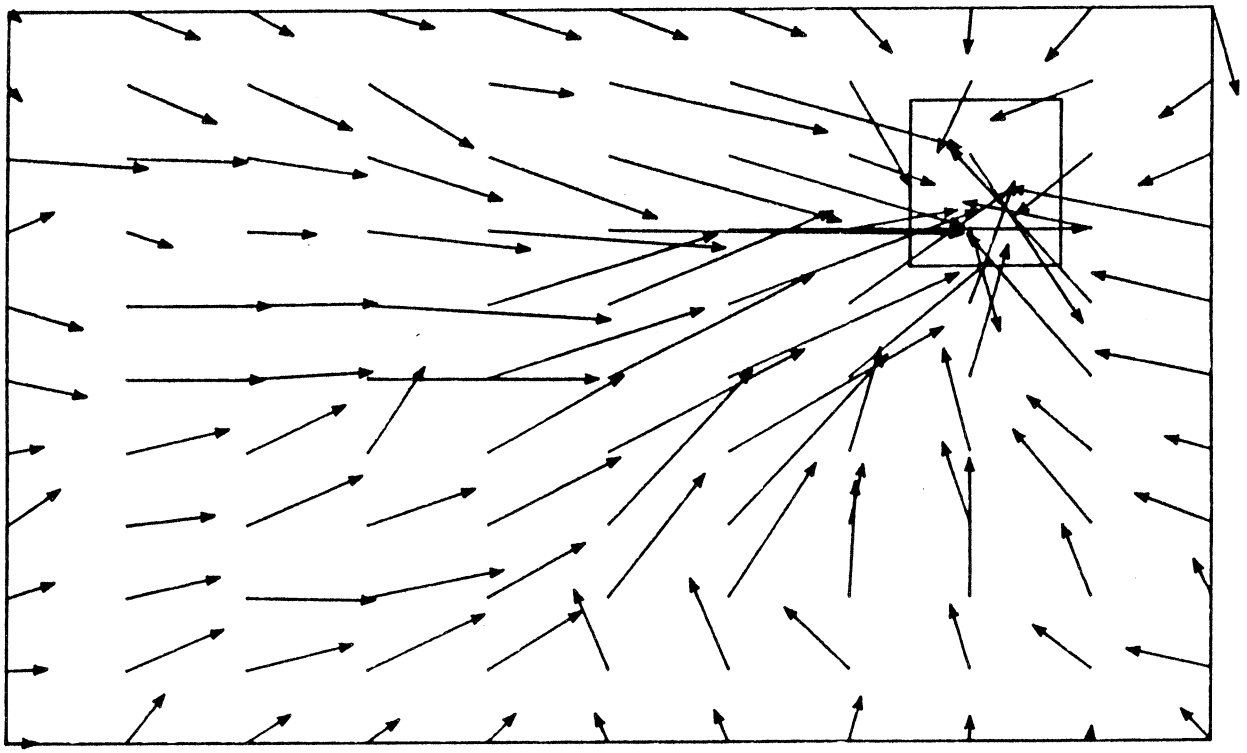
Policy after 100 Successes



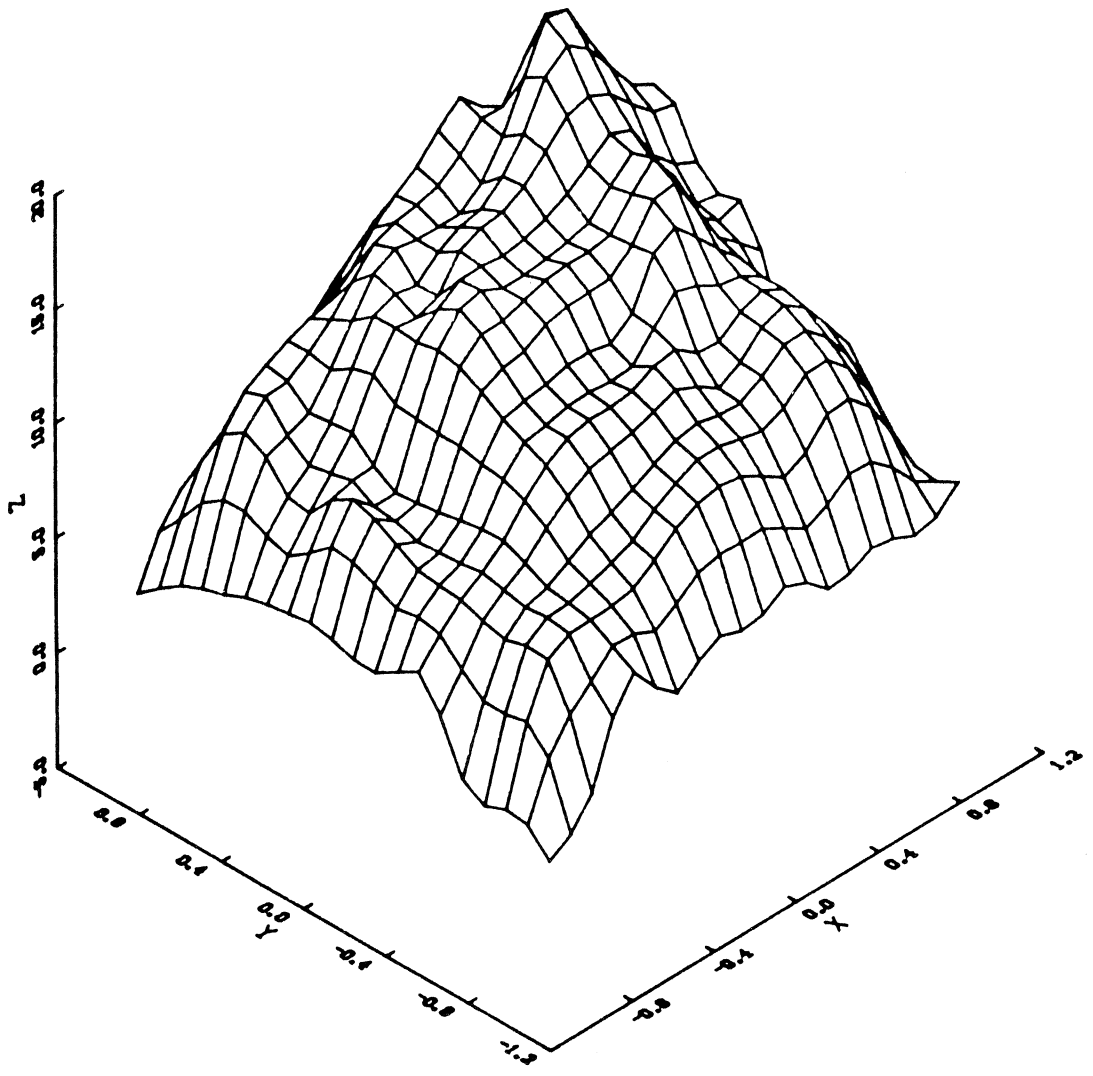
Value Function after 100 Successes



Policy after 1000 Successes



Value Function after 1000 Successes



The course of learning may be described as follows. Since experimental actions are imperfectly separated from policy actions—that is, the rejection factor is not equal to infinity—the agent's estimation policy is in effect stochastic. It therefore finds that it accumulates costs during its initial random walk, and in those parts of the state space that it has visited during its random walk, the value function is reduced below zero, and slight modifications may be made to the policy.

Eventually, the agent lands in the target. On the next move, it experiences a large reward, and it lands in a randomly chosen point in the state space. The initial effect is that a peak in U develops over the target. The first actions that receive rewards are those taken in the target, but because all actions in the target have the same effect, these actions are reinforced equally in all directions (on average). However, once a peak in U has developed over the target, actions that lead the agent to 'climb the sides' of this peak will be reinforced, so that the actions now turn in the direction of the gradient of U. The U 'mountain' gradually spreads across the state-space, and the policy adjustments shift the arrows to point up its slopes.

After 100 successes (a 'success' being what occurs when the agent lands in the target), the hill of U has spread across most of the state space. An interesting phenomenon is visible at this stage. The estimated value function as it spreads is not a symmetrical cone with its peak over the target; instead, it is shaped more like a mountain, with spurs radiating out from it. This is no accident of the particular run, but it is something that occurs consistently in every run, often to a more marked extent than is visible in the results shown. The reason that it happens can be seen from the diagram showing the policy after 100 successes: this shows that there are 'well worn paths' leading into the target. These are 'paths' in the state space along with the policy leads the agent

to the target relatively efficiently; because the policy is relatively efficient on the paths, the value of U on the paths is relatively high. The policy adjustment method tends to direct the policy arrows towards regions of state space with relatively high estimated value. The result is that the policy tends to change so as to direct the agent towards the existing well-worn paths: the effect of this is to cause the agent to travel along the well worn paths still more often, so that the policy on the well worn paths becomes yet more efficient.

After 1000 successes, the policy is to move towards the target from all parts of the state space. I then introduced an 'obstacle' into the problem. The obstacle consisted of a rectangle in the middle of the state space, with the property that if the agent was in the obstacle, then the lengths of its moves were reduced by a factor of 10, but the cost of a move remained the same. That is, if the agent was in the obstacle and chose to perform an action that would normally cause it to travel a distance of 1 unit, then it would pay the cost of travelling 1 unit, while in fact travelling only 0.1 units. This penalty applies only to moves that start when the agent is in the obstacle: the agent can jump over the obstacle with no penalty. The obstacle has two effects on the problem: first, it costs the agent more to travel through the obstacle because each move inside the obstacle carries a cost penalty, and second, if the agent enters the obstacle, it may need more time steps to reach the target, so that the target reward is discounted more. Once the obstacle has been introduced, therefore, the agent should alter its policy so as to either travel round it or else to jump over it.

Note that the agent cannot 'sense' that it is in an obstacle: all it knows is its position in the state space. Nor does the agent notice that it travels a shorter distance than before when it is in the obstacle. All it notices is the difference in the estimated return from actions.

Plots of the policy and value function 500 and 5000 successes after the introduction of the obstacle are overleaf.

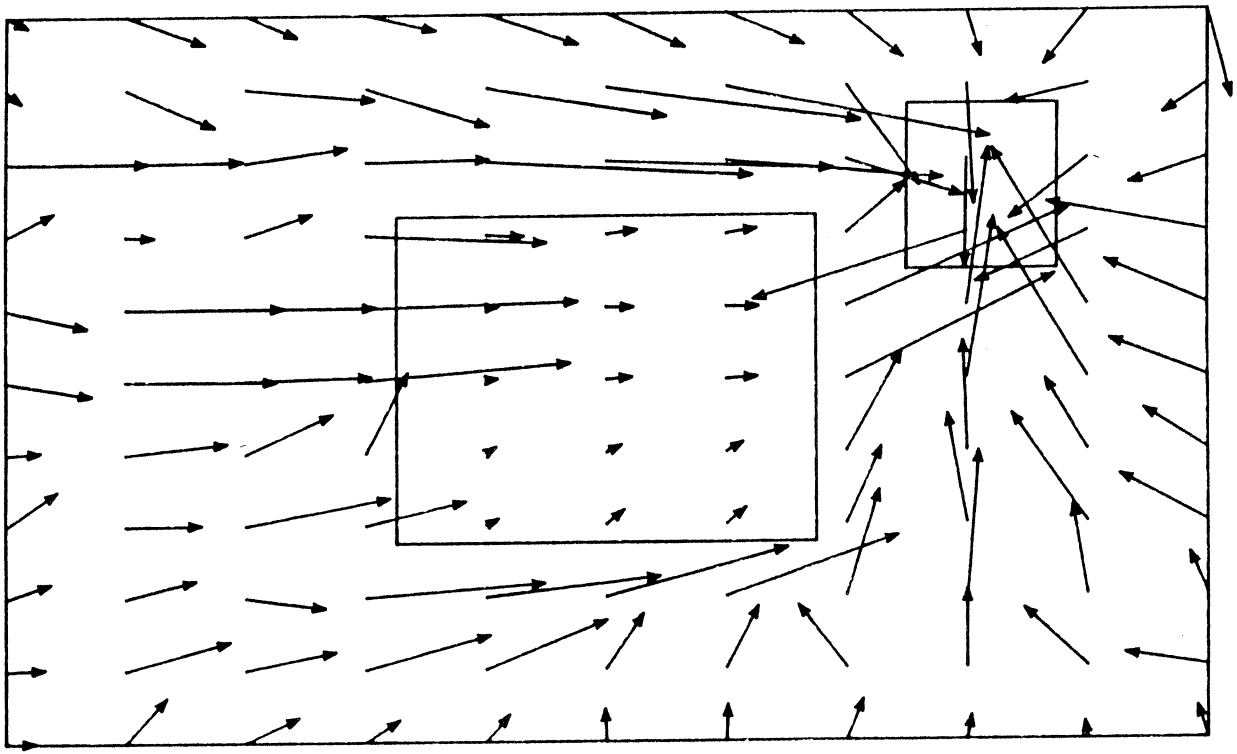
500 successes after the introduction of the obstacle, the value function in the region of the obstacle is greatly reduced, but the policy is not yet much changed. For instance, note that if the agent starts to the left of the obstacle, it still travels straight into it. The estimated value function for states to the left of the obstacle is, as a result, negative. The value function for the extreme bottom left hand corner still appears high, possibly because it has not yet been visited often enough to be reduced to a level appropriate to the changed problem.

But 5000 steps after the introduction of the obstacle, the estimated value of states at the bottom left has recovered somewhat, as the policy now leads the agent around or over the obstacle from virtually all points in the state-space outside the obstacle. Within the obstacle itself, however, the value is still low because of the time taken and cost paid in escaping.

One reason that learning proceeds so smoothly in this demonstration is that the agent is repeatedly placed in randomly and uniformly chosen positions in the state space. The agent will, therefore, gain experience of all parts of the state space eventually. It will, of course, visit those parts of the state space near to the target more often than it visits the edges of the state space, because the agent travels to the vicinity of the target on every trial.

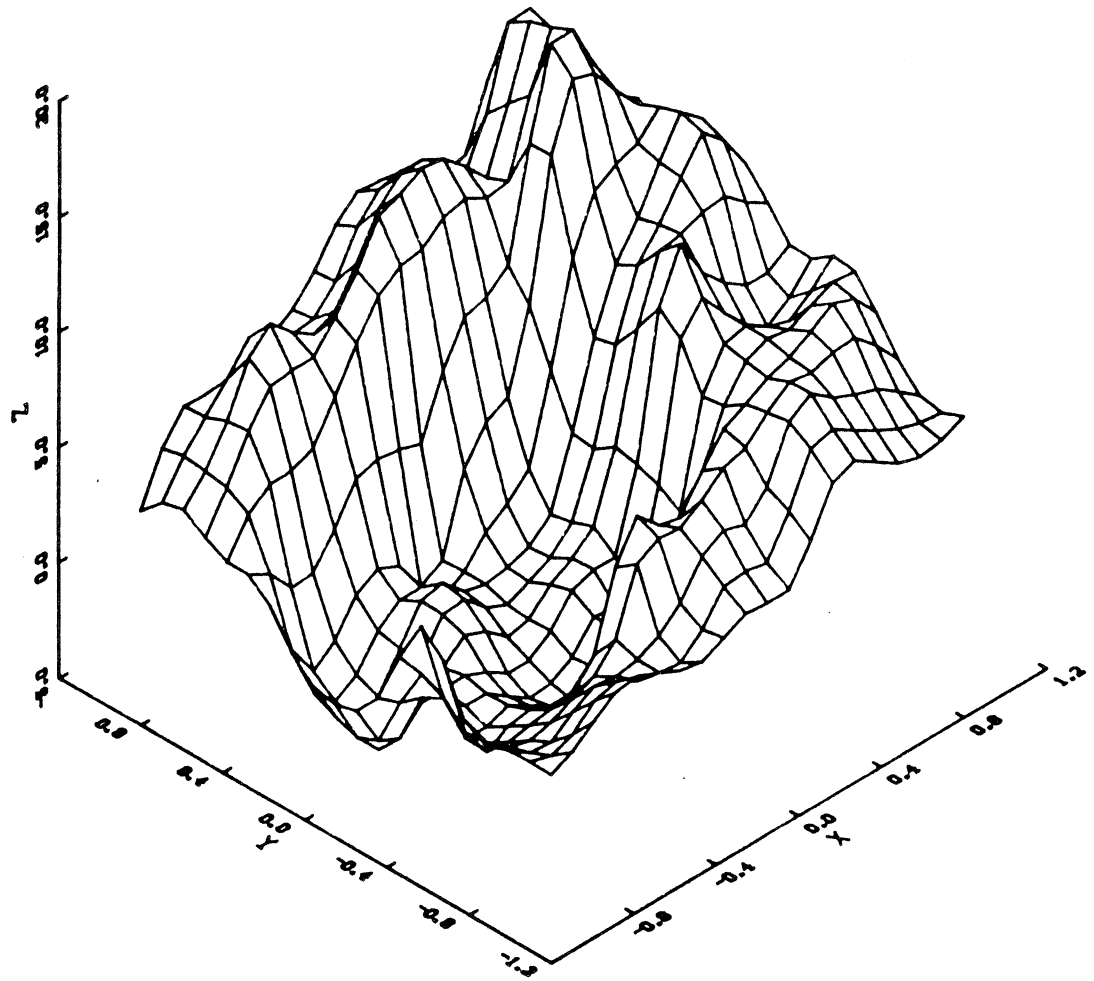
Policy 500 Successes after

Introducing Obstacle



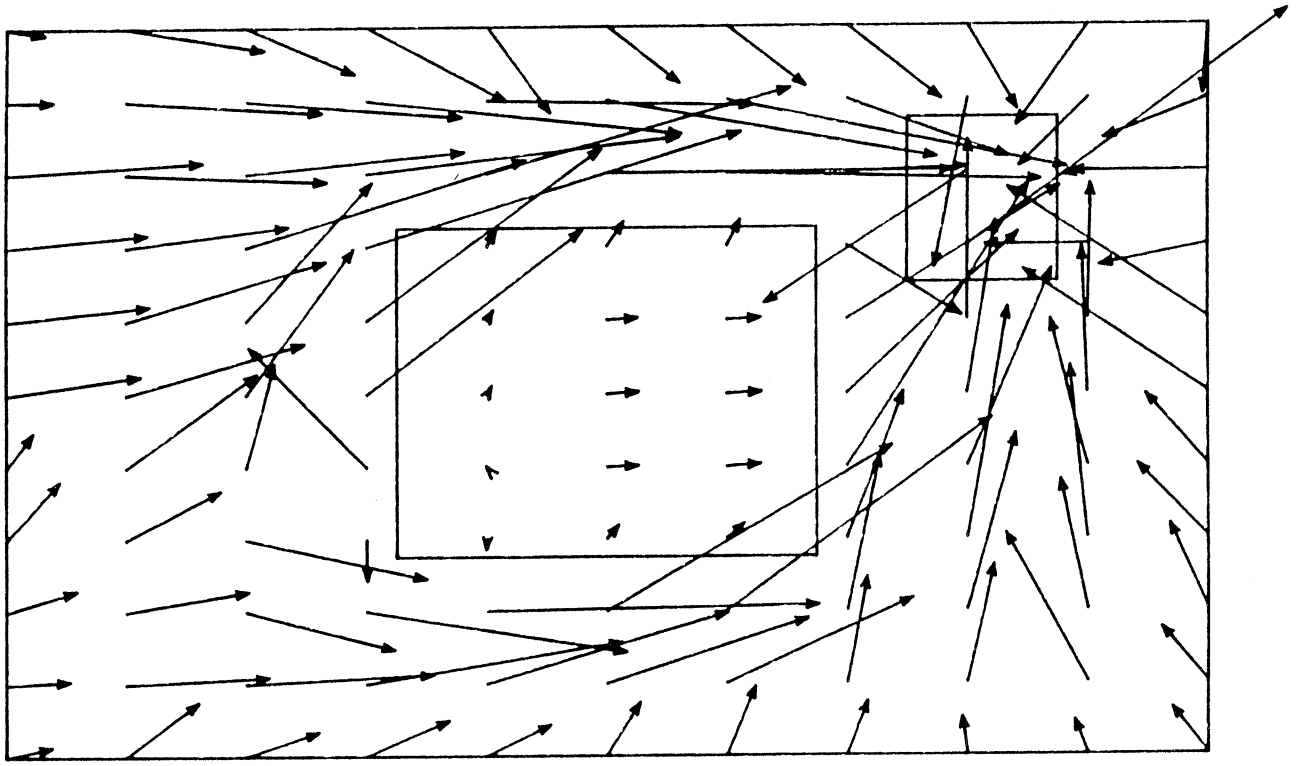
Value Function 500 Successes after

Introducing Obstacle



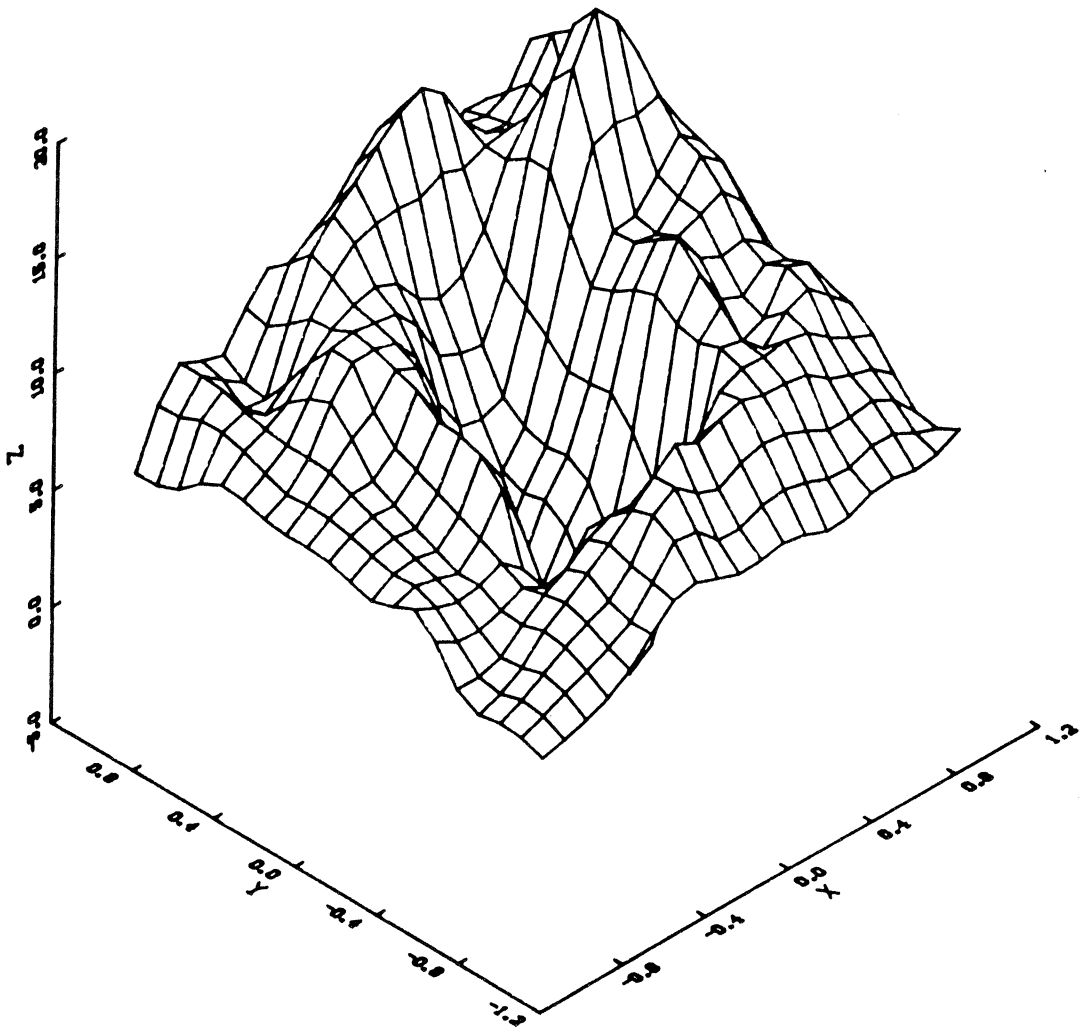
Policy 5000 Successes after

Introducing Obstacle



Value Function 5000 Successes after

Introducing Obstacle



4. A Learning Problem Analogous to Conditioning

The second computational demonstration is analogous to a simple form of conditioning. The problem chosen is that at each time step an agent may choose either to perform or not to perform a certain action (such as 'pressing a lever' or 'pecking a key'). If the agent performs the action, there is a small cost; if the agent does not perform it, the cost is zero. This is a reasonable assumption, since performing a response such as a lever press or a peck will cost an animal more energy than sitting still. I will call the action 'pecking'. Occasionally, the agent may receive a large reward, analogous to the delivery of food to a hungry animal in a Skinner box.

To determine the optimal policy in a Skinner box with no prior knowledge of the reinforcement schedule is not a trivial problem. This demonstration shows the operation of a Q-learning algorithm faced with a range of different reinforcement schedules. The following schedules of reinforcement are implemented.

Fixed Interval (FI)

At any time the reward is either *available* or *unavailable*. When the reward is available, the agent receives the reward only if it pecks, and when the reward is unavailable, the agent does not receive a reward whether it pecks or not. If a reward is made available at the tth time step, the agent will receive it if it pecks at time t.

The fixed interval schedule may be described as follows: for the $n-1$ steps following each reward the agent receives, the reward is unavailable. On the nth step, the reward is made available again, and the reward remains available until the agent pecks, and so receives it; the cycle then begins again. The agent may, therefore, receive a reward on at most one in

n time steps.

In this and subsequent schedules, the agent cannot perceive whether the reward is available or not, and there is no perceptible change other than the passage of time to show that a reward has become available.

Variable Interval (VI)

If the agent receives a reward at time t, the reward becomes unavailable. On each subsequent step, from time $t+1$ onwards, if the reward is still unavailable, then it is made available with probability $\frac{1}{n}$; if the reward has already become available, then it stays available until the agent pecks and receives it. The effect of this schedule is that the lengths of the times during which the rewards are unavailable are exponentially distributed, and if the agent pecked on every time step, the mean interval between rewards would be n.

Classical Fixed Interval (CFI)

The agent receives a reward on every nth step, whatever it does.

Classical Variable Interval (CVI)

On each step, the agent receives a reward with probability $\frac{1}{n}$ whatever it does, and independently of what happened on previous steps.

Fixed Ratio (FR)

The agent receives a reward on every nth peck, regardless of how much time has passed.

These are the *objective* problems that the agent may face. What remains to be done is to define the agent's *subjective* problem.

What I am going to do is to construct a simple and straightforward implementation of Q-learning for conditioning. I do not want to claim that the implementation that follows is a realistic model of animal learning in conditioning experiments: animal learning is likely to be considerably more sophisticated than the algorithm presented here.

The Subjective State Space

It is reasonable to suppose that, since rewards come at infrequent intervals, receiving a reward is a salient event for the agent. One dimension of the agent's state space, therefore, is a measure of the elapsed time since the last reward. The only other information the agent has is the history of its pecks, and the second dimension of the state-space is a measure of the number of pecks since the last reward.

An intuitively reasonable restriction is that the region of state space the agent may 'visit' is bounded, since one would not expect an agent to distinguish an infinite range of states.

Let the agent's measure of elapsed time since the last reward be m. At time t, the value of m is m_t . If the agent receives a reward at time t, then on the time step t+1 immediately following, m_{t+1} is zero. On each subsequent time step until the next reward, m is increased, but it never gets larger than 1:

$$m_{t+1} = 1 - 0.95\ (1 - m_t)$$

if the agent does not receive the reward at time t. That is, the difference between m and 1 is reduced by a factor of 0.95 at each time step, until the agent receives a reward, in which case m is reset to zero.

Let p be the measure of the amount of pecking since the last reward. p is calculated in a similar way to m:

$$p_{t+1} = \begin{cases} 1-0.95(1-p_t) & \text{if the agent pecks at time } t \\ p_t & \text{otherwise} \end{cases}$$

Hence the agent's subjective state space can be represented as a triangle, with vertices at (0,0), (1,0), and (1,1); the time axis is horizontal, and the peck axis is vertical. Each time the agent receives a reward, it returns to the origin, so that if a reward is received at time t, (0,0) is the state at time t+1. If the last reward was at time t, and the time is now t+i, and the agent has performed j pecks since the last reward, then the state is

$$m_{t+i} = 1-0.95^{i-1}$$

$$p_{t+i} = 1-0.95^{j}$$

In the program, I have also provided that the state changes may be subject to small random perturbations. On each state transition, both m and of p may have small random quantities added to them: the random quantities are drawn independently from uniform distributions of zero mean, and the widths of the uniform distributions may be set as required for m and for p separately. The

effect of this that m and p accumulate random error, and this reduces the amount of information the agent retains about the past, since the agent's 'memory' is its current position in the state space. Adding small random perturbations ensures that the accuracy of this memory declines gradually with time.

For this problem, I have used the method of learning action values, both to demonstrate it, and because the considerations in appendix 1 suggest that one-step Q-learning should be reliable. For each action, the action-value function Q is approximated over the state space using a CMAC, just as the value function was approximated in the previous demonstration.

5. The Learning Method

The method of one-step Q-learning was used, as described in section 4 of chapter 7. The updating rule is

$$Q_{t+1} = \textbf{adjust}(\, Q_t, \langle x_t , a_t \rangle, \alpha, r_t + \gamma\, U_t^Q(x_{t+1}, a_t) \,)$$

where $\langle x_t , a_t \rangle$ is the state-action pair for which Q_t is adjusted. and α is a learning factor. Only the action values are represented, and the estimated value of a state is taken to be the maximum of the estimated action values That is,

$$U^Q(x) = \max_a \{\, Q(x,a) \,\}$$

where $U^Q(x)$ is the estimated value of the state x. Taking the maximum of estimates as an estimate of the maximum is dubious statistical practice, but the argument in Appendix 1 shows that this method of estimating the value can be used in a convergent learning algorithm, although it is likely that a less biased estimator of the maximum of the action values would give better performance. But in this problem there are only two possible actions, so the bias is minimal.

In addition, results are presented for two demonstrations in which the learning rule is

$$Q_{t+M+1} = \textbf{adjust}(\, Q_{t+M} \,,\, x_t \,,\, \alpha \,,\, \mathbf{r}(t) \,)$$

with $\mathbf{r}(t)$ defined as in the previous demonstration, and with the learning period M equal to 3. In calculating $\mathbf{r}(t)$, λ_t was calculated according to

$$\lambda_t = \exp[\, -\eta(U_t^Q(x_t) - Q_t(x_t, a_t)\,)\,]$$

where η is the rejection factor, as before.

6. Behavioural Policy and Occupancy

The agent chooses its actions as follows. At each time t, the agent computes the estimated action values of pecking and of not pecking; then with probability $1-\pi_t$ it chooses the action with higher estimated value, and with probability π_t it 'experiments', and chooses the action with lower estimated value. The probability of experimenting—π_t—depends on the number of times that the agent has previously visited the current region of state space. If the agent has visited the current region many times, the probability of experimenting will be low; if the region near x_t is relatively unexplored, the probability of experiment will be high.

The program keeps track of the number of visits to each part of the state space with a CMAC function Y—the *occupancy*—which is incremented by a fixed amount at the current state on each time step. That is, Y is initially zero everywhere, and at each time step it is adjusted

$$Y_{t+1} = \textbf{adjust}(\, Y_t \,,\, x_t \,,\, 1 \,,\, Y_t(x_t) + c \,)$$

where c is an amount kept constant during each run. That is, c is added to the value of Y stored for each CMAC patch containing x_t, so that the value of $Y_t(x)$ will be proportional to the number of visits to points in state space near x

prior to time t.

The occupancy Y is used to control both the probability of experiment π_t and the learning factor α_t. Intuitively, it is reasonable to arrange that both π and α should be larger if the current state is in a little-explored part of the state space, in well-explored parts of state-space, π and α should be small. In the program, this is arranged by calculating π and α at each time step according to

$$\pi_t = \frac{T}{T + Y_t(x_t)} \pi_0$$

and

$$\alpha_t = \frac{T}{T + Y_t(x_t)} \alpha_0$$

where π_0 and α_0 are parameters that set the initial values of π and α, and T is a positive number called the *rate parameter* which determines how rapidly π and α decline with increasing occupancy. If T is small, π and α will decline rapidly with increasing occupancy; if T is large, they will decline more slowly. T, π_0, and α_0 are parameters that are passed to the program at the start of a run.

Note that the probability of experiment does not depend on the difference between the estimated action values for the current state. Better methods of setting π might take into account both the current occupancy and the difference in action values.

7. Choice of Parameters

In the demonstrations, I have selected the following parameter values as the 'default' values; these are the values the parameters have unless it is specifically stated otherwise.

The rewards and costs are:

Rewards and Costs

Cost of a peck in FI, CFI, VI, CVI schedules	-0.5
Cost of a peck in FR schedule	-0.25
Cost of doing nothing	0
Reward	10.0
Max. reward frequency	1 in 10

That is, rewards are once every 10 time steps in the FI schedule; the reward is set with probability 0.1 in the VI schedule; rewards are given every 10 time steps in the CFI schedule; the chance of receiving a reward is 1 in 10 in the CVI schedule; and 10 pecks are needed to obtain a reward in the FR schedule. It is reasonable for the default peck cost to be less in the FR schedule because 10 pecks are required to obtain a reward in this schedule, whereas in all the other schedules a reward can be had for just one peck, or for no pecks at all.

Unless otherwise stated, the parameter values used for the learning method are

Learning Parameters

γ	0.95
α_0	0.5
π_0	0.5
T	100
c	0.25
M	1

The values of c, and of the CMAC patch size (below), and of the parameters determining state transitions are arranged so that the highest levels of occupancy (near the origin) are approximately equal to the number of trials.

The parameter values for the state space are:

State-Space Parameters

Time Measure Factor	0.95
Peck Measure Factor	0.95
Span of Added Noise	±0.025

The CMAC parameters are:

CMAC Parameters

Side of a CMAC Patch	0.2
Number of Patches Containing each Point	10

8. Results

All the results presented were obtained by performing 50 runs with the same parameter values but starting with a different random seed in each run. Each run consisted of a number of *trials*: a trial is a sequence of actions starting from the state (0,0), and ending either when the agent receives a reward, or else after 100 time steps, whichever comes soonest. The number of trials on which learning took place in each run was $20T$, so that when T=100, each run had 2000 learning trials. The number of learning trials was made proportional to T in this way so that values of π and α would reach similar levels by the end of runs with different values of the rate parameter T.

In each run, after the $20T$ learning trials, the experiment and learning parameters π_0 and α_0 were set to zero, so that no more learning or experimentation took place. Then the state transitions for each of twenty 'test trials' were recorded. The point of this is that in these test trials, the agent followed the estimate of the optimal policy that it had constructed during the learning trials; the agent adopts its 'subjectively optimal' strategy.

The records of performance in the test trials for each set of parameter values used are presented as graphs of the average cumulative number of pecks plotted against the elapsed time since the previous reward. The average cumulative number of pecks was obtained by averaging the records of all 20 test trials for all 50 runs for each combination of parameter values. The average cumulative number of pecks after n time-steps was calculated by averaging the cumulative number of pecks in all trials that had not yet ended after n time steps; trials that ended before n time steps did not contribute to the average at n time steps.

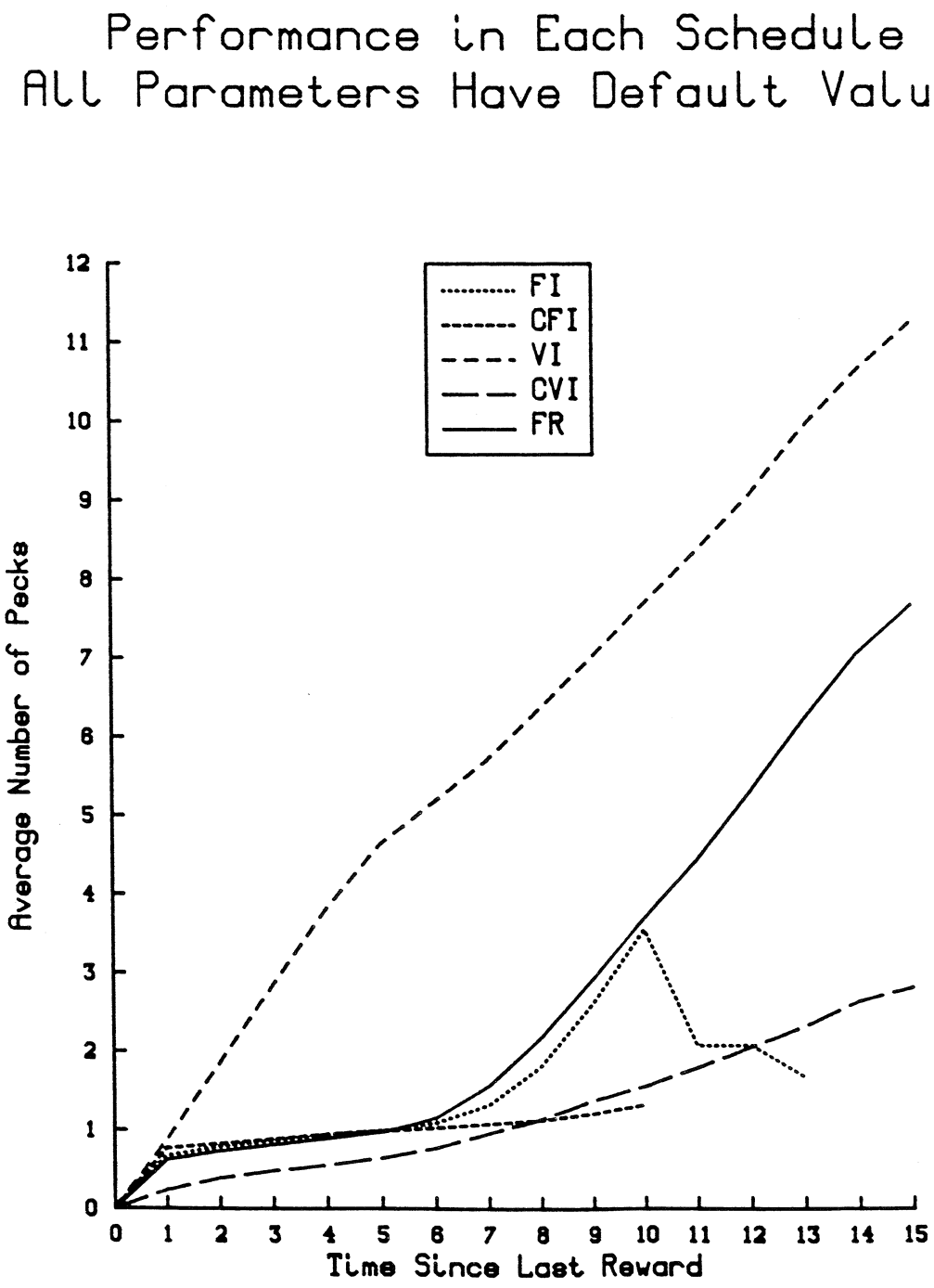
The value function $U^Q(x)$ and the difference between the estimated action values of pecking and of not pecking $Q(x, peck) - Q(x, no\ peck)$ were also

recorded for a grid of points over the state space at the end of the learning trials in each run. These data are presented as contour plots over the state space. Each contour plot depicts the average of 50 surfaces, one from each run with the same parameters.

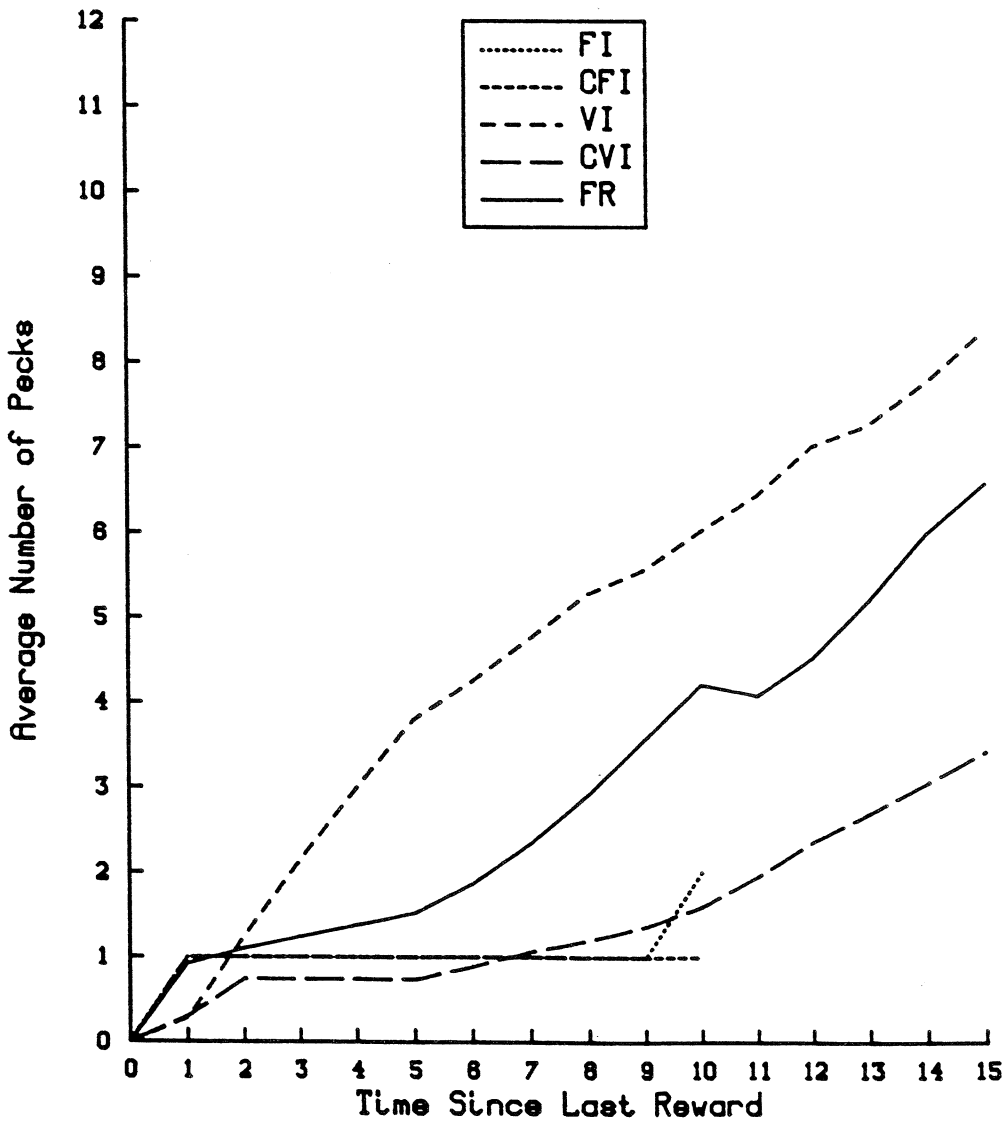
The contour plots of the differences in estimated action values enable one to see trends in the policies found: if the value is negative, then not pecking is the preferred action, whereas if the value is positive, then pecking is preferred. Contour lines for negative values are dotted; those for positive values are solid; this enables areas of state space in which different actions are preferred to be easily distinguished.

Summary of Plots

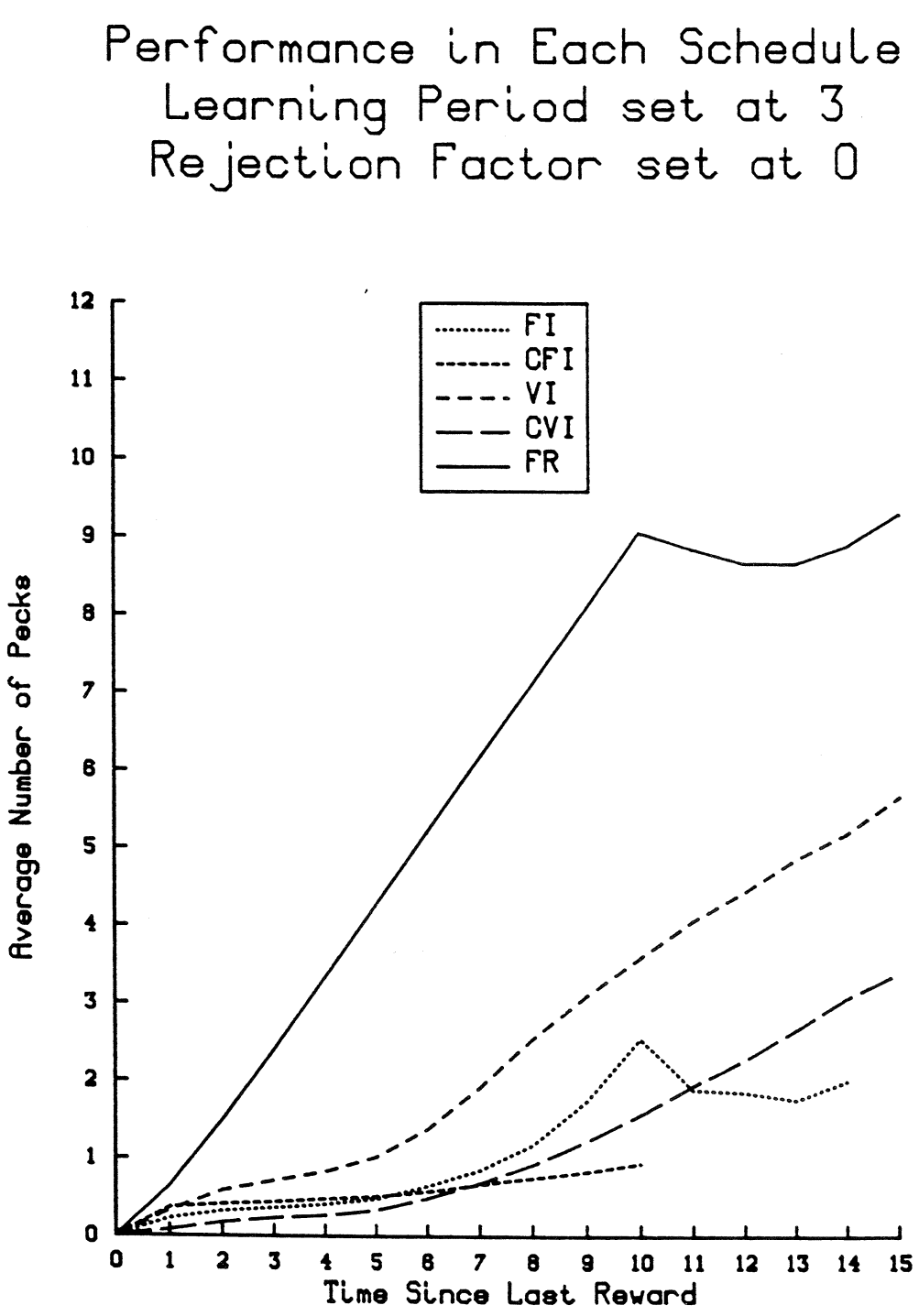
Plot Number	Schedules	Parameters	Plot Type
1	All	All default	Performance
2	All	No noise	Performance
3	All	$M = 3$ $\eta = 0$	Performance
4	All	$M = 3$ $\eta = 20$	Performance
5	FI	$T = 30, 100, 300$	Performance
6	FI	All default	Value Function
7	FI	All default	Q Difference
8	CFI	$T = 30, 100, 300$	Performance
9	CFI	All default	Value Function
10	CFI	All default	Q Difference
11	VI	Not applicable	Expected Return
12	VI	$T = 30, 100, 300$	Performance
13	VI	Peck Cost = -0.125, -0.5, -2	Performance
14	VI	All default	Value Function
15	VI	All default	Q Difference
16	VI	Peck Cost = -2	Value Function
17	VI	Peck Cost = -2	Q Difference
18	CVI	$T = 30, 100, 300$	Performance
19	CVI	All default	Value Function
20	CVI	All default	Q Difference
21	FR	$T = 30, 100, 300$	Performance
22	FR	All default	Value Function
23	FR	All default	Q Difference


Performance in Each Schedule
All Parameters Have Default Values
FI
CFI
VI
CVI
FR
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

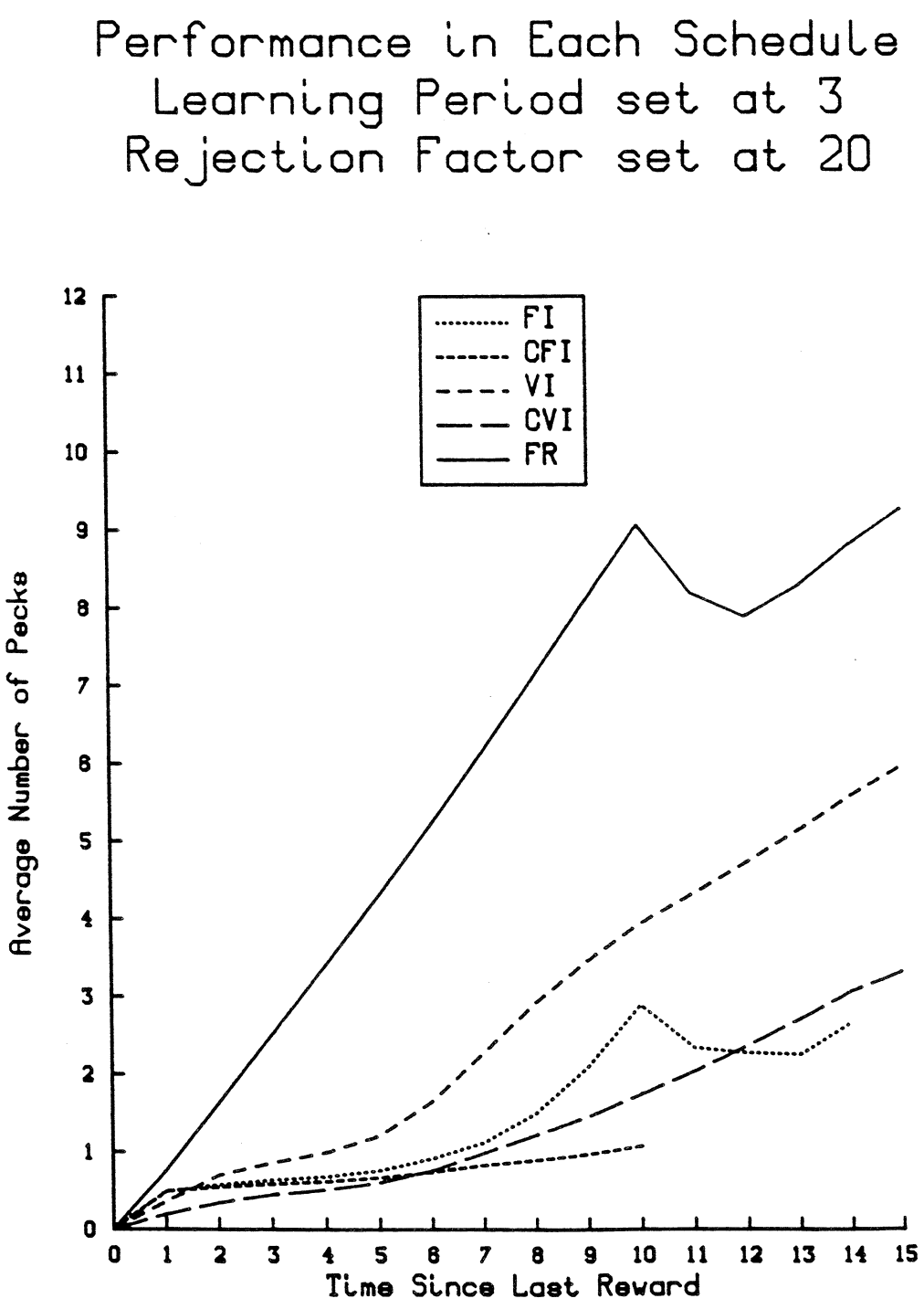
Plot 1


Performance in Each Schedule
No State-Transition Noise
FI
CFI
VI
CVI
FR
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

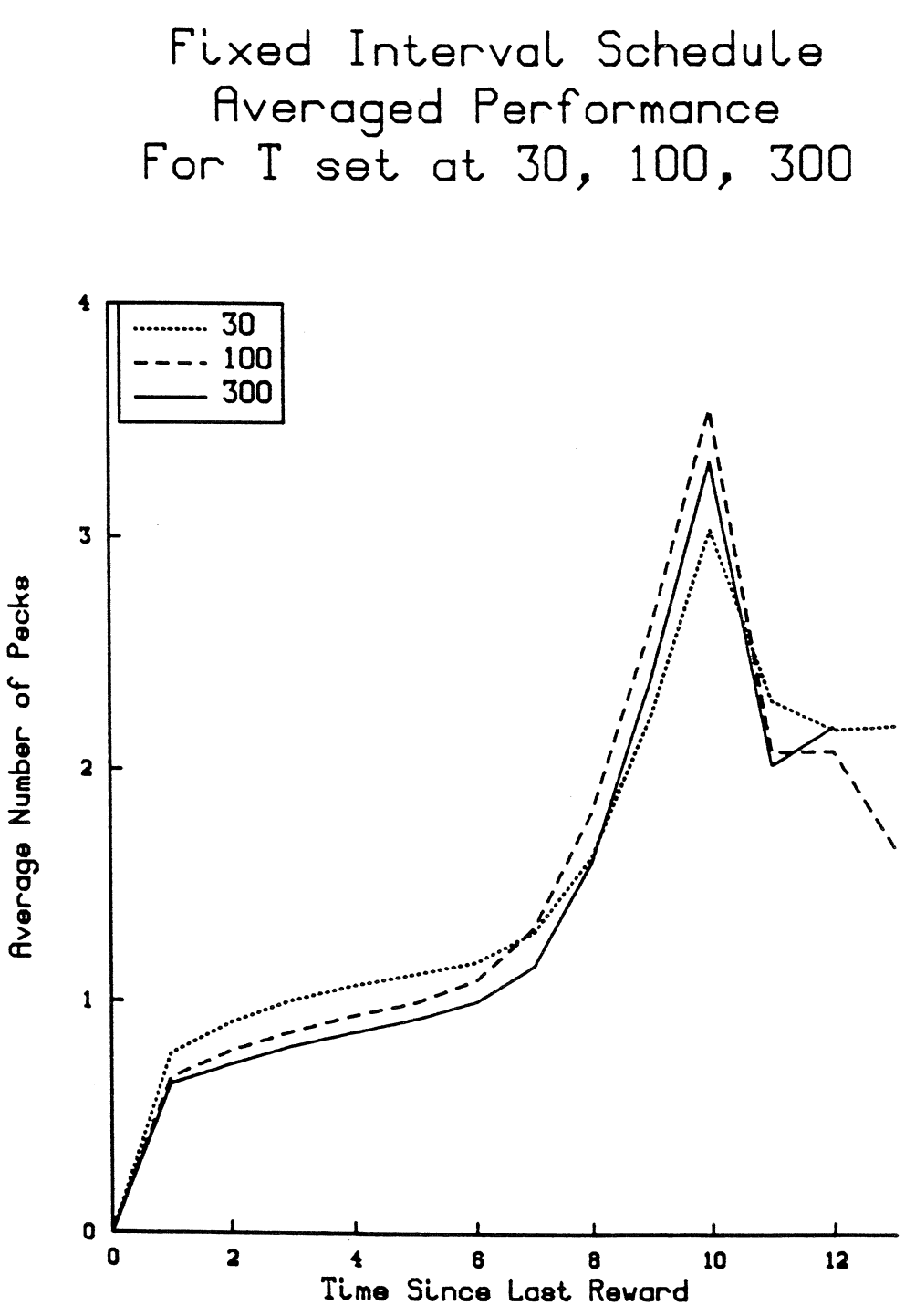
Plot 2


Performance in Each Schedule
Learning Period set at 3
Rejection Factor set at 0
FI
CFI
VI
CVI
FR
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

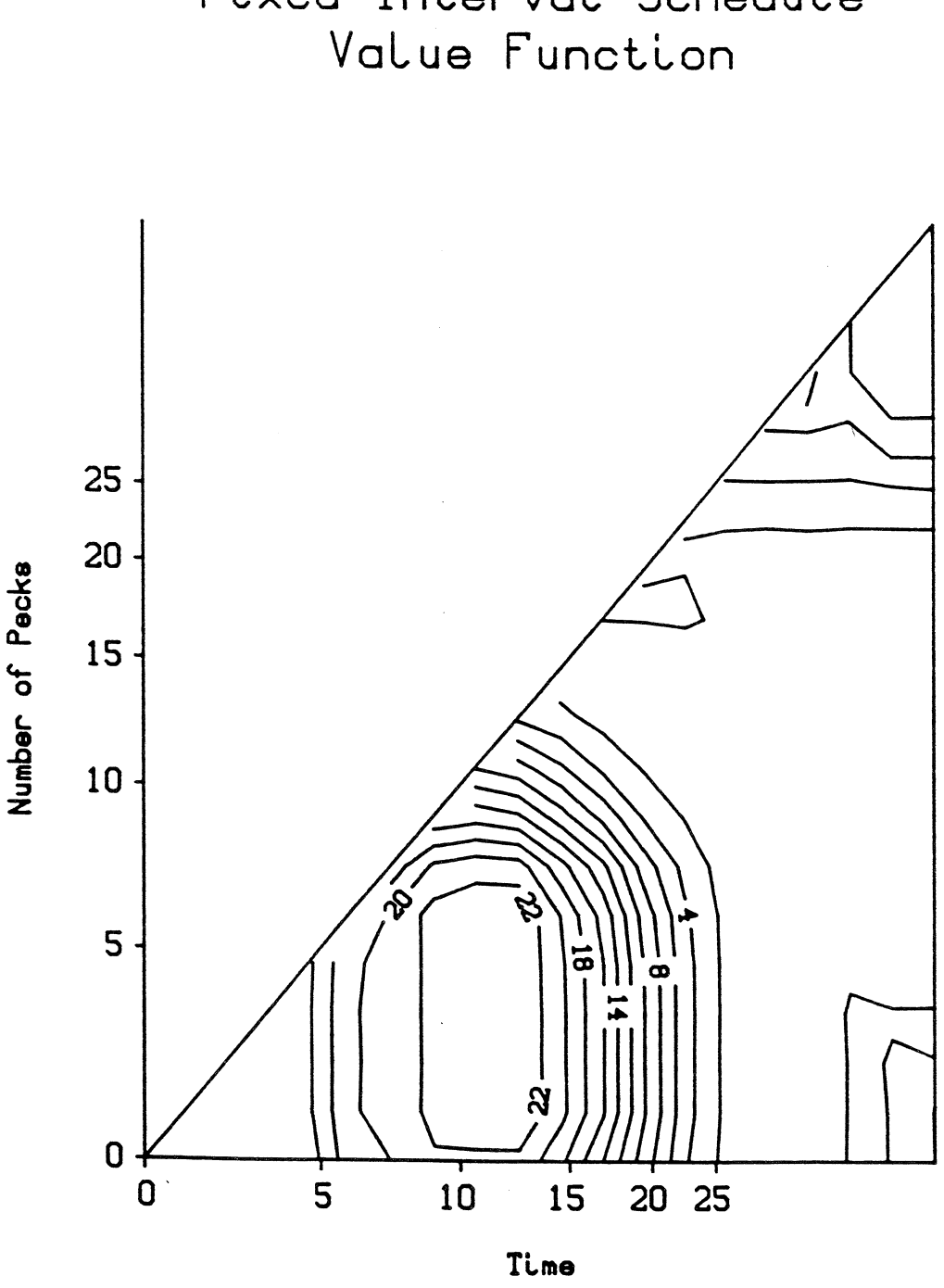
Plot 3


Performance in Each Schedule
Learning Period set at 3
Rejection Factor set at 20
FI
CFI
VI
CVI
FR
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

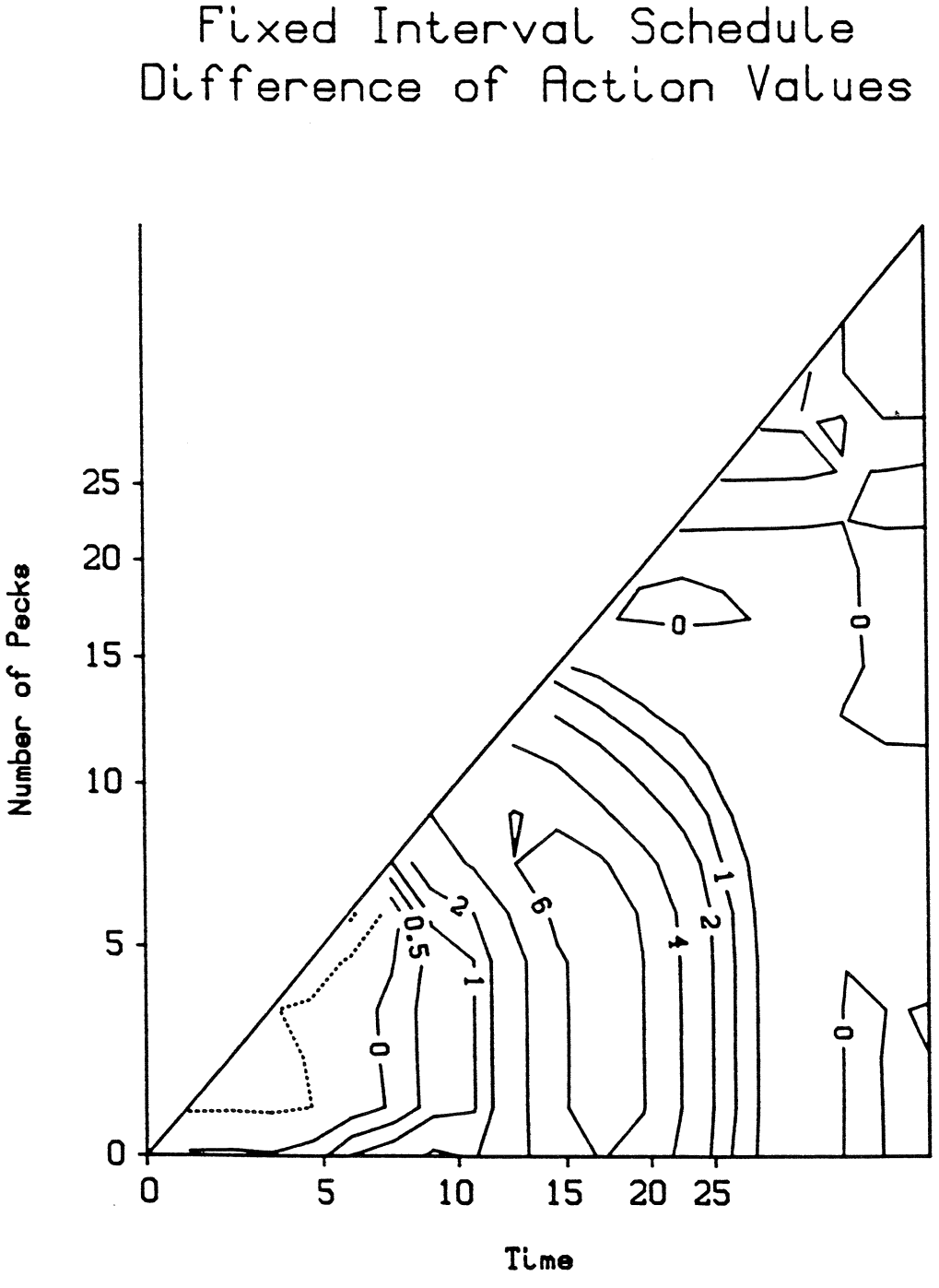
Plot 4


Fixed Interval Schedule
Averaged Performance
For T set at 30, 100, 300
30
100
300
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
0
2
4
6
8
10
12

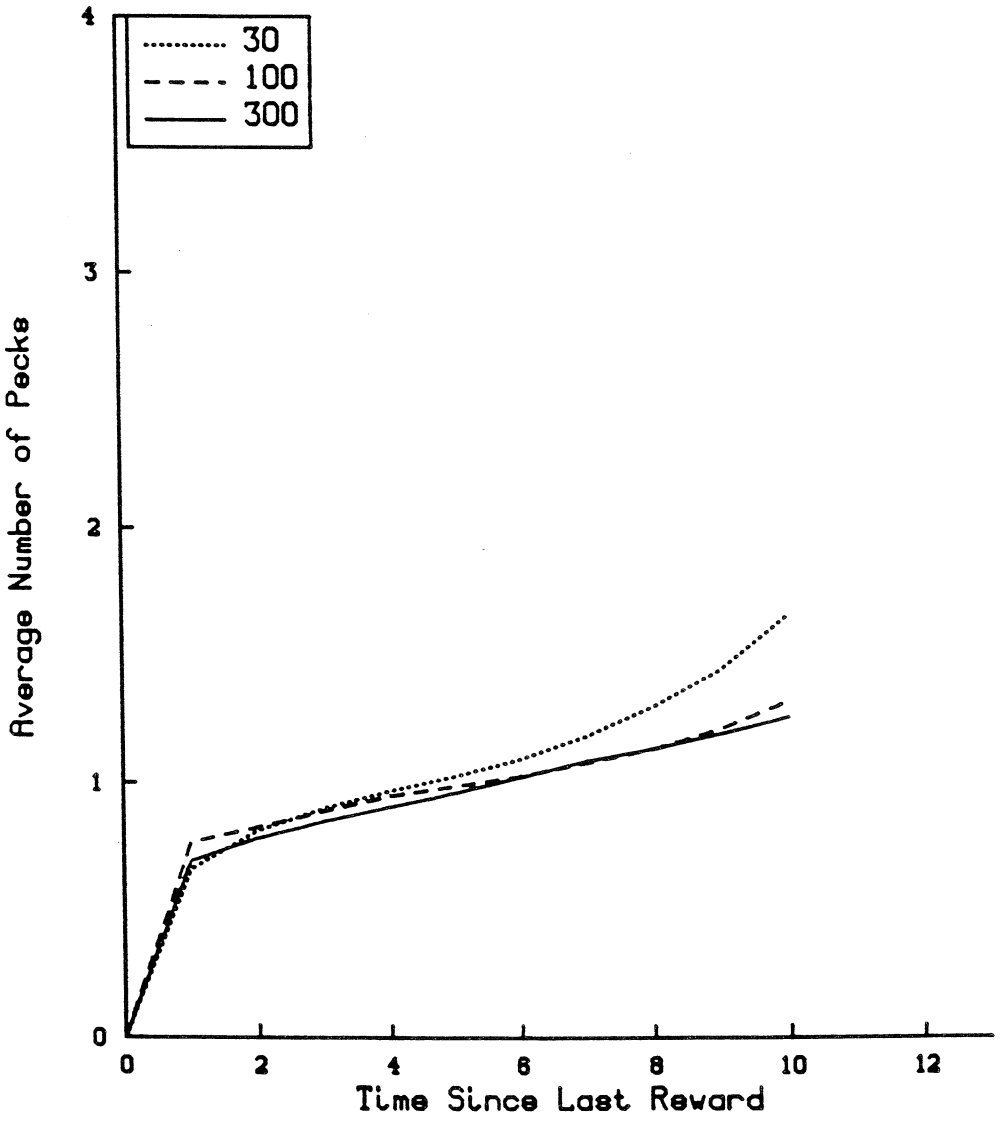
Plot 5


Fixed Interval Schedule
Value Function
Number of Pecks
25
20
15
10
5
0
0
5
10
15
20
25
Time
20
22
18
14
8
4
22

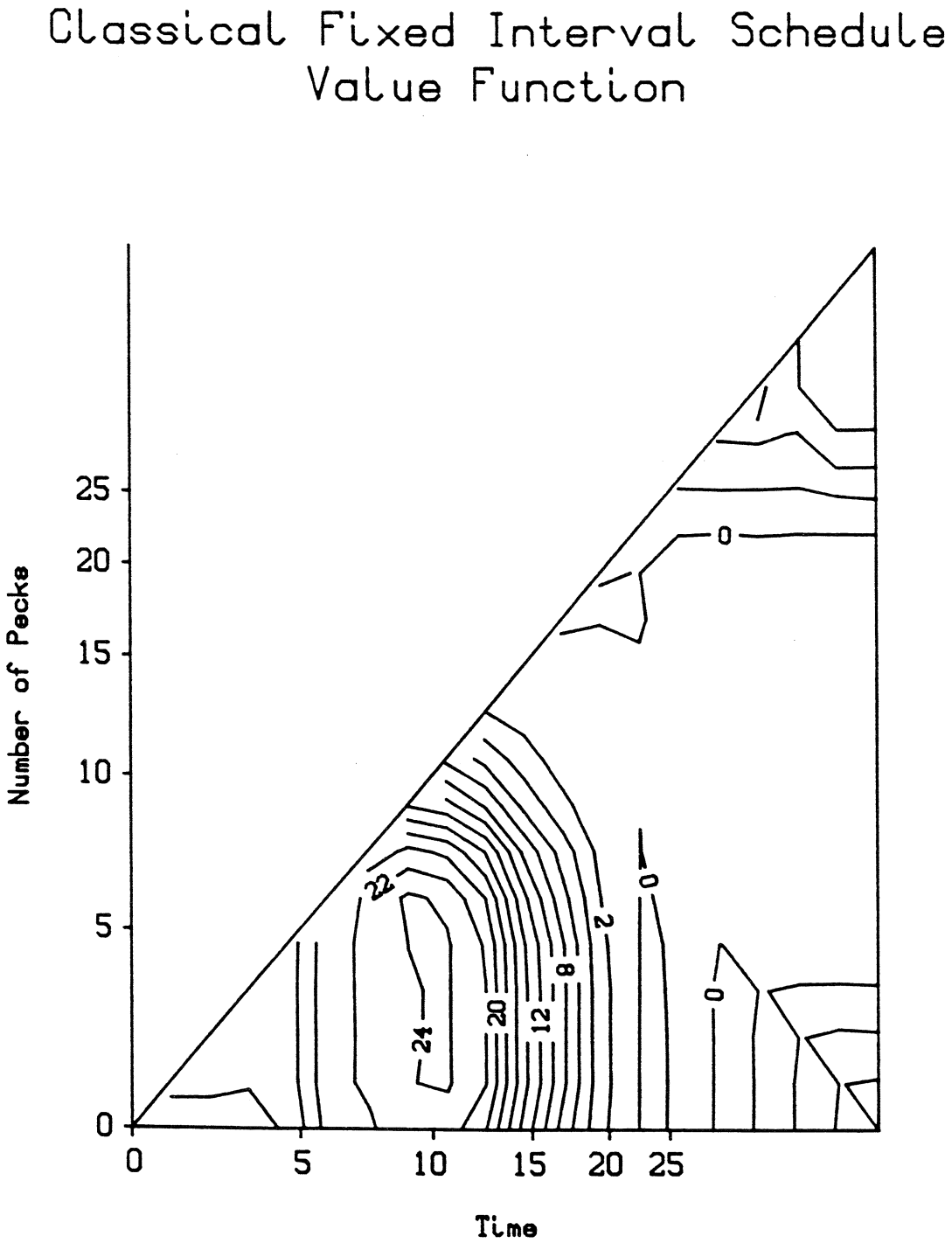
Plot 6


Fixed Interval Schedule
Difference of Action Values
Number of Pecks
Time
0
5
10
15
20
25
0
0.5
1
2
4
6

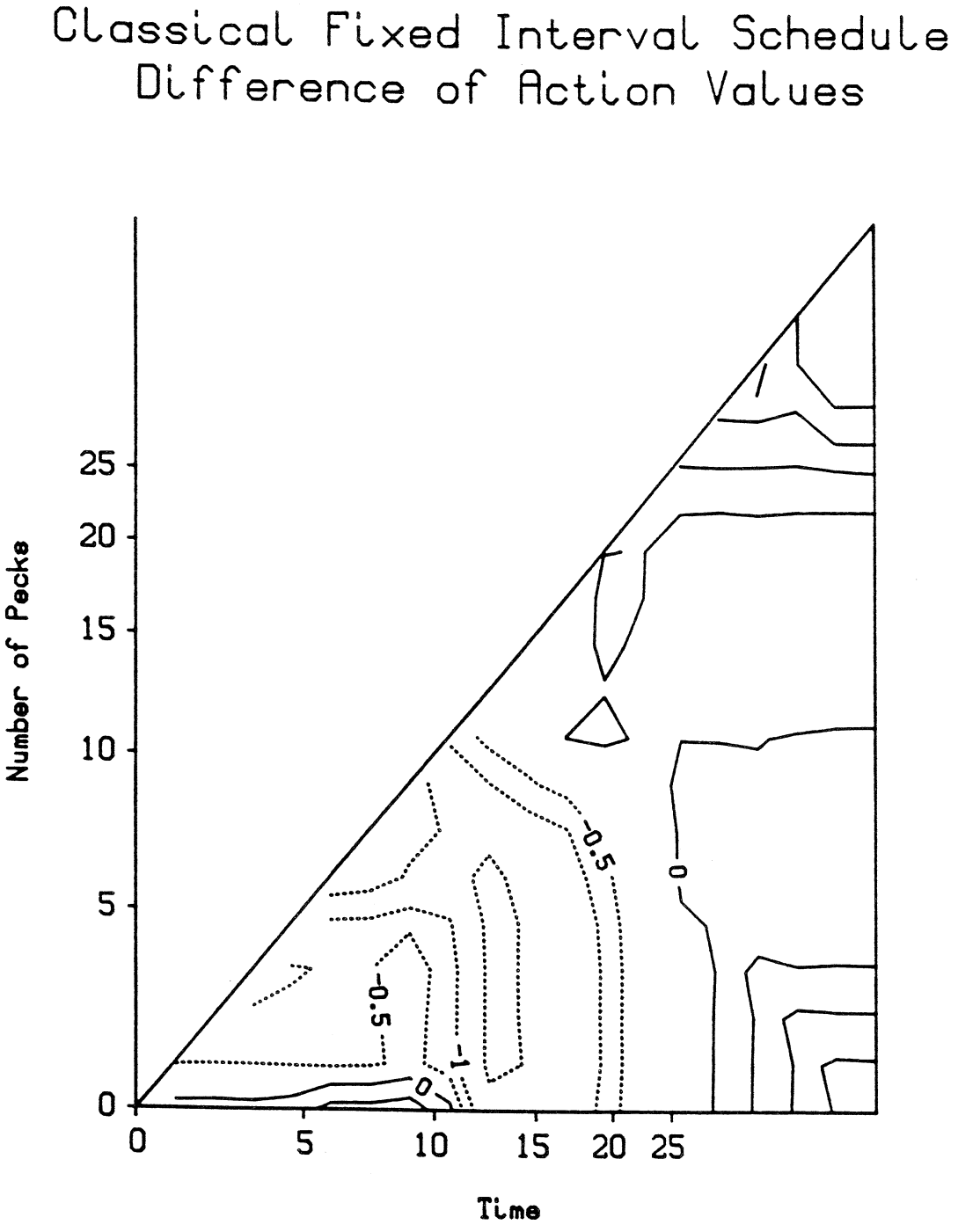
Plot 7


Classical Fixed Interval Schedule
Averaged Performance
For T set at 30, 100, 300
30
100
300
Average Number of Pecks
0
1
2
3
4
0
2
4
6
8
10
12
Time Since Last Reward

Plot 8

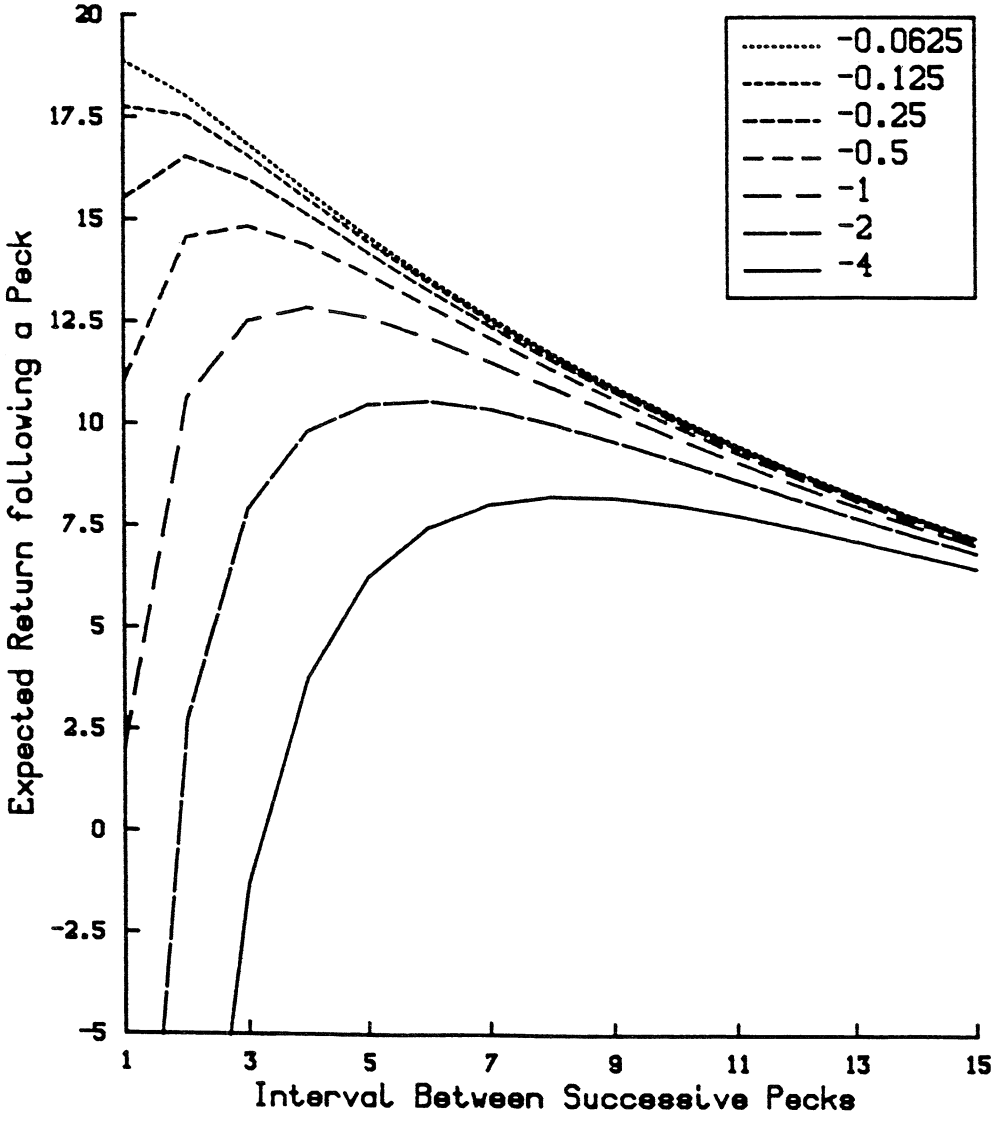

Classical Fixed Interval Schedule
Value Function
Number of Pecks
Time
0
5
10
15
20
25
0
22
24
20
12
8
2
0
0
0

Plot 9

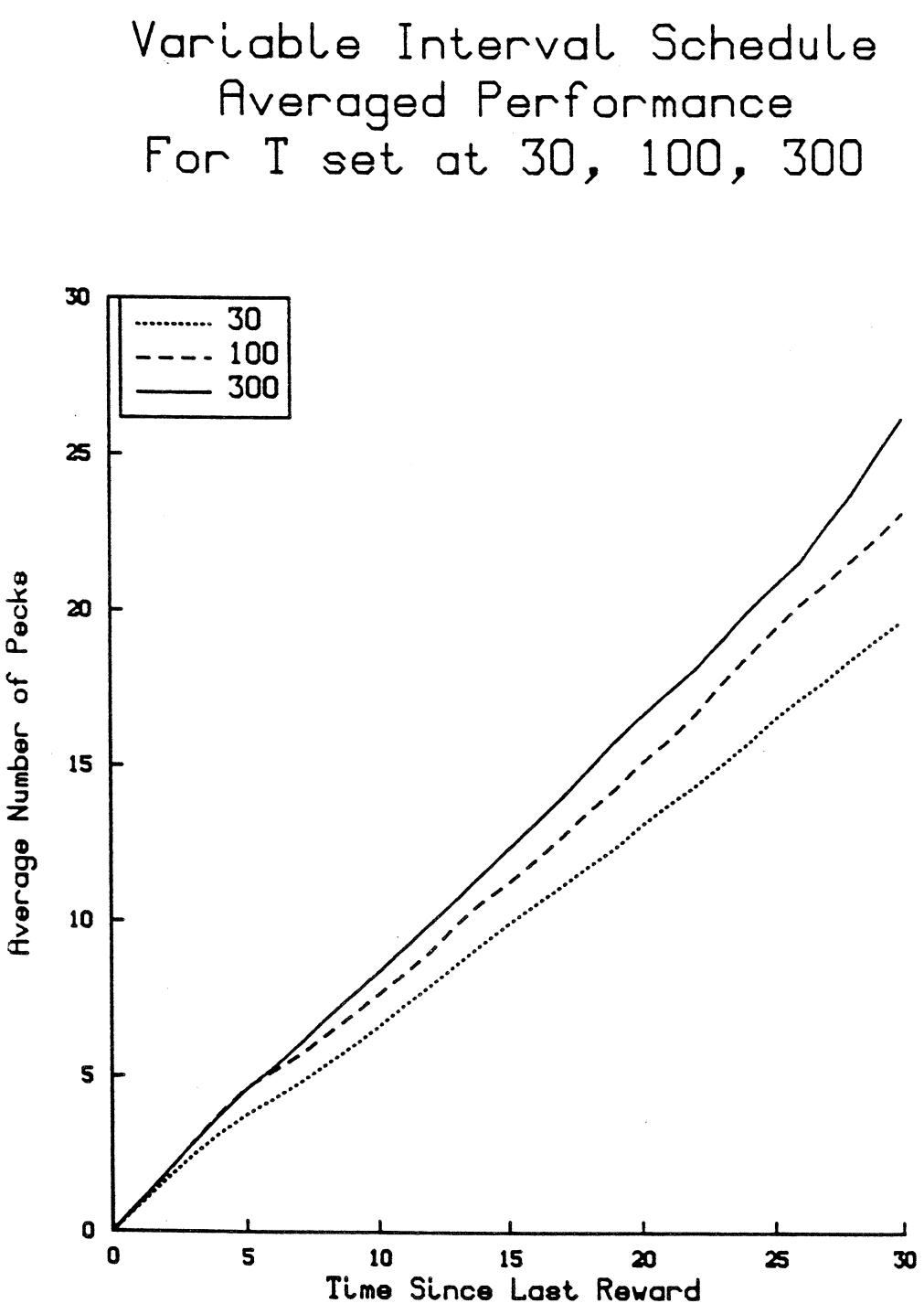

Classical Fixed Interval Schedule
Difference of Action Values
Number of Pecks
Time
0
5
10
15
20
25
-0.5
-1
0
1

Plot 10

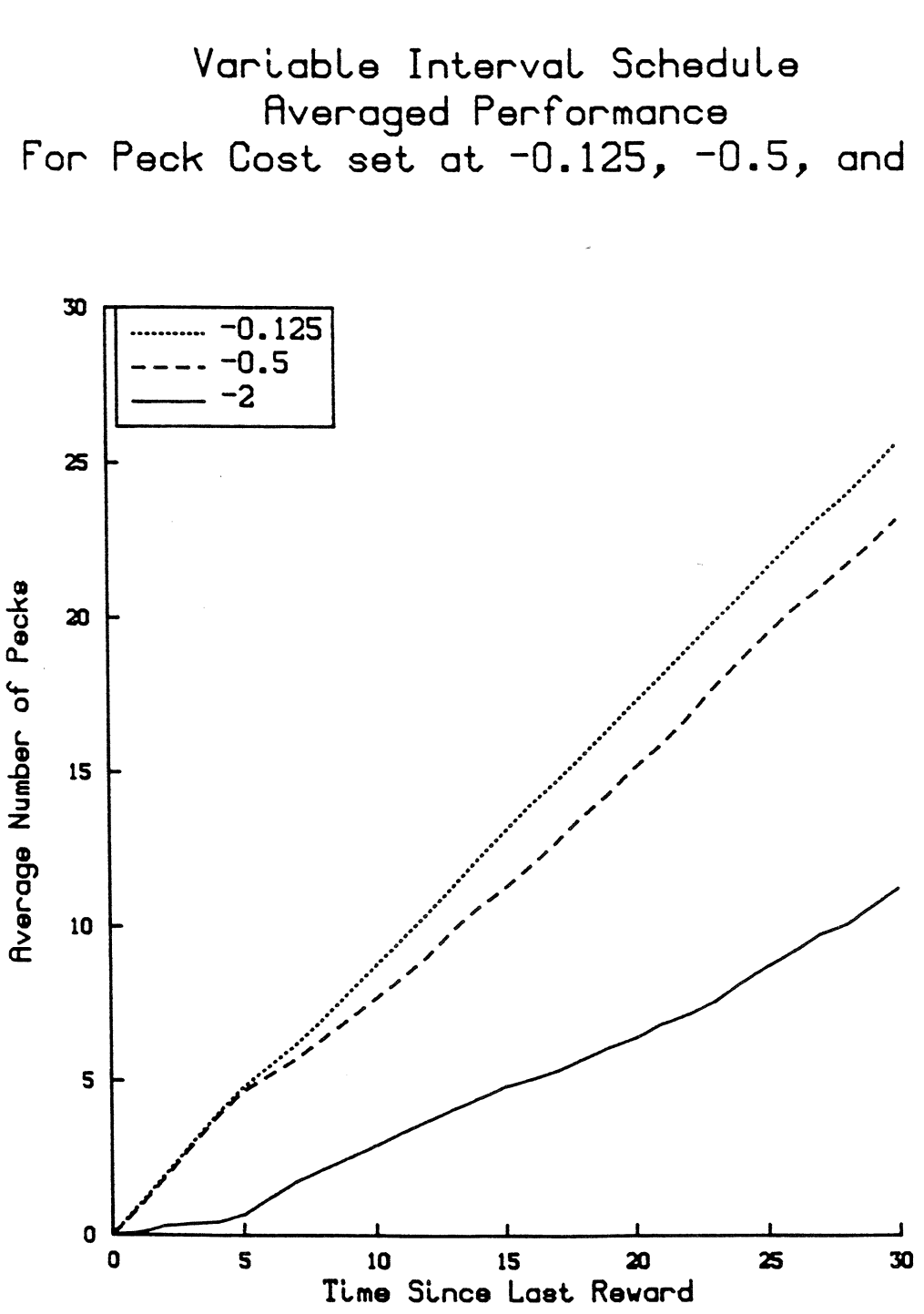
Variable Interval Schedule
Theoretical Expected Returns
with Peck Cost Equal to
-0.0625, -0.125, -0.25, -0.5, -1, -2, -4



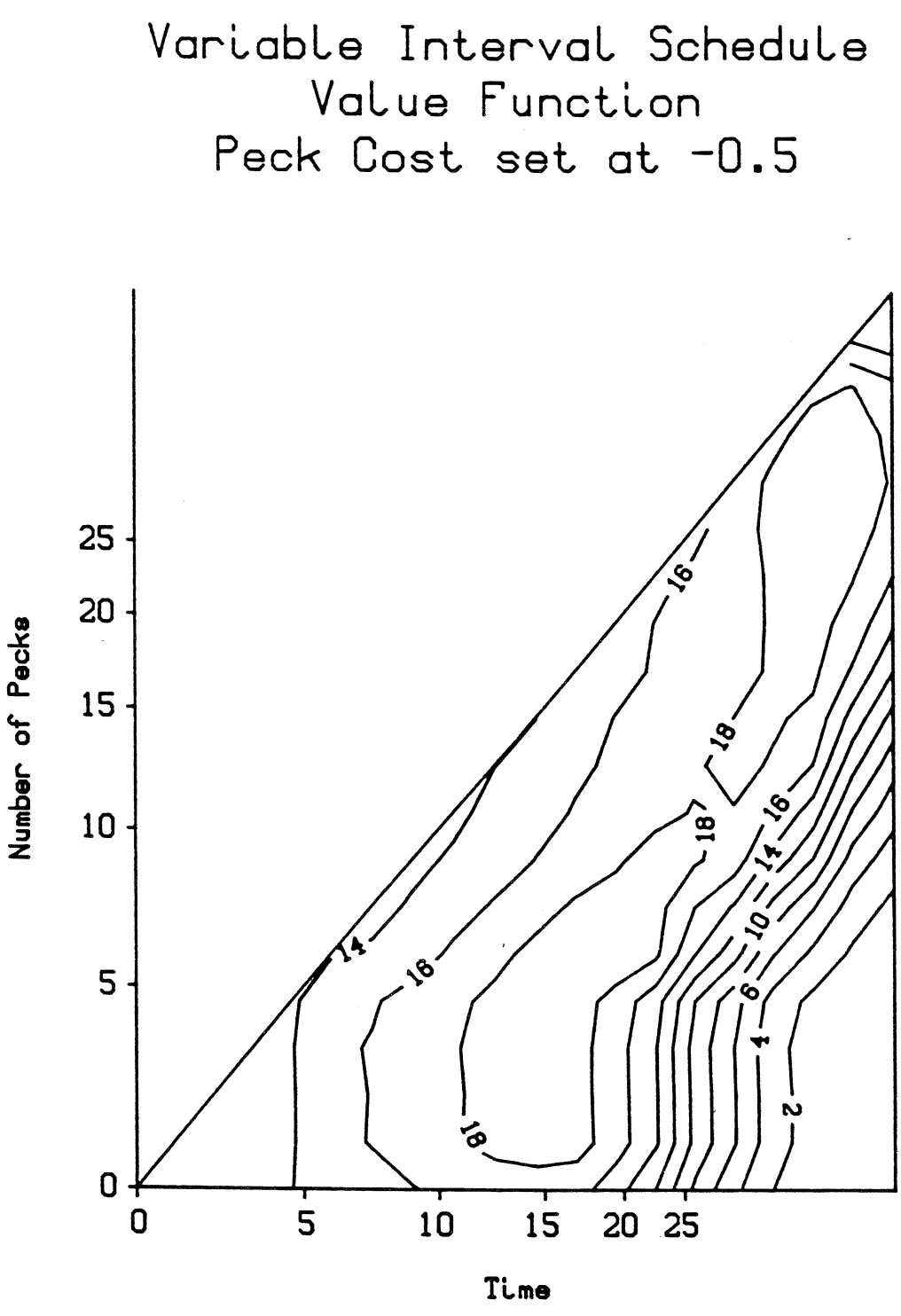
Plot 11


Variable Interval Schedule
Averaged Performance
For T set at 30, 100, 300
30
100
300
Average Number of Pecks
Time Since Last Reward
0
5
10
15
20
25
30

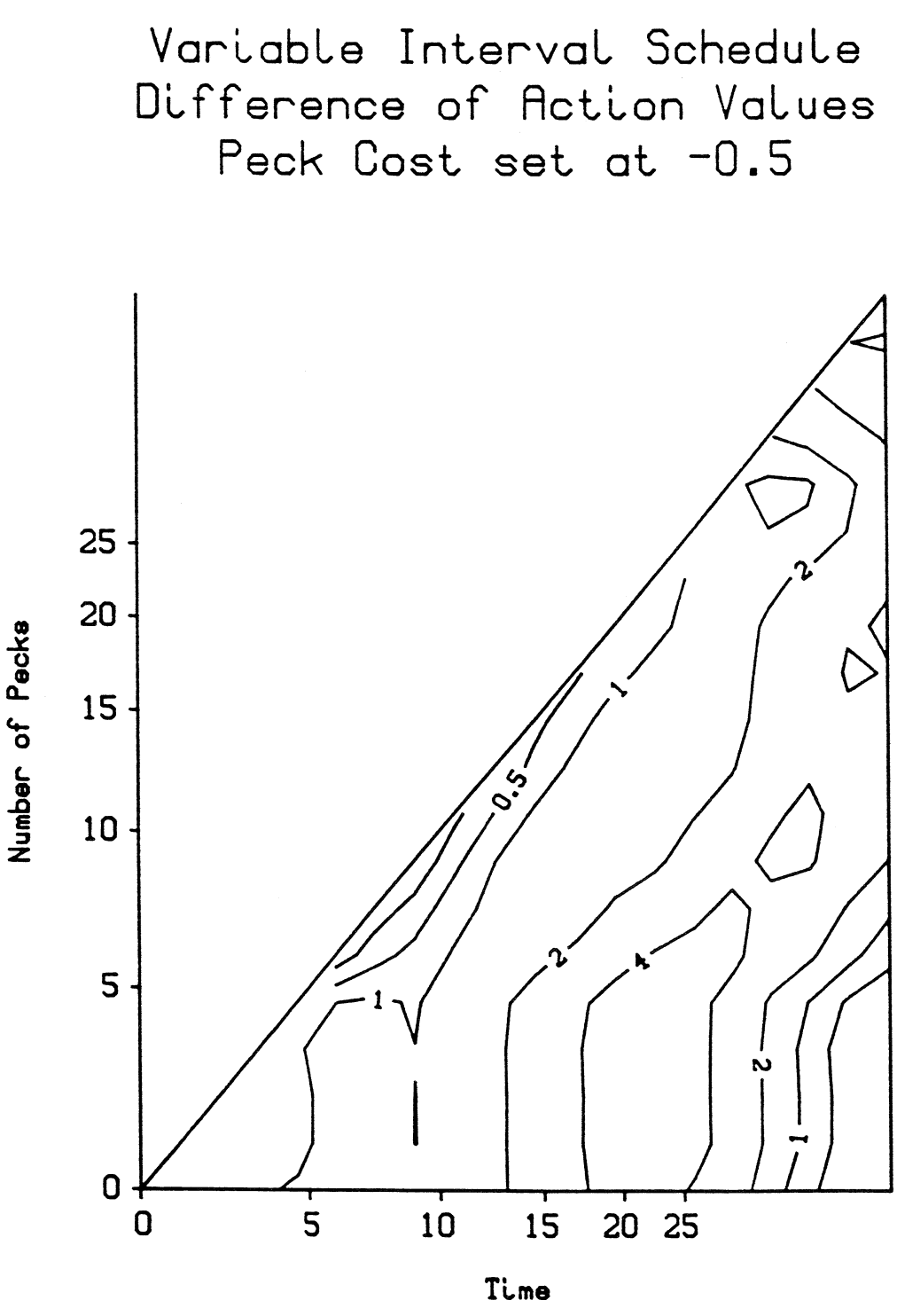
Plot 12


Variable Interval Schedule
Averaged Performance
For Peck Cost set at -0.125, -0.5, and -2
-0.125
-0.5
-2
Average Number of Pecks
Time Since Last Reward
0
5
10
15
20
25
30

Plot 13

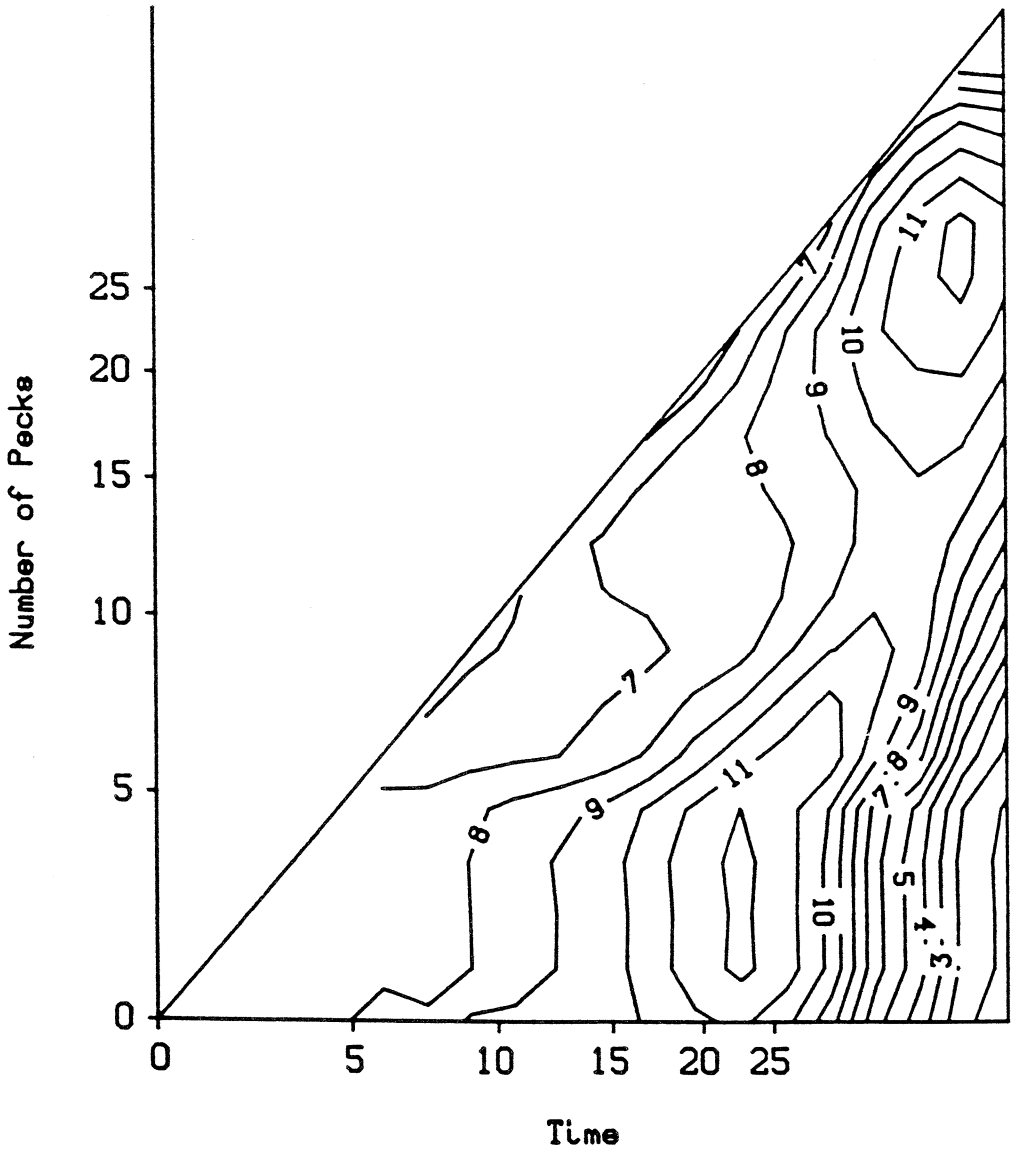

Variable Interval Schedule
Value Function
Peck Cost set at -0.5
Number of Pecks
Time
0
5
10
15
20
25
14
16
18
18
18
16
14
10
6
4
2
18

Plot 14

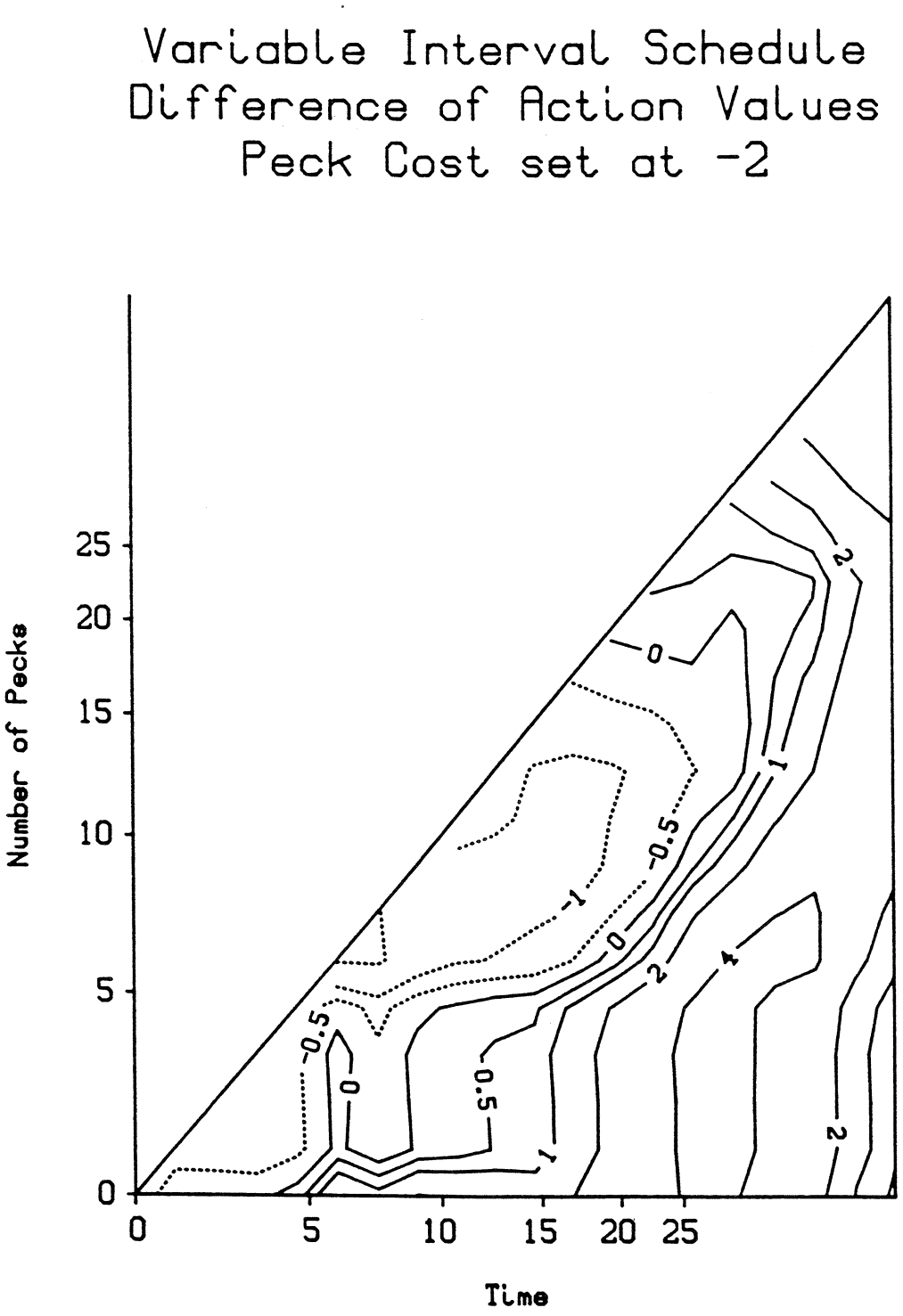

Variable Interval Schedule
Difference of Action Values
Peck Cost set at -0.5
Number of Pecks
Time
0
5
10
15
20
25
0.5
1
2
4

Plot 15

Variable Interval Schedule
Value Function
Peck Cost set at -2

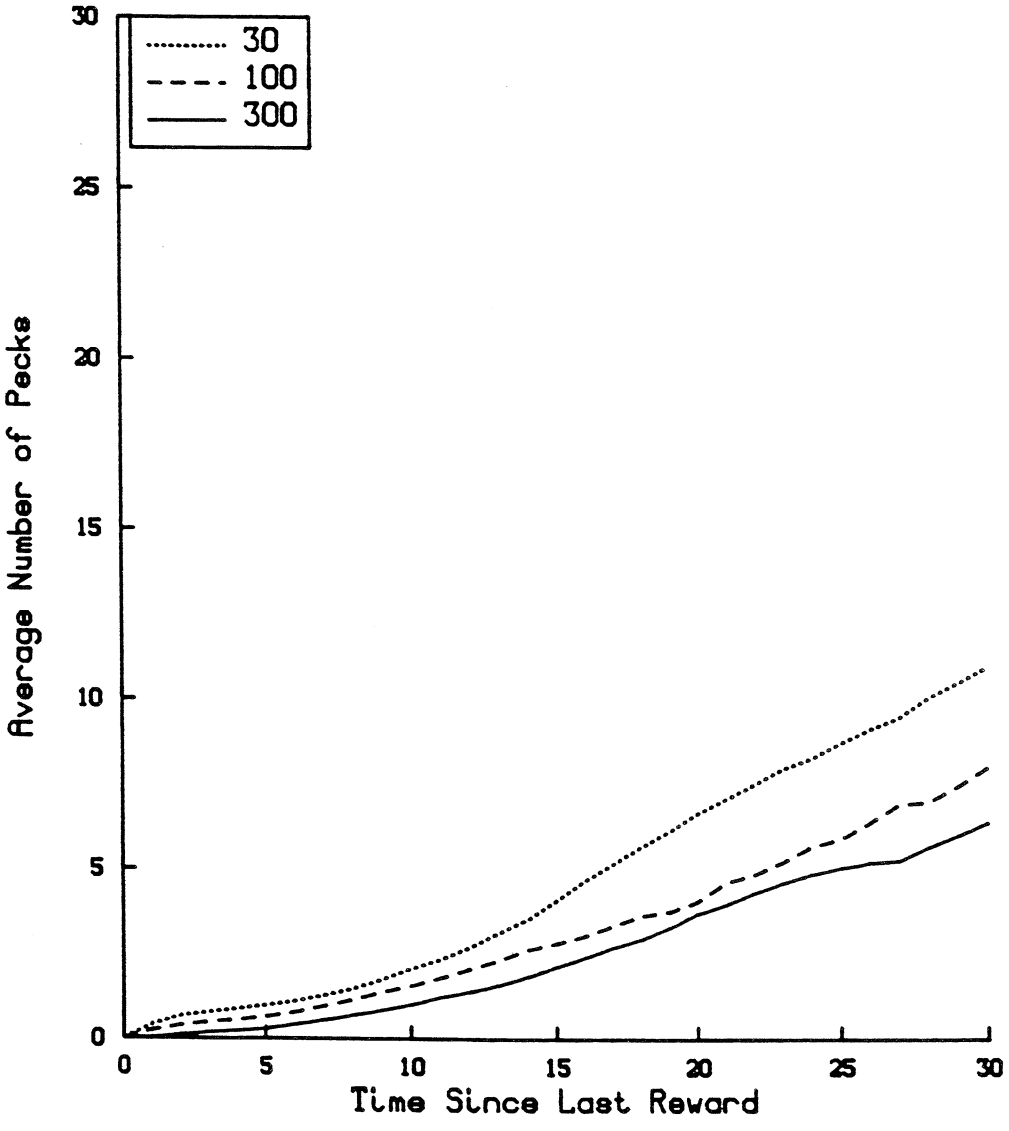


Plot 16


Variable Interval Schedule
Difference of Action Values
Peck Cost set at -2
Number of Pecks
Time
0
5
10
15
20
25
-0.5
0
-1
1
2
4

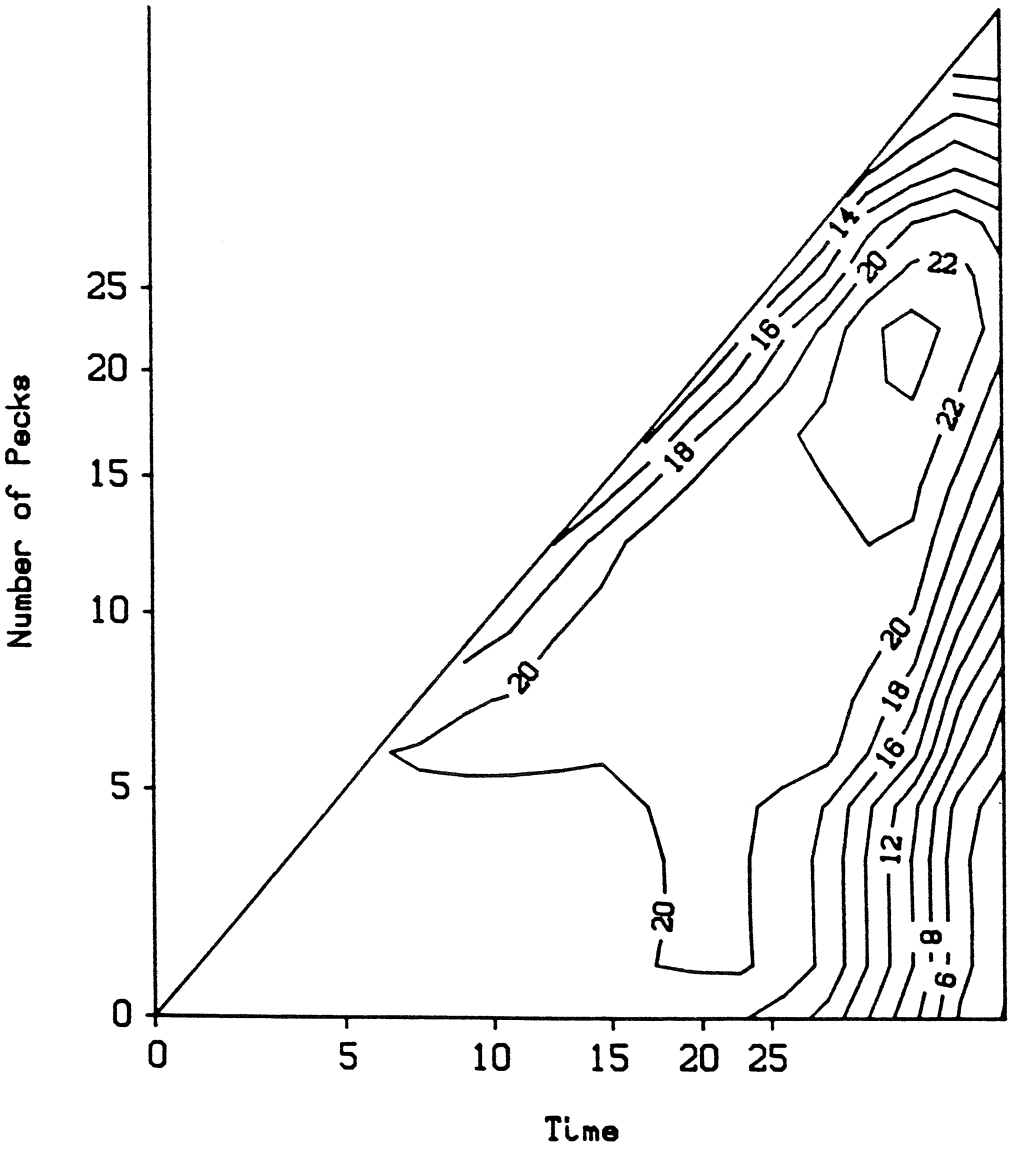
Plot 17

Classical Variable Interval Schedule
Averaged Performance
For T set at 30, 100, 300

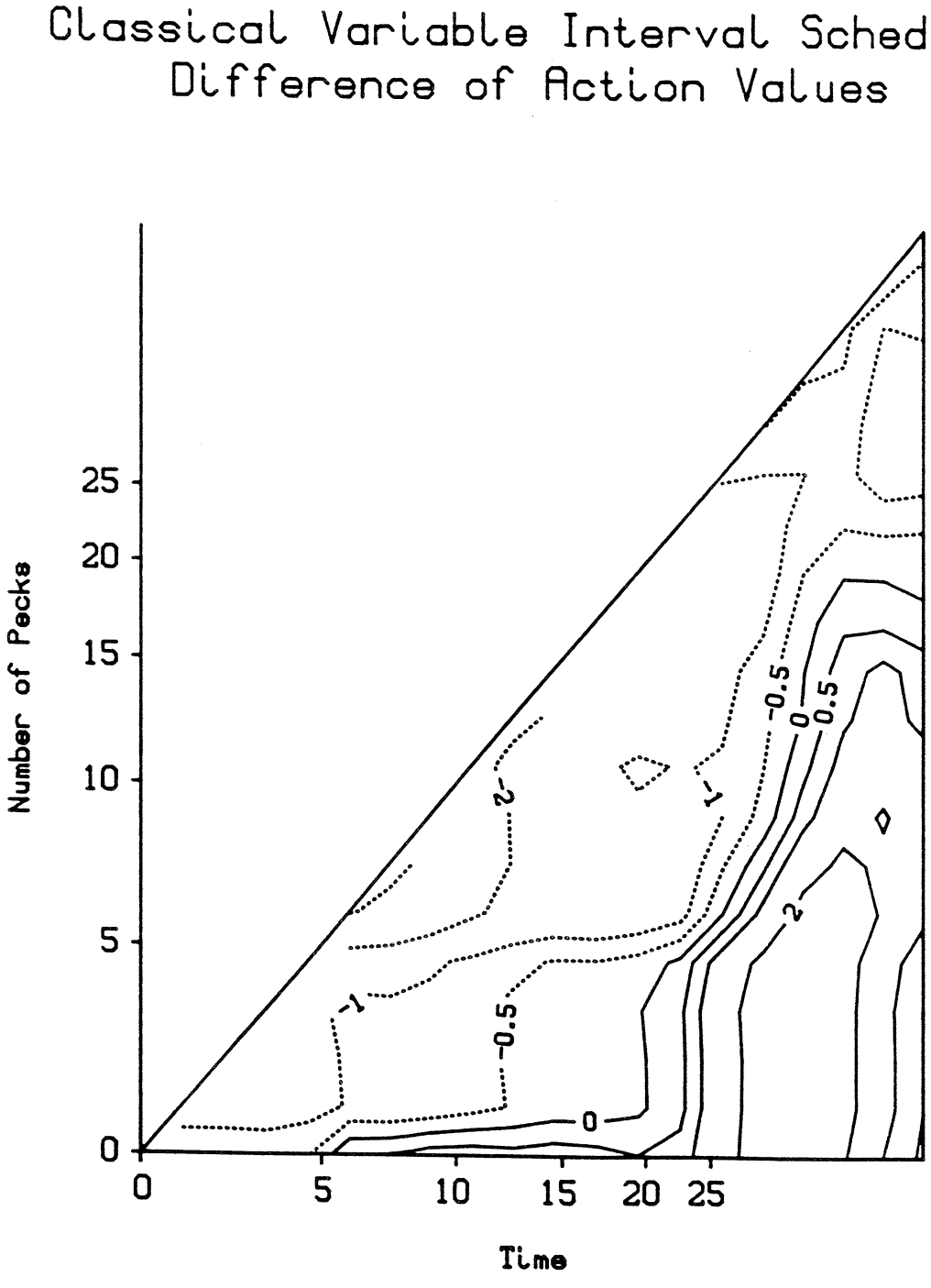


Plot 18

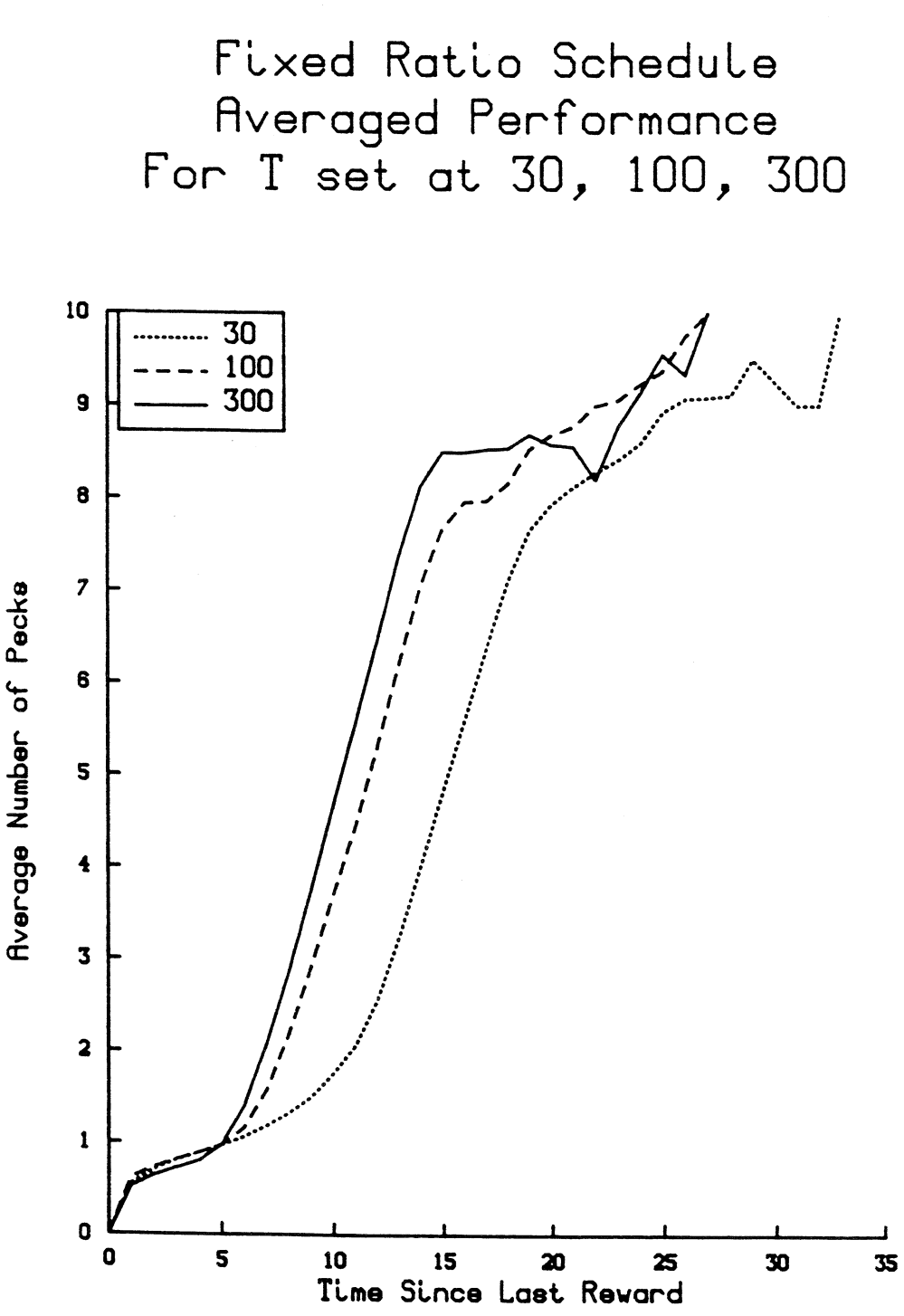
Classical Variable Interval Schedule Value Function



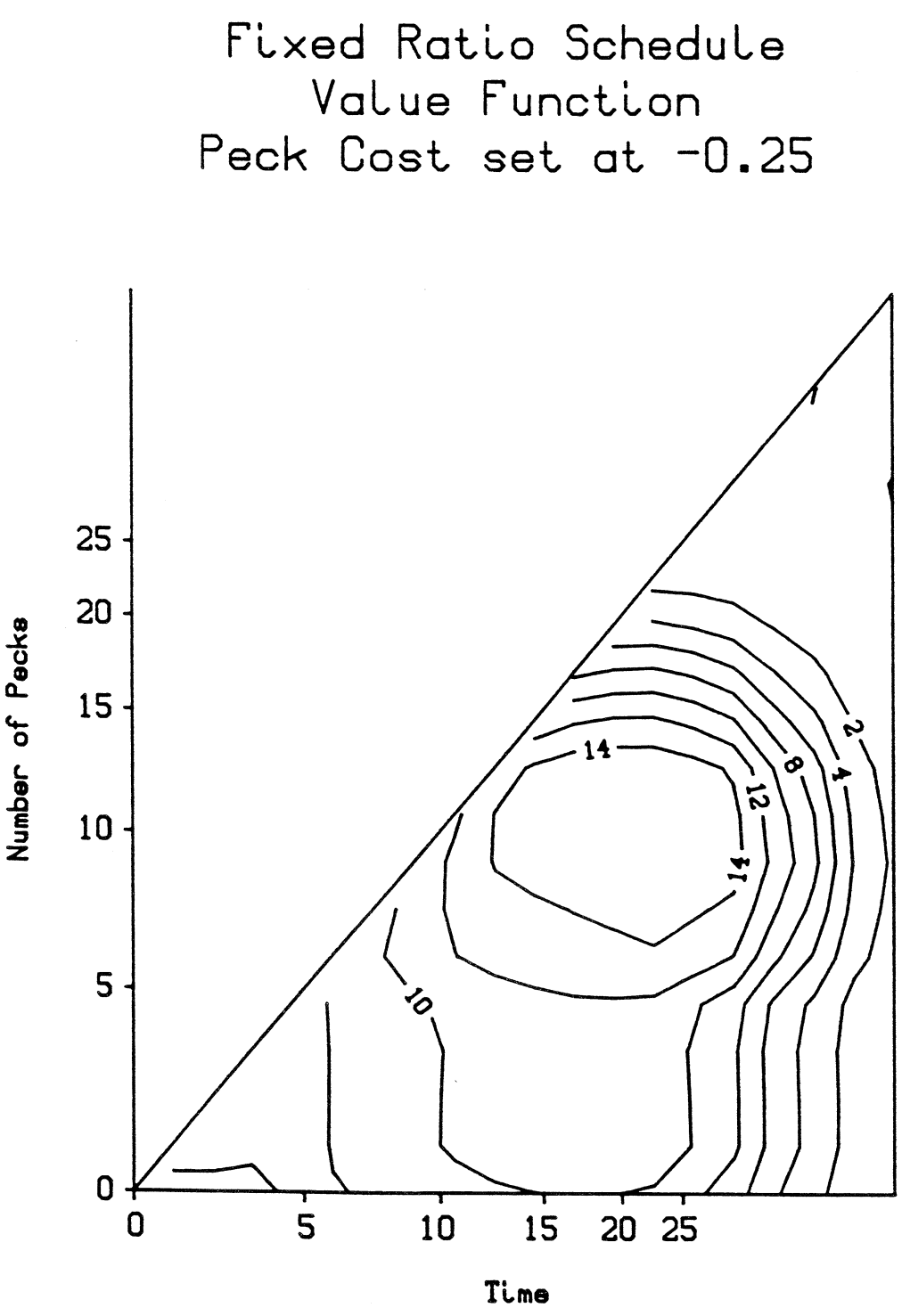
Plot 19


Classical Variable Interval Schedule
Difference of Action Values
Number of Pecks
Time
0
5
10
15
20
25
-2
-1
-0.5
0
0.5
1
2

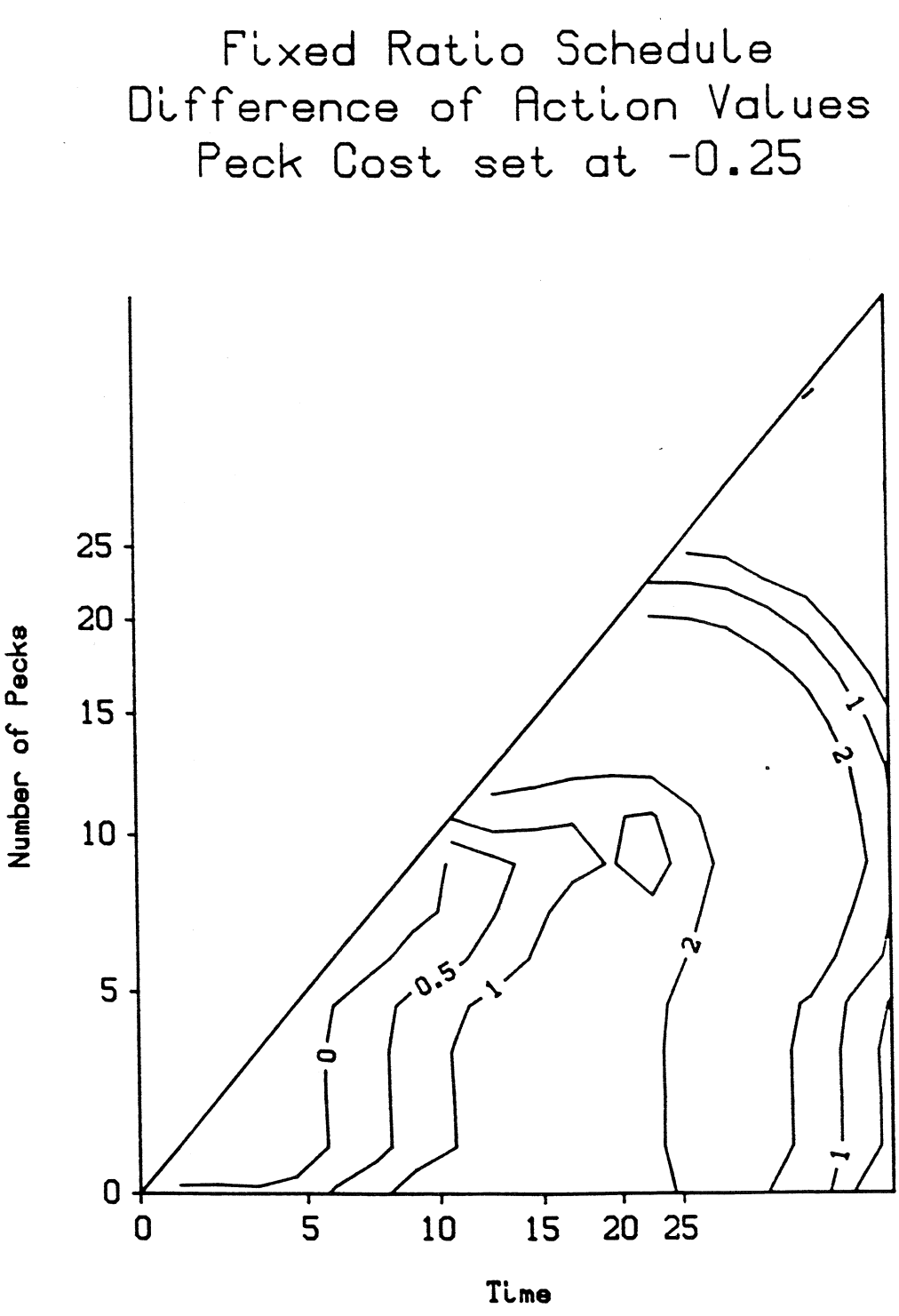
Plot 20


Fixed Ratio Schedule
Averaged Performance
For T set at 30, 100, 300
30
100
300
Average Number of Pecks
Time Since Last Reward
0
1
2
3
4
5
6
7
8
9
10
0
5
10
15
20
25
30
35

Plot 21


Fixed Ratio Schedule
Value Function
Peck Cost set at -0.25
Number of Pecks
Time
0
5
10
15
20
25
1
2
4
8
12
14
11
10

Plot 22


Fixed Ratio Schedule
Difference of Action Values
Peck Cost set at -0.25
Number of Pecks
Time
0
5
10
15
20
25
0
0.5
1
2
-1

Plot 23

8.1. General Comparisons

Plots 1 to 4 each show the averaged performance for all five schedules under different conditions. Plot 1 shows performance with the 'default parameters'. Plot 2 shows performance with the default parameters except that no noise is added to state transitions. Plot 3 shows performance with the default parameters (and with noisy state transitions) except that the learning period M is set to 3, and the rejection factor η is set to zero. The parameters for plot 4 are the same as those for plot 3, except that the rejection factor η is set to 20.

With the default parameters, performance is not optimal under any schedule, but there is some adaptation to each.

Plot 2 shows some striking differences from the first plot. The increased angularity of the curves is because there is now no variation in performance between trials, since state transitions are deterministic. The most notable phenomenon is the very consistent learning in the fixed interval and classical fixed interval schedules: in each of these schedules, all 50 runs resulted in identical final performance. In both the FI and the CFI schedules, the agent always pecked on the first step, which is sub-optimal: I am at a loss to explain this. After the first step, the performance was optimal under both schedules.

Plot 3, with the learning period $M = 3$, shows better performance than plot 1 in all schedules except the classical variable interval. The most striking improvement in performance is in the fixed ratio schedule, where now the optimal policy is followed in the majority of trials: the mean number of pecks after 10 time steps is over 9, so that in most trials the agent pecks continuously until it receives a reward, which is the optimal policy with these parameters (see plot 11). In the VI schedule, the agent makes on average approximately 5 pecks in 15 time steps, which is the optimal policy (I will discuss the optimal VI policy below). In both the FI and CFI schedules, there is a small

improvement over plot 1, in that there fewer pecks on the first time step. Only with the CVI schedule is performance approximately the same as in plot 1.

A plausible explanation for the superiority of the condition M=3 is that under this condition values may be propagated back through the state space faster than is possible with M=1. That is, the transfer of information about state values in one-step Q is slower than in three-step Q-learning.

Plots 3 and 4 are very similar: the rejection factor η appears to have little effect on learning under any of the reinforcement schedules. One reason for this apparent lack of effect is that the performance is nearly optimal on all except for the classical variable interval schedule, so that there is perhaps little scope for any effects of η on performance to emerge. However, performance in the CVI schedule is similar for all four conditions.

8.2. The FI and CFI Schedules

The pronounced FI 'scallop' is apparently the result of state transition noise making it difficult for the agent to judge the passage of time; under these circumstances, the agent may mis-judge when to peck for the reward. In optimal performance, therefore, the agent will sometimes peck too soon and sometimes too late. With the parameter values used, the cost of pecking one step too soon is less than the cost of pecking one step too late, so we may expect the agent to start pecking earlier than later, and this is what apparently happens. This explanation is supported by the fact that there is no 'scallop' in the classical fixed interval schedule—see plot 8.

For the fixed interval and classical fixed interval schedules, the rate of learning had little effect. One reason for this may be that except for the initial peck, the performance achieved may have been the best possible given the state space and the level of state transition noise. No differences will be observed if

near-optimal performance is reached under all conditions.

8.3. The VI Schedule

The VI schedule deserves a special discussion for two reasons: first, the determination of the optimal policy is more complex than for the other schedules; and second, the state space used in the experiments is not adequate for the VI schedule, yet nevertheless passable performance has been achieved.

First, what is the optimal policy under the VI reinforcement schedule? It is easy to show that the form of the optimal policy is to peck periodically, with some constant interval between pecks. An adequate state space for the VI schedule is a measure of how much time has elapsed since the last peck. This information, however, is not carried in the state space used by the learning algorithm. The learning algorithm cannot, therefore, necessarily be expected to find the optimal policy.

The optimal interval between pecks will depend on the probability that the reward is set during each time step, the cost of a peck, and the value of a reward. Let the reward be r, the cost of a peck be c, the probability of the reward being set on each time step be p, and let the expected return on the first step after a peck, for a policy of pecking on every nth step, be $\mathbf{r}$. Then

$$\mathbf{r} = \frac{\gamma^{n-1}}{1-\gamma^n}\,[\,(1-p)^n c + (1-(1-p)^n) r\,]$$

Plot 11 shows the theoretical values of $\mathbf{r}$ calculated according to this equation are plotted against n for r=10, p=0.1, γ=0.95, and for various values of c. For the default parameters, with peck-cost equal to -0.5, the optimal inter-peck interval is 3 time steps, with an interval of 2 time steps being very nearly optimal. The expected return from a policy of pecking on every step is over 25% below the expected return from the optimal policy.

Plot 12 shows the effect of different values of learning rate T: slower learning, with T=300, has the effect of leading the agent to peck on almost every turn. Final performance appears to get worse with increasing T.

Plot 13 shows the average performance with three different values of the cost of a peck, and the rest of the parameters at their default values. Although optimal performance is not achieved in any condition, in each case performance is in the region of 25% below optimal; this is encouraging, considering that the state-space is not adequate.

Plot 17—the difference in action values obtained when the peck cost is set to -2—shows how this behaviour was achieved. The agent 'tracks' the zero contour; if it is on the left hand side of the zero contour, not pecking is preferred, and the state-transitions take the agent horizontally to the right at each time step. When the agent crosses the zero contour, pecking is the preferred action, and the state transition takes the agent upwards and to the right, so that it may find itself on the left hand side of the zero-contour again. The result is that the agent will tend to peck intermittently. How can the agent develop such a policy, when the state-space does not make the necessary distinctions? The answer, I believe, is that when the experiment-choice parameter has become small, the agent does not take just any route to a given point in state space: the agent has a habitual path in the state space, so that its position on the path indicates not only the information carried by the state itself, but also the fact that the agent followed its current policy to get where it is.

8.4. The CVI Schedule

In the classical variable interval schedule, the agent should optimally not peck at all, and it should merely accept the rewards that are intermittently given to it. Why, then, does the agent in these experiments keep on pecking, albeit at

a lower rate than in the operant variable interval schedule? A plausible explanation—and one which is confirmed by examining the contour plots of the value function and the difference of action values below—is that the optimal policy is extreme: not to peck at all. In the early stages of learning, the agent will experiment by pecking on about half of all steps; the states in which it receives the rewards will be states in which it has already pecked several times. It is in these areas of state space that the value function will at first increase; and it is to these areas of state space that the agent will tend to return. When the agent experiments by pecking less, it enters regions of state space where the estimated value function is still low; when this happens, the one-step ahead estimate of eventual return is low, so that not pecking appears to be a bad choice. Once a 'path habit' in state space has formed, the associated ridge in the value function will move only slowly. The initial adventitious correlation of past pecking with rewards will thus cause the agent to develop a 'superstition' (Skinner 1948) that the pecking leads to the rewards. Superstitions of this type are difficult for an agent to get rid of if it is restricted to making short term estimates of returns.

8.5. The FR Schedule

The appearance of the performance for the FR schedule in plot 21 is deceptive because of the averaging: the FR curve is the result of averaging many curves that start off with few pecks and finish with a peck on every time step. The initial periods of procrastination under the FR schedule are of different lengths, so that the average of the curves has a sigmoid shape.

The optimal policy in the FR schedule is either not to peck at all, or else to peck at every opportunity. So why should the agent procrastinate in this way? A plausible explanation is that the agent starts off pecking in 50% of the

time steps, so that it collects its reward after 10 pecks and, on average, 20 time steps. A peak in the value function starts to develop in this area of state space—this peak is visible in plot 22. A 'mountain' of the value function will spread across the state space; the 'well worn path' on which a 'spur' of the mountain will develop will consist of procrastinating for a time, and then of pecking continuously. The first path that consistently leads to rewards tends to persist. With M=1, it takes a considerable time for the good news that after 10 pecks there will be a large reward to percolate back to the states corresponding to one or two pecks. In these states, pecks appear to be costly, so that in the initial stages of learning, the agent learns *not* to peck at early times, and to peck continuously at a later time in the trial. This can be seen by examining the zero contour in plot 23—the shape of the zero contour is similar to that for the FI schedule.

9. Discussion

The Q-learning algorithm is capable of acquiring near-optimal policies under a variety of reinforcement schedules. The most significant findings are first that a longer learning period (M=3) gives a considerable improvement in performance. Second, even though the state space is inadequate for the VI reinforcement schedule, the average performance is not grossly sub-optimal, and it varies with peck-cost appropriately.

But, I believe, the main message of these results is that the course of learning is strongly affected by adventitious correlations and by the initial policy and value function. The state-action combinations that happen to precede rewards will tend to be repeated, whether the actions caused the reward or not. Because there is no way for the agent to find out which actions are necessary and which are unnecessary other than by systematic trial and error, some of the

agent's actions will appear 'superstitious' to a more knowledgeable observer, but the agent cannot avoid this during the early stages of learning.

The learning method might be improved by making λ_t depend on Y; λ should be close to 1 for small Y, and λ should be smaller at states with large Y. Setting λ adaptively in this way might combine the best features of one-step and many-step Q-learning.

Chapter 12
Conclusion

1. What has been Achieved?

A family of simple algorithms for associative learning has been described systematically; these algorithms may be implemented as computer programs, and they can be applied to models of problems that have been given to animals. The algorithms themselves have been developed by an argument from first principles, and not with the intention of explaining particular experimental results. They can be motivated, and in one case justified, as forms of incremental dynamic programming, and they can be viewed as methods for optimising short to medium term averages of rewards and costs that result from action. The algorithms can also be viewed as direct implementations of associative learning according to the law of effect. They can be applied to a wide range of simple tasks, and not just to the tasks that have been used in the study of animal learning.

The value of the incremental dynamic programming approach is that it provides a framework according to which a variety of learning algorithms may be described in common terms. The framework makes clear both the potential scope and also the limitations of the learning methods.

2. Further Work

The first area in which further work is needed is in enabling autonomous agents to tune the values of the parameters for their learning algorithms. If, for example, an agent chooses a learning factor that is too small, then it will make insufficient use of its experience; but if the learning factor is too large, the policy learning may become unstable. An autonomous learning agent needs to have some method of choosing its suitable values for its learning parameters itself.

A second area that requires further work is that of methods of representation and approximation of functions. I have used only one method—the CMAC—which is crude and simple. An important question is that of what other representations may be used in conjunction with these learning methods. In particular, are there connectionist methods of learning functional mappings that could be used as component modules of strategy learning mechanisms?

Third, the life of even a simple animal cannot be treated as a single Markov decision problem, because the learning problem becomes too complex. I feel that the most interesting possibility for further work is that of linking Markov decision problems together, into hierarchies and other configurations, in such a way that the control of each decision process can be learned individually. One attraction of this approach is that there are likely to be mathematical methods for studying the interactions between linked decision problems. Although I have not yet implemented any such program, the approach appears most promising.

3. Computational Theories of Intelligence

As I said at the beginning of the introduction, the fields of cognitive science and artificial intelligence have been concerned mainly with analysing and modelling the abilities of humans, rather than those of animals. Many examples of human performance in performing various cognitive tasks have been analysed, and computer programs have been constructed that can solve some of the same problems that people can solve, in apparently similar ways. Has this led us much closer to an understanding of human intelligence?

I do not think so, because there is an insidious methodological problem with this approach. The problem is that when researchers set out to study people's higher thought processes, they set their subjects tasks to perform, and then study their subjects' performance *on those tasks*. The result of the work is a computational theory of how people perform *those tasks*. The computational theory consists of a description of hypothetical 'cognitive operations' that people perform in accomplishing the tasks; ideally, this description should take the form of a computer program, in which certain blocks of code or defined procedures correspond to and perform the same role as the cognitive operations that the subjects perform. In other words, the computer program serves as a description of an algorithm that the subjects follow in performing the task.

The methodological problem is that what has been achieved is to describe the algorithm that the subjects have chosen to use: what has *not* been done is to explain how the subjects decided what mental algorithm to use. The mental algorithm is merely a *product* of the subjects' intelligence.

A classic example is also one of the clearest: after reading Newell and Simon's (1972) study of cryptarithmetic, one feels one has learned some useful tips on how to solve cryptarithmetic problems, but very little about people. Newell and Simon devised an elegant notation to describe what people do in

solving DONALD + GERALD = ROBERT, in terms of formulating goals and subgoals, and searching, more or less systematically, for assignments of digits to letters that are consistent with the sum given. This is an excellent description of *what* people do while they are tackling a cryptarithmetic problem, and the computational model can no doubt be used to predict which cryptarithmetic problems should be easy and which should be difficult, what errors people usually make, how long people should take to find solutions on different problems, and so on. But the computational theory has absolutely nothing to say about *why* people choose to proceed in that way, or why they believe that the procedure they use should lead to a solution if one exists. Newell and Simon describe what their subjects did, but they do not describe how or why their subjects approached the problem in that way, which is a much more interesting question.

In later work, such as that of Laird, Rosenbloom, and Newell (1986), there is an attempt to construct a system that is able to formulate problem descriptions of this type by solving a higher level problem of the same form. This is perhaps the only significant attempt so far in the field of AI to give a general theory of intelligence, but I do not think that the attempt comes close to success because, as far as I can see, the problem formulations that the program finds have to be built into the program beforehand in a rather specific way.

The basic problem in the computer modelling of human thought is that the researcher faces a dilemma: the model he or she constructs needs to be simple enough to be supported by the experimental evidence that can be collected, and yet a model can only be plausible as an explanation if it can be presented as a part of, or as a product of, some vastly more complex and unknown system.

But animal cognition is likely to be simpler, and that of very simple animals is much simpler. In this thesis I have set out systematically a range of

ways in which simple agents might control their behaviour, and a range of learning algorithms that such agents might use to optimise their behaviour according to certain plausible criteria. It should be feasible to construct computer simulations of autonomous agents that learn to fend for themselves in simulated environments. Indeed, Wilson (1987) has already attempted to do this. Although considerable work would be needed, it seems by no means an impossible objective to construct a relatively simple, autonomous, learning program that shows, in simulation, a similar range of associative learning abilities to those that have been demonstrated in the rat.

A promising approach to the study of intelligence is to start off by considering what simple intelligences are. The aim should be to give general abstract definitions of simple forms of intelligence, and to construct such intelligences in their entirety.

Appendix 1
Convergence of One-Step Q-Learning

The agent learns using an initial estimate of Q, and data from experience, which consists of observations of the form

$$[\, x \quad a \quad r \quad y \,]$$

which are respectively the state, the action taken at the state, the immediate reward received, and the subsequent state reached. Let us assume that the agent's data consists of an infinite sequence of observations, numbered 1,2,3, $\cdots$ which are used successively to update Q. Let the nth observation in the list be $[\, x_n \;\; a_n \;\; r_n \;\; y_n \,]$.

The observations are assumed to be independent observations of state-transitions and rewards in a Markov process. There is no assumption that the observations come from a connected sequence of actions—x_{n+1} does not have to be the same as y_n. The observations, therefore, can be collected from short disconnected sequences of behaviour, and the choices of actions at states may be arbitrary. The only constraint on the sequence of observations is that there must be sufficient observations of each action at each state: this will be made precise below.

To show that the learning method converges, I will first show how to construct a notional Markov decision process from the data: this notional decision process is a kind of 'action replay' of the data. Next, I will show that the Q values produced by the one-step Q-learning method after n training examples have been used are the exact optimal action values for the start of the action-replay process for n training examples. Finally, I will show that, as more data

is used, the optimal action-value function at the start of the 'action replay' process converges to the optimal action-value function of the real process. For brevity, I will refer to the action-replay process as ARP and to the real process as RP.

1. Using the Observations to Adjust Q During Q-Learning

Recall that the one-step Q-learning method is as follows. The initial values of Q, before any adjustments have been made, are $Q_0(x,a)$ for each state x and action a. After the nth observation has been used to update Q, the values of Q are written Q_n . The estimated value of a state x at the nth stage is

$$U_n(x) = \max_a \{ Q_n(x,a) \}$$

The nth observation, for $n = 1, 2, 3, \cdots$ is

$$[x_n \quad a_n \quad r_n \quad y_n]$$

and it is used to calculate Q_n from Q_{n-1} by

$$Q_n(x,a) = \begin{cases} (1-\alpha_n)\, Q_{n-1}(x,a) + \alpha_n [r_n + \gamma\, U^Q_{n-1}(y_n)] & \text{if } x = x_n \text{ and } a = a_n \\ Q_{n-1}(x,a) & \text{otherwise} \end{cases}$$

The learning factor α_n may depend on x_n and a_n. I will discuss the requirements that the learning factors must satisfy later.

2. The 'Action-Replay' Markov Decision Process

The action replay process is a purely notional Markov decision process, which is used as a proof device. This process is constructed progressively from the sequence of observations. The ARP consists of layers of states, numbered $0, 1, 2, \cdots, n, \cdots$. In each layer—in the kth layer, say—there is a state $\langle x,k \rangle$ in the ARP corresponding to each state x in the RP. That is, the state in

the kth layer of the ARP corresponding to state x in the RP is $\langle x,k \rangle$. The kt layer of states of the ARP is constructed when the kth observation is processed.

At the state $\langle x,k \rangle$, the same actions are possible as at state x in the RP but their effects are different.

Actions in the ARP are defined in the following way. The essential idea is that an action in the ARP is a 'replay' of an observation. The ARP is a 'model' of the RP, in which performing an action a in state x is simulated by recalling an observation $[\, x\ a\ r\ y\,]$ of performing a in x, and then using the observed r and y as the simulated reward and new state respectively.

To make this more precise, suppose that one is at state $\langle x,k \rangle$ in the ARP, and one wishes to perform action a. To do this, one must find one of the first k observations to 'replay'; an observation is *eligible* for replaying if it was observed before observation k and if it is of the form $[\, x\ a\ .\ .\,]$, where the x and the a correspond to the state $\langle x,k \rangle$ one is at and the action one wishes to perform respectively. If the lth observation, which is $[\, x\ a\ r_l\ y_l\,]$, is eligible, and is selected for replay, the reward obtained is r_l, and the next state in the ARP reached is $\langle y_l, l-1 \rangle$. At this new state $\langle y_l, l-1 \rangle$, one may repeat the process: one may choose an action to perform— b, say—and then one may look for a suitable observation to replay. This time, however, only the first l observations are eligible. With each action taken in the ARP, the list of observations that are eligible for replay becomes shorter, until finally there is no observation eligible for replay. When there is no eligible action left, as must eventually happen, a final payoff is given, which is $Q_0(z,c)$, where z and c are the state one is at and the action one is trying to perform when one runs out of actions to replay. Starting at any state in the ARP, it is, therefore, only possible to perform a finite number of actions before running out of observations to replay.

I have not yet explained exactly how observations are selected for replay. this is done according to the following (randomised) algorithm, which I will first describe in words, and which I will then give in 'pseudocode'. To perform a in $<x, k>$, one first checks whether the kth observation is eligible. If not, one examines the $k-1$th observation, then the $k-2$th, and so on, until one finds a an eligible observation—number l, say. Let α_l be the learning factor that was used when Q was adjusted when observation l was processed. Then, one takes a random choice: with probability α_l one 'replays' observation l, in that one goes to $<y_l, l-1>$ and one takes an immediate reward r_l. If the random decision goes the other way, one continues to scan back along the list of observations until another eligible observation is found, and then one repeats the random choice. If one reaches the beginning of the list of observations, and then one takes the immediate reward $Q_0(x,a)$, and no further actions are possible, so that the ARP terminates.

To put this another way, let the current state be $<x,k>$ and let the action to be performed be a, and let the eligible observations be numbered n_1 , n_2 , . . . , n_i , where

$$n_1 < n_2 < \cdots < n_i \leq k$$

Then the probability of replaying observation n_i is α_i ; the probability of replaying observation n_{i-1} is $(1-\alpha_i)\alpha_{i-1}$, and so on. The probability of not replaying any of the eligible actions is

$$(1-\alpha_i)(1-\alpha_{i-1}) \cdots (1-\alpha_1)$$

If no eligible action is selected for replay, the ARP terminates, with a final payoff of $Q_0(x,a)$.

Procedural instructions for performing action a in state $<x,k>$ may be given recursively as

To perform a in $\langle x,0\rangle$,

terminate the ARP with an immediate reward of $Q_0(x,a)$,

and halt.

To perform a in $\langle x,k\rangle$, for $k > 0$,

if $x = x_k$ and $a = a_k$

then

begin

either (with probability α_k)

go to $\langle y_k, k-1\rangle$ with an immediate reward of r_k

and halt,

or (with probability $1 - \alpha_k$)

perform a in $\langle x, k-1\rangle$

end

else

perform a in $\langle x, k-1\rangle$.

The ARP is a decision process: if performing a in $\langle x,k\rangle$ leads to the state $\langle y,k-m\rangle$, then one may choose to perform any of the actions possible in the RP at y in $\langle y,k-m\rangle$. It is not possible to perform an infinite sequence of actions in the ARP—no matter what actions are chosen, if one starts at level k, each action will lead to a new state at a lower level, until finally one reaches level 0 and the process terminates.

The return of a sequence of replays of observations k_1 , k_2 , . . . , k_n (such that $k_1 > k_2 > \cdots > k_n$) is

$$r_{k_1} + \gamma r_{k_2} + \cdots + \gamma_n r_{k_n} + \gamma_{n+1} Q_0(y_{k_n} , a)$$

where a is the action chosen in the last state $\langle y_{k_n} , k_n-1\rangle$ reached before the

ARP terminates.

It is straightforward to show that Q_n defines the optimal action values for ARP at stage n. Let Q^*_{ARP} be the optimal action-value function for the ARP; that is $Q^*_{ARP}(<x,n>,a)$ is the optimal action value for action a at state $<x,n>$ of the ARP, and let U^*_{ARP} be the optimal value function of the ARP.

The 'Action-Replay' Theorem:

For all x, a, $Q_n(x,a)$ is the optimal action value at stage n of the ARP. That is,

$$Q_n(x,a) = Q^*_{ARP}(<x,n>,a)$$

for all x, a, and for all $n \geq 0$.

Proof:

By induction. From the construction of the ARP, $Q_0(x,a)$ is the optimal—indeed the only possible—action value of $<x,0>$, a. So

$$Q_0(x,a) = Q^*_{ARP}(<x,0>,a)$$

Hence the theorem holds for $n = 0$.

Suppose that the values of Q_{n-1}, as produced by the one-step Q-learning rule, are the optimal action values for the ARP at stage $n-1$, that is

$$Q_{n-1}(x,a) = Q^*_{ARP}(<x,n-1>,a)$$

for all x, a. This implies that $U_{n-1}(x)$ are the optimal values at the $n-1$th stage, that is

$$U^*_{ARP}(<x,n-1>) = \max_a Q_{n-1}(x,a)$$

Recall that Q_n is calculated from Q_{n-1} in the following way. Except for the value for the state-action pair x_n, a_n , at stage n, Q is unaltered by the learning procedure, so that $Q_n(x,a) = Q_{n-1}(x,a)$ for all x, a not equal to x_n , a_n . And

$$Q_n(x_n, a_n) = \alpha_n(r_n + \gamma U_{n-1}(y_n)) + (1-\alpha_n)\, Q_{n-1}(x_n, a_n)$$

Now, consider the nth stage of the ARP. For all x, a not equal to x_n, a_n , performing a in $<x,n>$ in the ARP gives exactly the same results as performing a in $<x,n-1>$; therefore,

$$Q^*_{\text{ARP}}(<x,n>,a) = Q^*_{\text{ARP}}(<x,n-1>,a)$$

for x, a not equal to x_n, a_n. Hence

$$Q^*_{\text{ARP}}(<x,n>,a) = Q_n(x,a)$$

for all x, a not equal to x_n, a_n respectively.

Now, consider the optimal action value of $<x_n,n>$, a_n in the ARP. Performing a_n in $<x_n,n>$ has the effect of

- with probability α_n, yielding immediate reward r_n, and new state $<y_n,n-1>$, or
- with probability $1-\alpha_n$, the same effect as performing a_n in $<x_n, n-1>$.

Hence the optimal action value in the ARP of $<x_n, a_n>$ is

$$Q^*_{\text{ARP}}(<x_n,n>,a_n) = \alpha_n\,(r_n + \gamma\, U^*_{\text{ARP}}(<y_n,n-1>)) + (1-\alpha_n)Q^*_{\text{ARP}}(<x_n,n-1>, a_n)$$

$$= \alpha_n\,(r_n + \gamma\, U_{n-1}(y_n)) + (1-\alpha_n)Q_{n-1}(x_n, a_n)$$

$$= Q_n(x_n,a_n)$$

Hence, by induction,

$$Q_n(x, a) = Q^*_{\text{ARP}}(\langle x,n\rangle,a)$$

for all x, a, and $n \geq 0$, which was to be proved.

3. Convergence of Q^*_{ARP} to Q^*

Under what conditions will the optimal action values for the action replay process at the nth stage converge to the optimal action values for the real process as $n \rightarrow \infty$?

Sufficient conditions are that for each state-action pair x , a :

- There is an infinite number of observations of the form [$x\ a\ r_n\ y_n$]
- The learning factors α_n for observations of the form [$x\ a\ r_n\ y_n$] are positive, decrease monotonically with increasing n, and tend to zero as $n \rightarrow \infty$.
- The sum of the learning factors α_n for observations of the form [$x\ a\ r_n\ y_n$] is infinite.

Note that it is required that the learning factors decrease monotonically for observations of the form [$x\ a$ . .] for each x, a—the learning factors need not decrease monotonically in along the sequence of observations.

To demonstrate that these conditions are sufficient, the method is to show that if one starts from the nth layer of the replay process, then the replay process will approximate the real process to any given degree of accuracy for any given finite number of stages, provided that n is chosen to be large enough. The replay process 'approximates' the real process in the sense that, for any k, the state $\langle x,k\rangle$ in the replay process corresponds to the state x in the real process; and actions and rewards in the replay process correspond directly with actions and rewards in the real process.

Let the *depth* of a state-action pair $d(\langle x,k\rangle, a)$ in the replay process be the sum of the learning factors for all observations of the form $[\ x\ a\ r_l\ y_l\ ]$ with $l \leq k$. If one follows the procedure for performing a in $\langle x,k\rangle$ in the replay process, the probability of reaching $\langle x,0\rangle$ becomes arbitrarily small as $d(\langle x,k\rangle, a)$ becomes large.

There are a finite number of state-action pairs, and according to the third assumption above, the $d(\langle x,n\rangle, a)$ tends to infinity as n tends to infinity. For any given D, and any given ε, it is possible to choose n such that

$$\max_{m>n} \{\ \alpha_m\ \} < \varepsilon$$

Given any such n, it is then possible to choose n' such that

$$\min_{x,a} \{\ d(\langle x,n'\rangle,a) - d(\langle x,n\rangle,a)\ \} > D$$

For any ε and any D, it is possible to choose a sequence of values $n_1\ ,\ n_2\ ,\ n_3\ ,\ \cdots$ such that the depths of *all* state-action pairs increase by D between each value of n in the sequence. It is, therefore, possible to choose an n so large that the minimum possible number of 'replayed' observations is larger than any chosen k with a probability as close to one as desired, and such that the maximum learning factor α is so small that the transition probabilities and reward means of the ARP are, with a probability as close to 1 as desired, uniformly as close as desired to the transition probabilities and reward means of the RP. Hence it is possible to choose an n so large that Q^*_{ARP} at the nth level of the ARP is, with a probability as close to 1 as desired, uniformly as close as desired to the corresponding optimal action values of the RP; and this is what needed to be shown.

References

Albus, J.A. (1981)
Brains, Behaviour, and Robotics
BYTE Books

Anderson, C.W. (1987)
Strategy Learning with Multi-Layer Connectionist Representations
Proceedings of the Fourth International Workshop on Machine Learning, University of California at Irvine, Morgan Kaufman

Barto, A.G., and Sutton, R.S. (1981)
Landmark Learning: An Illustration of Associative Search
Biol. Cybern. **42** pp1-8

Barto, A.G., and Anandan, P. (1985)
Pattern-Recognising Stochastic Learning Automata
IEEE Transactions on Systems, Man, and Cybernetics **15** pp360-374

Barto, A.G., Sutton, R.S., and Anderson, C.W. (1983)
Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems
IEEE Trans. Systems, Man, and Cybernetics, Vol. SMC-13, pp834-846

Bellman, R.E., and Dreyfus, S.E. (1962)
Applied Dynamic Programming
RAND Corp.

Bertsekas, D. (1976)
Dynamic Programming and Stochastic Control
Academic Press, New York

Browne, M.P. (1976)
The Role of Primary Reinforcement and Overt Movements in Auto-Shaping of the Pigeon
Anim. Learn. Behav. **4** *287-92*

Bush, R.R., and Mosteller, F. (1955)
Stochastic Models for Learning
Wiley, New York

Connell, M.E., and Utgoff, P.E. (1987)
Learning to Control a Dynamic Physical System
Proceedings AAAI-87, July 1987, pp 456-460

DeGroot, M.H. (1970)
Optimal Statistical Decisions
McGraw-Hill

Denardo, E.V. (1982)
Dynamic Programming: Models and Applications
Prentice-Hall

Dickinson, A. (1980)
Contemporary Animal Learning Theory
Cambridge University Press

Dreyfus, S.E., and Law, A.M. (1977)
The Art and Theory of Dynamic Programming
Academic Press

Dynkin, E.B. (1979)
Controlled Markov Processes
Springer-Verlag, New York

Feigenbaum, E.A., and Feldman, J. (1963)
Computers and Thought
McGraw-Hill, 1963

Ferster, C.B., and Skinner, B.F. (1957)
Schedules of Reinforcement
Appleton-Century-Crofts, New York.

Gould, S.J., and Lewontin, R.C. (1979)
The Spandrels of San Marco and the Panglossian Paradigm: a Critique of the Adaptationist Programme
Proc. Roy. Soc. London (B) Vol. 205 pp581-598

Hampson, S.E.
A Neural Model of Adaptive Behaviour
PhD Thesis, University of California, Irvine, CA, 1983

Houston, A.I., Clark, C., McNamara, J.M., and Mangel, M. (1988)
Dynamic Models in Behavioural and Evolutionary Ecology
Nature, Vol. 332, No. 6159, pp 29-34, 1988

Houston, A.I., and McNamara, J.M. (1988)
A Framework for the Functional Analysis of Behaviour
Behavioural and Brain Sciences **11** 1 pp117-163

Houston, A., and Sumida, B.H. (1987)
Learning Rules, Matching, and Frequency Dependence
Journal of Theoretical Biology, (1987), **126**, 289-308

Howard, R.A. (1960)
Dynamic Programming and Markov Processes
MIT Press

Koshland, D.E. (1979)
A Model Regulatory System: Bacterial Chemotaxis
Physiol. Rev. **59** pp811-862

Krebs, J.R., Kacelnik, A., Taylor, P. (1978)
Test of Optimal Sampling by Foraging Great Tits
Nature, Vol. 275, No. 5675, pp 27-31

Kumar, P.R., and Becker, A. (1982)
A New Family of Optimal Adaptive Controllers for Markov Chains
IEEE Trans. Automatic Control, Vol. AC-27, No. 1, February 1982

Laird, J., Rosenbloom, P., Newell, A. (1986)
Universal Subgoaling and Chunking
Kluwer Academic

Lakshimivarahan, S. (1981)
Learning Algorithms: Theory and Applications
Springer-Verlag

Liepins, G.E., Hilliard, M.R., and Palmer, M. (1989)
Credit Assignment and Discovery in Classifier Systems
In preparation.

Lippman, R.P. (1987)
An Introduction to Computing with Neural Nets
IEEE ASSP Magazine, pp4-22, April 1987

Mandl, P. (1974)
Estimation and Control in Markov Chains
Advances in Applied Probability **6**, pp40-60

Mangel, M. and Clark, C.W. (1988)
Dynamic Modeling in Behavioral Ecology
Princeton University Press

McNamara, J. (1982)
Optimal Patch Use in a Stochastic Environment
Theor. Popul. Biol. 21 pp269-288 1982

McNamara, J., and Houston, A.I. (1980)
The Application of Statistical Decision Theory to Animal Behaviour
Journal of Theoretical Biology, (1980), **85**, pp673-690

McNamara, J., and Houston, A.I. (1985)
Optimal Foraging and Learning
Journal of Theoretical Biology, (1985), **117**, pp231-249

McNamara, J., and Houston, A.I. (1987)
Memory and the Efficient Use of Information
Journal of Theoretical Biology, (1987), **125**, pp385-395

McNamara, J., and Houston, A.I. (1987)
Partial Preferences and Foraging
Animal Behav. **35** pp1084-1099 1987

Michie, D., and Chambers, R.A. (1968)
BOXES: An Experiment in Adaptive Control
Machine Intelligence 2, pp137-152, Oliver and Boyd 1968

Michalski, R.S., Carbonell, J.G., and Mitchell (Eds.) (1983)
Machine Learning: an Artificial Intelligence Approach
Tioga, Palo Alto, Ca. 1983

Mitchell, T.M., Utgoff, P., and Banerji, R. (1983)
Learning by Experimentation: Formulating and Generalising Plans from Past Experience
in 'Machine Learning: an Artificial Intelligence Approach', Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.) Tioga Publishing Company 1983

Narendra, K.S., and Thathachar, M.A.L. (1974)
Learning Automata—a Survey
IEEE Trans. Systems, Man, and Cybernetics, Vol. SMC-4, No.4, July 1974 pp323-34

Newell, A., and Simon, H.A. (1972)
Human Problem Solving
Eaglewood Cliffs, N.J.: Prentice-Hall.

Omohundro, S. (1987)
Efficient Algorithms with Neural Network Behaviour
Complex Systems, 1:273-347, 1987

Pearl, J. (1984)
Heuristics: Intelligent Search Strategies for Computer Problem Solving
Addison-Wesley

Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T. (1986)
Numerical Recipes: the Art of Scientific Computing
Cambridge University Press

Rubinstein, R.Y. (1981)
Simulation and the Monte-Carlo Method
Wiley

Rumelhart, D.E., and McClelland, J.L. (1986)
Parallel Distributed Processing: Explorations in the Microstructure of Cognition Vol 1: Foundations
MIT Press

Samuel, A.L. (1959)
Some Studies in Machine Learning Using the Game of Checkers
Reprinted in Feigenbaum and Feldman (1963)

Samuel, A.L. (1967)
Some Studies in Machine Learning Using the Game of Checkers II - Recent Progress
IBM J. Res. Develop. **11** (1967) 601-617

Selfridge, O.G. (1984)
Some Themes and Primitives in Ill-Defined Systems
in Selfridge et al, 1984

Selfridge, O.G., Rissland, E.L., and Arbib, M.A. (Eds.) (1984)
Adaptive Control of Ill-Defined Systems
Plenum Press, New York 1984

Shrager, J., Hogg, T., and Huberman, B.A. (1988)
A Graph-Dynamic Model of the Power Law of Practice and the Problem Solving Fan Effect
Science **242** (1988) pp414-416

Skinner, B.F. (1948)
Superstition in the Pigeon
J. Exp. Psychol. **38** *168-172 (1948)*

Stephens D.W., and Krebs, J.R. (1986)
Foraging Theory
Princeton University Press 1986

Sutton, R.S. (1984)
Temporal Credit Assignment in Reinforcement Learning
Doctoral Dissertation, Dept. of Computer and Information Science, University of Massachusetts at Amherst, 1984

Sutton, R.S. (1988)
Learning to Predict by the Methods of Temporal Differences
Machine Learning 3: 9-44, 1988

Sutton, R.S., and Barto, A.G. (1987)
A Temporal-Difference Model of Classical Conditioning
GTE Labs. Report TR87-509.2, March 1987

Sutton, R.S., and Pinette, B. (1985)
The Learning of World Models by Connectionist Networks
Proceedings of the Seventh Annual Conference of the Cognitive Science Society August 1985

Wheeler, R.M., and Narendra, K.S. (1986)
Decentralised Learning in Finite Markov Chains
IEEE Trans. Automatic Control Vol. AC-31, No. 6, June 1986

White, D.J. (1963)
Dynamic Programming, Markov Chains, and the Method of Successive Approximations
Journal of Mathematical Analysis and Applications **6**, pp373-376, (1963)

Widrow, B., Gupta, N.K., and Maitra, S. (1973)
Punish/reward: Learning with a Critic in Adaptive Threshold Systems
IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, pp455-465, 1973

Wilson, S.W. (1987a)
Classifier Systems and the Animat Problem
Machine Learning 3(2) 1987 pp199-228

Witten, I.H. (1977)
An Adaptive Optimal Controller for Discrete-Time Markov Environments
Information and Control, **34**, *286-295, 1977*