Reinforcement Sailing

Philip Jonathan Sterne

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2004

Abstract

This thesis examines applying reinforcement learning to sailing. We give two conceptually different models of a simple sailing boat. Standard tabular reinforcement learning is shown to be ineffective in controlling these naturally continuous models. We examine a method [Smith, 2001b] which adaptively quantises both the state and action spaces and show that it has similar performance to the tabular case but requires only a fraction of the resources. Finally we examine the continuous method of wire-fitting [Baird and Klopf, 1993] combined with advantage learning [Baird, 1995].

Our findings suggest that for tasks which require smoothly-changing actions in response to slowly-changing states (such as sailing) the best results are obtained by using a continuous method. Wire-fitting was found to have a slower convergence rate, but might form a good starting point for a yacht's autopilot.

Acknowledgements

Many people deserve acknowledgement for helping me through this year. First and foremost is Gillian Hayes who once a week managed to point out the glaringly obvious which until then I could not see. She also read through this thesis twice while it was still in its infancy and gave very useful feedback. George Konidaris also helped a lot in commenting on the original proposal, and also reviewing a draft of the thesis just days before moving to America.

Thanks to everyone at Kitchener House for ensuring a memorable year, even if they all felt that having a summer holiday was more important than hanging around me. (Not that I'm bitter at all!)

Finally I would like to thank Katie Mylonas who for the past month put up with all my sentences containing words such as 'thesis', 'reinforcement' and 'sailing', and yet never complained.

This research was made possible through a commonwealth scholarship (ref. ZACS-2003-335). I still find the concept strange that I have been funded to do something really interesting and exciting and am very grateful for such an opportunity.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Philip Jonathan Sterne*)

Table of Contents

Chapter 1

Introduction

Sailing is a complex interaction of forces generated by moving through both wind and water. Machine learning is the automated process of trying to see patterns inherent in data. This thesis looks at using sailing as a test problem and applying several reinforcement learning algorithms to it.

The sailing is modelled using a computer simulation, and it is not claimed that the final solutions found would be applicable to a real-world yacht. Moreover current sailing regulations forbid the use of automated systems in setting the sails, however there is still scope in creating an autopilot for a yacht.

In [Adriaans, 2003] a description of such an autopilot is given. This system has a hybrid architecture in which various sailing rules are provided by an expert. The system then learns the optimal parameters for these rules. Adriaans claimed that the task of sailing is a difficult one, with a vast array of sensor information making convergence of learning algorithms very slow.

This thesis argues that Adriaans discarded reinforcement learning prematurely. Instead of approaching the problem as a monolithic reinforcement learning problem, it should have been subdivided into behaviours corresponding to Adriaans' agents, several for each different time-scale. Reinforcement learning could then be applied to those behaviours which would benefit from such an approach[1]. It would have then

[1]We are not claiming that all such behaviours are appropriate candidates for reinforcement learning, for example the rules for determining right of way would be better encoded in a symbolic form, this would give one more confi dence that all possible variations would be correctly obeyed.

been possible to learn each problem in a reasonable amount of time. Moreover a computer simulation could provide a good first approximation of the policy to be learnt. This has the advantage of requiring very little expert knowledge, which makes the method more broadly applicable.

In this thesis we aim to explore one such layer in the architecture - the basic sailing layer. In this layer, we avoid complications such as obstacle avoidance, and obtain a starting point for adding other layers onto this architecture.

However sailing is a task which naturally requires continuous actions, and reinforcement learning for continuous states and actions only has weak guarantees for convergence. Moreover the rate of convergence is normally much slower than tabular reinforcement learning. In this thesis we explore the trade-offs in several methods while gradually increasing the complexity of the solution.

1.1 Structure of the dissertation

The following chapter provides some of the related work in the field of continuous state and action reinforcement learning, as well as examining Adriaans' system in slightly more detail. Chapter 3 provides a brief introduction to sailing and presents two models of a sailing boat. These models are compared and the performance of hand-coding a control strategy is presented. Chapter 4 examines the performance of the standard discrete methods of reinforcement learning when applied to these models and discusses their limitations.

In chapter 5 we present a method of adaptively quantising the state and action space using two separate self-organising feature maps. In Chapter 6 the final method of using a neural network to predict the reinforcement received. Finally in Chapter 7 we provide a conclusion and discuss the findings of this research.

Chapter 2

Related Work

2.1 Introduction

In this chapter we review much of the work concerning reinforcement learning in the domain of continuous states and actions, an area of research which incorporates ideas from standard mathematical control theory, standard reinforcement learning theory and approximation theory. We initially cover work done for similar problems such as learning to fly, or control an underwater autonomous vehicle. Research into reinforcement learning with either a continuous state or continuous action space but not both is presented. We then show some of the difficulties of learning in a continuous state and action space such as divergence. Lazy learning and policy gradient methods are explored at the end of this chapter. This thesis assumes the reader is familiar with Reinforcement Learning, if not [Sutton and Barto, 1998] or [Kaelbling et al., 1996] are both excellent introductions to the subject.

2.2 Learning Motion Control

In this section we review research attempted for similar problems, such as learning to fly, or the control of an autonomous underwater vehicle. We also review the approach taken by RoboSail, a company in the Netherlands, which are developing more intelligent autopilots for ocean-going sailing boats. In this thesis we do not cover related

work such as learning to fly a helicopter [Bagnell and Schneider, 2001], or the control of mobile robots [Smart and Kaelbling, 2002], although there is a significant amount of work relating to these problems. This exclusion is simply due to a matter of space and we felt that control problems covered here are more relevant.

2.2.1 The RoboSail project

Machine learning techniques have been applied in the domain of sailing already. In [Adriaans, 2003] the application of reinforcement learning to sailing was examined, however they reported very slow training times in [van Aartrijk et al., 2002]. Adriaans also argued that different aspects of sailing required different time scales of evaluation. As an example: while steering the boat requires sub-second reaction times, other tasks such as navigation only need to be evaluated hourly. This discouraged the authors and they instead pursued a hybrid approach.

In their approach, basic knowledge of sailing was used to form rule sets, each operating on a different time-scale. The rules were of the form:

> If the apparent wind angle[1] is between *x* and *y* then the sail should be set at *z*.

where *x*, *y* and *z* were defined by fuzzy rules. The initial estimates for *x*, *y* and *z* were supplied by expert knowledge and then fine-tuned through experience which took the form of a large database containing actual sailing episodes.

[Adriaans, 2003] split the task into four main agents, each operating on a different time-scale. They are as follows:

- *Skipper* - This agent is responsible for tactical decisions, it examines weather maps and tidal information to establish a goal way point. (This process is only partly automated, and requires input from the human skipper as well.) The *Skipper* makes decisions roughly every 3-6 hours.

- *Navigator* - This agent takes the goal way point set by the *Skipper* and the boat's current position, it then decides on a compass heading that is to be followed to best obtain this way point. Decisions at this level are made every 15-30 minutes.

[1] The apparent wind is the wind felt on the boat, and is described further in section 3.2.1

- *Watchman* - The *Watchman* uses the compass heading it is given as well as the current heading to determine a rudder target. This decision also takes into account other factors such as the current wind speed, and sailtrim[2]. At this level a decision is made every second.

- *Helmsman* - The *Helmsman* accepts the rudder target and the current speed of the boat. It outputs the force to be exerted by the motors. To achieve smooth control this is run ten times a second.

In this thesis we explore using reinforcement learning in what would be the *Watchman* layer. We use only a simple state description under the assumption that other layers could be added in a subsumption-style manner to modify the original behaviour when necessary. In this manner it is possible to use reinforcement-learning in an incremental fashion.

2.2.2 In the skies

In [Isaac and Sammut, 2003] a method of learning to fly a plane is presented. Rather than use reinforcement learning to perform *tabula rasa* learning they employ a method of learning from an expert. A person skilled in using the flight simulator guides a plane through set manoeuvres. This training data provides the basis for training several local controllers. Each controller is responsible only for a small part of the task, and a central coordinator determines which controller is active at any given time.

The learning is split into two areas, the learning of goal rules (i.e. how fast one should turn to reach a certain position) and learning of control rules (i.e. how one can achieve that turning rate). This decomposition requires an intimate knowledge of the relationship between all the variables, it is argued that this method allows a degree of robustness and transparency, which few other methods can offer. However the rules that these controllers find are difficult to interpret, consisting of multiply-nested if-then branches, with several seemingly-arbitrary learnt constants involved.

An impressive result is that the plane is able to perform novel manoeuvres, which had not been seen previously, such as a climbing (or descending) turn. This is a side

[2]The sailtrim refers to how the sails are set

effect of decoupling the control problem from the goal problem, as well as removing dependencies from irrelevant variables.

In order to faithfully reproduce the behaviour of the expert the control outputs have to be delayed to coincide with a human reaction time. This system is also shown to work in the presence of turbulence.

2.2.3 Under the water

[Gaskett et al., 1999b] shows a potential control system for an Autonomous Underwater Vehicle (AUV). While at the moment the system is still only in simulation, they do present results for a simplifed form of the problem. In this simplified task the vehicle operates on a two-dimensional plane, but still has to deal with problems such as momentum, so it is possible to overshoot the target. They use a method known as wire-fitting, which will be presented more thoroughly in chapter 6.

They also show that a method called 'Advantage updating' significantly increases the learning rate. Advantage updating is dealt with more fully in section 2.5.

Using these methods [Gaskett et al., 1999b] are able to obtain controllers which efficiently move from one way point to the next, although only half are successful in the learning phase, the unsuccessful controllers are discarded. This may obscure the true performance of this method as it could be the case that an unsuccessful controller may require many thousands of iterations before achieving reasonable control. However by discarding these, we remain ignorant of the learning performance in the worst case. It could be problematic if a successful simulation controller turns out to be an unsuccessful AUV controller.

2.3 Semi-Continuous Methods

This section covers some of the methods in which only one of the state space or the action space is continuous. While this is simpler than learning with both a continuous state and action space, it is still considerably more difficult than the tabular case of reinforcement learning.

2.3.1 CMAC

The CMAC (Cerebellar Model Articulation Controller) referred to in [Albus, 1975] is capable of outputting real valued actions from discrete input states. It does this by maintaining many overlapping tiles. When an input is provided it activates a subset of these tiles. The result returned is the sum of the output of all the activated tiles. If a high amount of resolution is required then the tiles are usually hashed, to avoid having to store all possible combinations of the tiles.

The input space could be continuous, but then it has to be divided into a discrete set by hand. See [Santamaria et al., 1998] for an example in which the authors introduce some *apriori* knowledge by using a 'skewing' function to enable higher resolution in critical areas of the state space.

2.3.2 Gullapalli's SRV unit

Gullapalli [Gullapalli, 1992] has designed an algorithm called the Stochastic real-valued algorithm, for problems involving discrete states, but with continuous actions. Gullapalli makes the additional assumption that the maximum reward which can be received in any state is known. This enables him to decrease exploration as the reward approaches the maximum. In doing so he also manages to avoid all of the problems discussed above.

The basics of his method are as follows: for every state there is a parameter to estimate the mean of the best action as well as an estimate of the required exploration variance. The action that is chosen is distributed according to Gaussian distribution with the determined mean and variance. If the reward returned is greater than the current expected reward the mean is then updated towards the chosen action. As the reward approaches the maximum the variance is decreased appropriately.

Unfortunately Gullapalli's assumption that the maximum reward is known for every state limits the applicability of his method. Moreover problems which naturally require continuous actions also tend to require continuous states, a further limitation of the theory.

2.3.3 Neural Q-Learning

In [Touzet et al., 1997] the task of robot collision avoidance is explored. Rather than do any pre-processing on the robot's data (which is huge, the state space size $\approx 10^{24}$ and there are 400 possible actions), a self organising map (SOFM) is used to cluster the tuple (state,action,reward). When a novel situation is encountered the best node is chosen as being a reasonably close node with the best predicted reward. The action from this node is carried out and the SOFM is updated. There is also a small probability of a completely random action (i.e. ε-greedy exploration).

While this approach did seem to work, the authors did encounter some problems in getting the robot to perform as expected. The reinforcement function they used simply encouraged getting the obstacle out of sight as quickly as possible. The robot discovered that sometimes the quickest way is to reverse, however once the obstacle is out of sight the default behaviour of moving forward occurs and the obstacle is encountered again. To get around the problem the authors expressly forbid certain sequences of actions. This is a rather clumsy solution and a much better approach to reinforcement learning using a SOFM due to [Smith, 2001b] is presented in section 5.

2.4 Continuous State Reinforcement Learning

One of the main problems in extending Reinforcement Learning theory to continuous states is the problem of divergence. For problems with continuous states some sort of generalization is required. However many of the theoretical guarantees rely strongly on each state being experienced many times, not merely similar states. Thus many of the guarantees on convergence fall away, and in some cases divergence has been observed.

Divergence in some cases hasn't stopped researchers from exploring these methods. Many researchers have tried using a function approximator to estimate the value of the current state. The best known success is Tesauro's Backgammon player (TD-Gammon) which has learnt to play at grandmaster level. [3] This was achieved through

[3] While Backgammon is not a problem involving continuous states or actions the size of the state-action space is prohibitively large, thus requiring the generalisation needed for continuous reinforcement

repeated self-play with the information used to update a three layer back-propagation network. See [Tesauro, 1995] for more details.

However in [Boyan and Moore, 1995] several straightforward examples are shown which reliably diverge using a variety of different function approximation schemes. The simplest case of proven divergence is Baird's counterexample.

2.4.1 Baird's Counterexample

Baird's counterexample doesn't deal with a continuous state directly, instead it looks at the use of generalisation in a simple discrete case. The simplest form of such generalisation would be to use a linear function of the features present in a state to predict the value of the state and a gradient descent update rule, i.e.

$$\begin{aligned} V(x) &= \theta_i^T x \\ \theta_{i+1} &= \theta_i + \alpha \sum_s [E\{r_{t+1} + \gamma V_t(s_{t+1}) | s_t = s\} - V_t(s)] \nabla_{\theta_t} V_t(s) \end{aligned}$$

However in this simple case for certain discrete problems the estimated value can diverge, even though the linear function could represent the true value exactly. See [Sutton and Barto, 1998] (page 216) for more details. The lack of convergence for this simple case, is worrying and there are other cases where divergence occurs [Baird, 1999].

2.4.2 The Grow-Support algorithm

A solution [Boyan and Moore, 1995] propose for the problem of divergence is called the Grow-Support algorithm. This algorithm is only suitable for episodic problems where there is no noise in the reward function and a small number of actions. This algorithm maintains a set of sampled states for which the value function has been accurately calculated, (termed the set of support states). This set is then grown by randomly sampling the states and following the current greedy policy. If the action leads to a state which is within the support of a known state then its cost to goal can

learning.

be calculated explicitly, and it is included in the set of support states. The function approximator is then trained on the support states again.

This model is limited in the implicit assumption that the task is episodic. The assumption that a model of the environment is available and is noise-free also limits the applicability of this research.

2.4.3 Overestimating the Q-values

The problem of divergence partly occurs as a result of bootstrapping. This refers to the process of learning a new estimate of a state based on the estimates of other states. In [Thrun and Schwartz, 1993] reasons are given why bootstrapping can lead to divergence. In this paper they consider a reinforcement learning problem where reward is given only at the end of an episodic task. As a result the values of the initial states don't differ by much, as the decayed future reward is small. Taking a non-optimal move will result in only a small penalty. If the function approximation scheme introduces random, unbiased noise, then under certain conditions we can expect Q-learning to fail.

This is as result of the max operator which introduces bias into the unbiased noise. However this biased noise can easily dominate the useful information. In this case the network is expected to fail, and will not improve beyond a random controller.

2.5 Advantage updating

The problem of overestimation increases when the values of the possible actions don't differ by much. In this case even discrete Q-learning suffers from long training times. In [Baird, 1993] Baird argues that as a control problem approaches continuous time (i.e. the time between action selection decreases) then the cost of choosing a suboptimal action approaches zero. As it does so the time required to train a discrete controller increases exponentially. In [Baird, 1993] a solution is proposed where one learns 'advantages', which in essence represents the derivative.

Mathematically, as

$$\lim_{\Delta t \to 0} Q(s,a) \to max_{a'} Q(s,a')$$

which implies that $\forall_{a,a' \in A} Q(s,a) - Q(s,a') \rightarrow 0$, differentiating between the optimal and worst possible action becomes impossible. However if we define an advantage as:

$$A(s,a) = \lim_{\Delta t \rightarrow 0} \frac{Q(s,a) - max_{a'} Q(s,a')}{\Delta t} \tag{2.1}$$

where Δt is the time step then it does not converge to zero for all actions in the state. Advantage updating has been shown to have constant convergence time in simulations where the time step approaches zero [Baird, 1993]. Moreover since the advantage function does not approach zero for all actions there is more chance of representing it reasonably accurately with a function approximator (although divergence is still possible).

However one consequence of equation 2.1 is that the maximum advantage in any state is zero. This makes it hard to determine which states are more desirable than others. An estimate of the next state's value is also required to update the advantage estimate. Originally Baird proposed to learn the value function as well as the advantage function. However it is possible to modify the rule so that the maximum value is unchanged. We discuss this form of advantage learning in more depth in section 6.3.

2.6 Residual Methods

Residual methods are promising as they are only slightly different formulation of the problem, and they have convergence proofs. We briefly sketch a derivation of the formula below, by considering the Q-learning update (the interested reader should consult [Baird, 1999] and [Baird, 1995] for a more in-depth derivation).

Assume we have a parameterisation of the Q-values as $Q(x,u,w)$ where w is a vector of weights. A direct method attempts to minimise the temporal difference error directly:

$$\Delta w = \alpha \left(R + \gamma \max_{u'} Q(x',u') - Q(x,u) \right) \frac{\partial Q(x,u)}{\partial w} \tag{2.2}$$

however this can lead to instability and divergence as mentioned earlier. Rather we try to minimise the mean squared Bellman residual which is defined as:

$$E = \frac{1}{n} \Sigma_x \left[\langle R + \gamma \max_{u'} Q(x',u') \rangle - Q(x,u) \right]^2$$

This gives rise to the residual update:

$$\Delta w = \alpha \left[R + \gamma \max_{u'} Q(x', u') - Q(x, u) \right] \left[\frac{\partial}{\partial w} \gamma \max_{u'} Q(x', u') - \frac{\partial}{\partial w} Q(x, u) \right] \quad (2.3)$$

For this update convergence can be guaranteed to a local minimum of the Bellman residual. However the convergence can be very slow, to overcome this Baird proposes a weighted average of 2.2 and 2.3. If one is careful with the weighting factor then one can guarantee that the update is never away from the direction suggested by eqation 2.3. Thus the guarantee of convergence still holds.

This learning rule can be applied to other forms of learning such as advantage learning. In chapter 6 we use a neural network trained on targets which have been modified using a form of the above update, but modified to work on advantages rather than *Q*-learning.

2.7 Lazy Learning

A method of avoiding any problems associated with function approximation is to simply store each datapoint in a database. When one needs to query a new point one can search the database for all similar points previously encountered. One can then use a weighted average or fit a simple linear model to the similar points. This method is known as lazy learning (for a good introduction see both [Atkeson et al., 1997a] and [Atkeson et al., 1997b]).

This method is not as naïve as it first appears. It is possible to keep only sufficiently different states, which can reduce the storage requirements greatly. Also by using k-d trees one can reduce the required lookup to $O(n \log(n))$. This method is used in [Atkeson et al., 1997b] to control a devil sticking robot.

In [Smart and Kaelbling, 2000] a new algorithm called HEDGER is presented. This is a lazy learning reinforcement algorithm, which prevents extreme predictions by only interpolating the data rather than extrapolating the data, in this way they limit any extreme predictions made by the algorithm. They calculate the Independent Variable Hull of the data to see if the new point is within any points which have been experienced. Since the method is lazy they store all the data points previously seen in

a k-d tree structure, which is useful for both calculating the predicted control action and calculating the independent variable hull. Smart and Kaelbling show that learning is considerably faster than for the discrete case, as a result of correctly interpolating the data. They also show that calculating the independent variable hull is a necessary part of the algorithm as it can lead to divergence without it.

It is possible to use other approximation techniques to avoid storing the data, by fitting several local linear planes to regions of space which have been experienced. This is known as Receptive Field Weighted Regression (RFWR - see [Schaal and Atkeson, 1997]). The error for each plane approximation can be bounded, and the algorithm is constructive adding a new plane only when necessary. Moreover since each plane is independently trained and only focuses on local data it is possible to avoid the interference which occurs in most other approximation schemes such as neural networks.

2.8 Conclusion

In this chapter we reviewed a large section of the work relevant to this thesis. This included similar control tasks such as learning to fly or control an underwater vehicle. There is also a large amount of research into extending reinforcement learning into the continuous domain. Unfortunately a straightforward extension can lead to divergence, and some of the reasons for this divergence have been examined. Several modifications have been proposed to retain the guaranteed convergence of the tabular case.

These methods include lazy learning and residual methods. Lazy methods work as they lack the problem of interference which is common to most function approximation schemes. Residual methods are guaranteed to converge to a local minimum of the Squared Bellman residual, however the convergence times can be very slow. In the following chapter we examine the sailing task more closely.

Chapter 3

The Sailing models

3.1 Introduction

In this chapter we briefly introduce the basics of sailing and some sailing terminology. We go on to describe in detail the two models of a sailing boat which are used in the reinforcement learning algorithms. For each model we describe: the calculation of the forces involved; the state representation and the performance of a hand-coded controller. The chapter ends with some of the mathematics involved in transforming the calculated forces into motion.

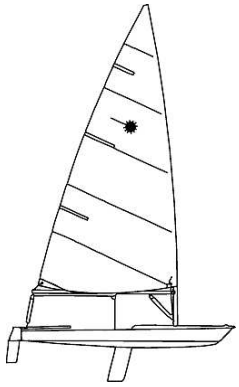
3.2 Introduction to sailing

3.2.1 Parts of a boat

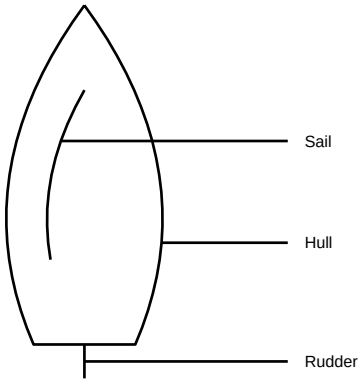
For readers unfamiliar with sailing we will briefly introduce the major parts of a sail boat. The boat shown in figure 3.1(a)[1] is the Laser, one of the most popular boats internationally, it is the basis of our model presented later on in the chapter. For the purposes of our models only three parts of a boat are relevant: the sail, centreboard[2] and rudder.

[1] Reproduced with permission from `AtlantaYachtClub.org`

[2] In this thesis the terms centreboard and keel will be used interchangeably, although there is a difference between them: a centreboard can be lifted, while a keel is immovable. They do perform the same function however and this distinction is irrelevant here.



(a) The basic parts of a boat, with A - Mainsail (or just sail), B - Rudder, C - Centreboard (or keel)



(b) A symbolic view of a boat from above. The centreboard is not shown as it is immovable (for the purposes of this thesis at least)

Figure 3.1: Overview of a boat

The Laser has a single sail (the mainsail) which is controlled by a single rope (the mainsheet). The wind determines which side of the boat the sail is on, while the mainsheet controls how far in or out the sail goes (the mainsheet cannot control which side the sail is on). The centreboard is used to ensure the boat isn't blown off course when the wind is blowing from the side. The rudder is used to control the direction. Turning the rudder when the boat is moving creates a large side force from the displacement of the water. However this force is only generated while the boat is moving forward (it is a passive control). As the boat increases its velocity the rudder's effectiveness then increases. If the boat is moving fast, the rudder only needs to be moved slightly to change direction. The wind felt on the boat also changes as it picks up speed (known as the apparent wind). The apparent wind is the difference between the true wind and the boat's velocity. A similar effect can be observed when driving, if one places a hand out the window one can feel a strong wind, though obviously this wind is mainly from the movement of the car, rather than the true wind.

Sailing is a non-holonomic control problem, as we are unable to turn on the spot.

This makes it a difficult problem to learn how to approach a target. In this thesis we ignore this problem of approaching a target and instead concentrate on sailing in a given direction. As we later show, if the controller is sufficiently good at sailing in a target direction, then it is able to sail a course, which involves approaching several targets. This is as a result of the targets being sufficiently far away from each other, so it is irrelevant if the boat cannot turn on the spot.

3.2.2 Theory of a sail

There is a lot of misinformation as to how a sail actually works, the most widespread theory being that the sail is curved similarly to a wing. As the air passes over the leeward side of the sail it has to travel a greater distance and so has to speed up. This increase in speed results in a decrease in pressure, which creates the force. This explanation however is *wrong*. In fact :

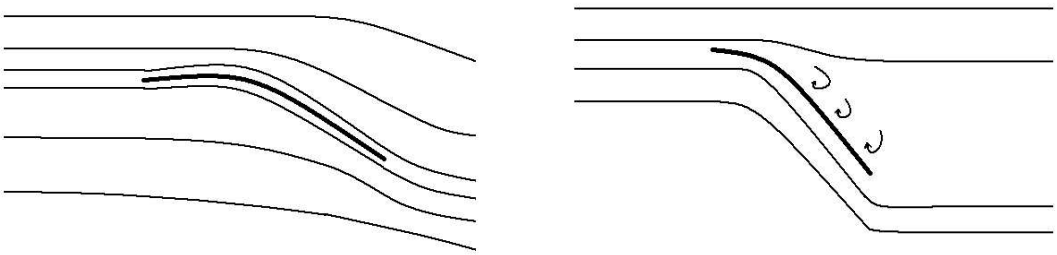
> ...when any wing is operating at peak efficiency, the air particles that split at the leading edge do not rejoin at the trailing edge; the particles travelling over the upper surface arrive at the trailing edge well before the particle on the underside of the wing. The same thing happens around a sail. [Ross, 1975]

If one bears in mind the following facts:

- Air at low speeds is considered incompressible.
- Air will flow in a reasonably straight line, while being attracted to low pressure regions.
- As air flows around a curved surface it will try to adhere to the convex surface.
- Air speeds up as it flows through a restricted area (as discovered by Venturi).

As the air flows around a sail it will try to adhere to the sail. However the undisturbed air slightly further out will try to move in a straight line, and can be thought of as a barrier. This means that the air is forced between a narrow channel and since it is incompressible it must speed up. However once it starts speeding up it creates more low pressure on the leeward side of the sail. (This is due to Bernoulli's theorem

which states that fast-flowing air has a lower pressure than the surrounding air.) This is the start of a chain reaction; as the low pressure develops it attracts more air from the windward side, thus forcing the air to move faster and lower the pressure further. However this does not go on indefinitely as the undisturbed air also moves in to fill this low pressure.



(a) The air flows from left to right. A low pressure develops above the wing (or sail). This creates an upward force. (Note also the downward displacement of the air - this creates an upward force as well.)

(b) If the angle of attack is too large then the air cannot adhere to the topside of the wing. As a result turbulent flow develops and most of the low pressure region is destroyed.

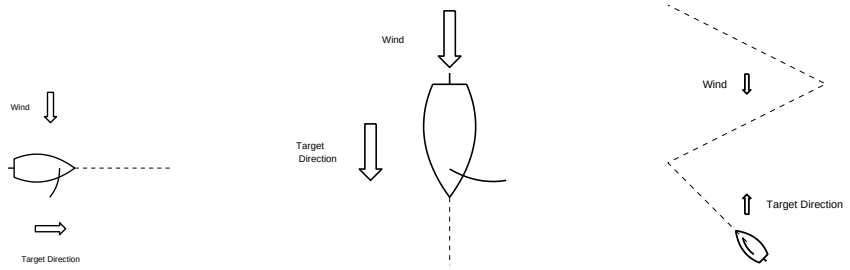
Figure 3.2: Wind flow over a sail

If the angle of attack is too great then air cannot adhere to the sail. The airflow on the leeward side becomes turbulent, and the low pressure region is not nearly as strong. For a wing this is known as a stall, and it can be quite dangerous. When a sail stalls there is a marked decrease in speed, however sometimes this is necessary (see the item 'Running' in the next section).

3.2.3 Basic control of a boat

Depending on the direction in which one wants to sail one needs different control strategies. Here we list the three basic strategies required:

1. *Reaching* This occurs when the desired direction is roughly 90^o to the wind. It is fairly easy to control the boat, and the boat can sail there directly in a reasonably



(a) Reaching (b) Running (c) Beating

Figure 3.3: The three different sailing behaviours

fast manner. (In fact some high-performance boats can reach faster than the wind, due to the increased apparent wind.)

2. *Running* If the target direction is directly downwind of the target then the boat can sail there directly. However the boat doesn't go particularly fast as it is sailing away from the wind (i.e. as it picks up speed, then the apparent wind decreases). The speed also decreases as the sail cannot act as an efficient wing, but must generate its force through drag. As the sail is let out all the way the force generated by the wind is entirely on one side of the boat. This can make steering the boat difficult.

3. *Beating* When the target is almost directly into the wind then a boat cannot sail to the target directly. Instead it sails roughly 45^o to one side of the wind and then turns[3] to sail on the other side of the wind. We can expect this to be the most difficult aspect of sailing to learn (as it is in real-life). This is because beating efficiently requires precise setting of the sail, as well as good control of the direction of the boat. If one sails too close to the wind, then not enough lift is generated and the boat sails slowly. If one sails too far from the wind then one doesn't make any progress towards the target.

[3]This turning is known as 'tacking'.

3.2.4 Continuous states and actions?

As a control problem, sailing warrants continuous states and actions. There are several reasons for this:

- In an ideal solution the rudder is to be used as little as possible. This is because every time the rudder is turned, it turns the boat but also acts as a brake, slowing the boat down. (The energy to turn the boat comes from the kinetic energy of the boat.) We are unable to interleave large discrete actions to approximate a continuous action as suggested by [Smith, 2001b] (see pages 7-8) as a large amount of energy is lost in the process.

- The lift generated by the sail is very sensitive to the angle of attack. Changes of less than 5^o can result in a significant loss of lift, and the boat will sail a lot slower. If the variables were to be discrete and have sufficient resolution the space requirements would be huge.

While both the rudder and sail require fairly sensitive control, we can still hope to learn the correct actions in a given context. This is because the performance varies in a fairly linear manner, as we approach the optimal action the reward increases in a fairly linear manner. However we will still have to deal with the problem of delayed reward; for example in both of our models if we are currently sailing slowly away from the target then the best action is to build up speed so that the rudder becomes effective, and we are then able to turn and sail directly towards the target. A short term solution would rather use the rudder as much as possible to slow down the boat, so that we sail away from the target as slowly as possible.

3.3 Timin's Model

The model consists of a single sail and keel, which are able to turn independently and was taken from the AnnEvolve website [Timin, 2004]. It was designed by Mitchell Timin for use in research into evolving a neural network controller. Originally the sailing model had to learn to sail around a circular island, however we modified the task so that it had to sail in a given direction. The group was able to evolve a controller

which sailed around the island in 90% of the trials. The original model also had a shifting wind, which made the task more complex. We chose to use a steady wind model, since we can rotate the boat and target direction until the wind is aligned to north. This allowed an easier comparison of the difficulty in learning the two models.

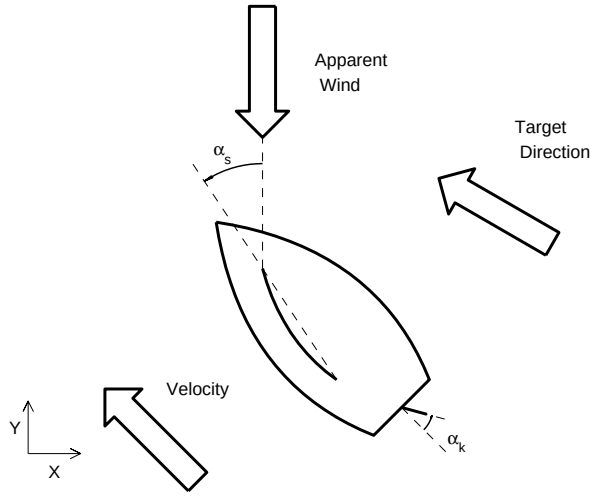


Figure 3.4: The AnnEvolve Model. α_s and α_k stand for the angle of attack in the sail and rudder respectively.

In this model the heading of the boat is derived from the direction of the velocity of the boat (i.e. the boat is always assumed to be moving forwards). The boat changes its heading through the use of the sails and rudder, which both generate forces parallel and perpendicular to the path of the boat. The forces change the velocity, which in turn changes the heading of the boat. An unrealistic side-effect of this is that the boat is easily maneuvered at low speeds (since the direction of the velocity can change quite drastically if there is only a small amount of momentum). It is still easy to maneuver when traveling fast, as the forces generated by the keel and sail increase with the square of the velocity (i.e. proportionally to the kinetic energy). The state to action mapping is as follows (with the ranges given in brackets) :

State		Action	
Speed	$[0, 0.5]$	Rudder	$[-\frac{\pi}{4}, \frac{\pi}{4}]$
Relative wind	$[-\pi, \pi]$	Sail	$[-\frac{\pi}{2}, \frac{\pi}{2}]$
Directional error	$[-\pi, \pi]$		

The Speed is the norm of the velocity. The Relative wind is the difference in angles between the target direction and the relative wind measured in radians. The directional error is the difference between the heading and the target direction. Originally a different state representation was used which contained only the x, y velocities, and the target direction. However all the learning methods tested here performed poorly on this state representation. The current state representation was derived by examining the hand-coded controller to find measures which were known to be useful in determining the action to be taken.

The Rudder is measured as the angle of attack (i.e. an angle relative to the moving water). The sail's angle of attack is specified with respect to the apparent wind, however with a larger range.

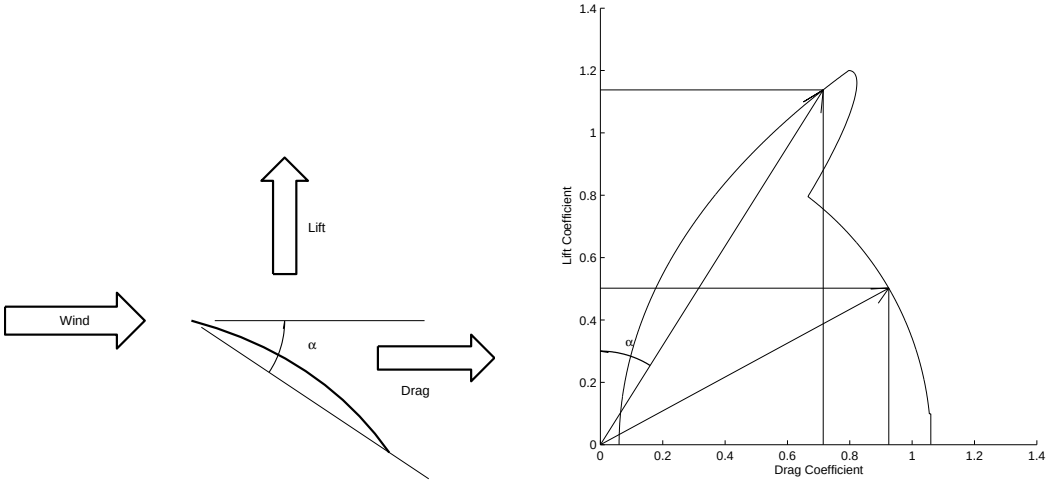
The angles of attack are converted into forces parallel (drag) and perpendicular (lift) to the wind and water (see figure 3.5(a). The drag and lift coefficients are calculated using the polar diagram shown in 3.5(b) and are multiplied by the sail area (or rudder area). The force is then:

$$F = c \times m \times V^2 \times A \tag{3.1}$$

where:

- c is the coefficient (lift or drag)
- m is a coefficient representing the density of the medium through which the sail/rudder travels. For air this is 1.226 kg/m^3, water is 1000 kg/m^3.
- V is the velocity.
- A represents the area of the sail or rudder.

(from [Marchaj, 1964] page 70).



(a) As the wind is diverted by the sail, lift and drag are generated.

(b) The coeffi cients of lift and drag as dependent on the angle of attack (α)

Figure 3.5: Calculating the resultant forces.

To obtain the movement of the boat, the resultant forces must be calculated. These are then integrated, since this integration is similar for both models we present it in a separate section after the models (see section 3.5). For Timin's model the forces are calculated as follows:

1. Specify the angle of attack for the rudder and sail.
2. Find the lift and drag coefficents for both the rudder and the sail.
3. Calculate the forces using equation 3.1 (these forces are assumed to go through the centre of mass, so there is no torque component to the resultant).
4. The heading of the boat is derived from the velocity.
5. The forces are transformed from boat coordinates to world coordinates, rotating them by the heading of the boat.
6. The forces are integrated, using the Adams-Bashforth update (given in section 3.5). This updates the position and velocity of the model.

7. The new heading of the model is computed from the velocity.

To transform the angles of attack into actual forces estimates of the boat's measurements were needed. We modified Timin's original figures to give a model which was slightly more manoeuvrable (by lowering the mass). This meant that the characteristics of both models were similar. The measurements were as follows:

Measure	Measurement
Sail Area	100m^2
Rudder Area	6m^2
Boat Mass	$10,000\text{kg}$

Initially we had hoped to be able to learn the tacking behaviour required for beating, by simply specifying a direction which was too close to the wind. Unfortunately none of the learning methods surveyed could handle this problem. All would be able to head somewhat into the wind, however the drag of the wind would continually slow them down, until their velocity was away from the target, however since heading is dependent on velocity, the boats would end up sailing away from the target. In some cases they might recover, and head towards the wind again however they would be losing ground, rather than gaining ground. To overcome this we specified the required headings manually, so the headings would never be pointing less than 45^o into the wind.

3.3.1 Hand-coding a controller

In order to provide a baseline performance we hand-coded a controller. This proved quite time-consuming as changing the policy often had unintended consequences. This was in part due to the simplifications the model made which made the task less intuitively like sailing. As an example we discovered the best method of tacking involved moving the sail rather than the rudder. Nevertheless we obtained a reasonably simple controller which had good performance.

The actions of the controller are calculated as:

$$\alpha_s \Leftarrow -0.4 \times \texttt{RW}$$

$$\alpha_r \Leftarrow \begin{cases} 0 & \text{if } \|\mathtt{DE}\| < \frac{pi}{9} \\ \text{sign}(\mathtt{DE})\frac{\mathtt{DE}^2}{2.3} & \text{otherwise.} \end{cases}$$

where DE stands for Directional error, and RW stands for the relative wind. By using the rudder only when the directional error was large we were able to sail quite fast, and still steer the boat using the sail. We used the square of the directional error when we did use the rudder so that if we were far from our target heading we would use a hard rudder turn. Also this controller is not optimal as we do not use the speed information, which does affect the handling of the boat.

3.3.2 Evaluating performance

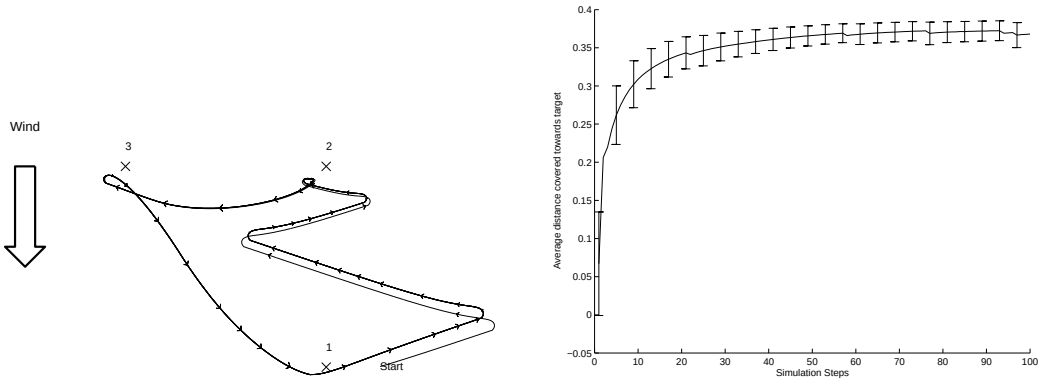
To test the performance we sailed a simple triangular course with it, see figure 3.6(a). In this course sailing from buoy 1 to buoy 2 is directly into the wind, which requires tacking. While our controller is capable of doing this, it does not do it particularly well if one notices how far away from the wind the boat has to sail. Also in moving from buoy 2 to 3 the controller does not sail a straight line.

However the boat is able to maintain a good speed, while sailing the course. This is shown in figure 3.6(b). To obtain this graph we randomly started the boat in 100 random positions far away from a target, the reward for 100 simulation steps was recorded. The average distance covered towards the target direction in each simulation step is plotted.

3.4 The OneSail Model

This model is called the OneSail model as initially we were thinking of having an additional model with two sails (to investigate which learning methods scaled better). As it turned out a single sail is hard enough to control by itself! This model consists of a sail, a keel and a rudder.

The keel is modelled as an immovable rudder, slightly larger than the rudder and fixed at the boat's centre of mass. Unlike Timin's model we use different 'polar' diagrams for the rudder and the sail (see figure 3.8). However in practise the difference was not noticeable. The data we obtained for the polar diagrams is from



(a) A trace of the route followed by the hand-coded controller. The X's represent target buoys which are changed when the boat is suffi ciently close to it.

(b) The average performance of the hand-coded controller

Figure 3.6: The hand-coded performance for Timin's model.

[Marchaj, 1964] (page 72). We measured the coefficients in 10 degree intervals and used a cubic spline to interpolate the points. To calculate the resultant force we use the spline to convert the angle of attack into lift and drag coefficients

A state description of the OneSail model is as follows, with the range of the states given in brackets[4] (see figure 3.7):

State		Action	
Forwards Velocity	$[-0.3, 0.7]$	Rudder (α_r)	$[-\frac{\pi}{4}, \frac{\pi}{4}]$
Lateral Velocity	$[-0.1, 0.1]$	Sail (α_s)	$[0, \frac{\pi}{2}]$
Relative Heading (θ)	$[-\pi, \pi]$		
Target error (γ)	$[-\pi, \pi]$		
Angular Momentum($\dot{\theta}$)	$[-0.3, 0.3]$		

In this model all the state information is given relative to the boat. This ensures that the learning algorithms learn with what would be available on a real boat. (Although we are not claiming this simulation is sufficient for a real yacht's autopilot.) The

[4]The ranges for the forwards velocity, lateral velocity and angular momentum are the extremes observed when running the model, there is nothing explicitly limiting these values.

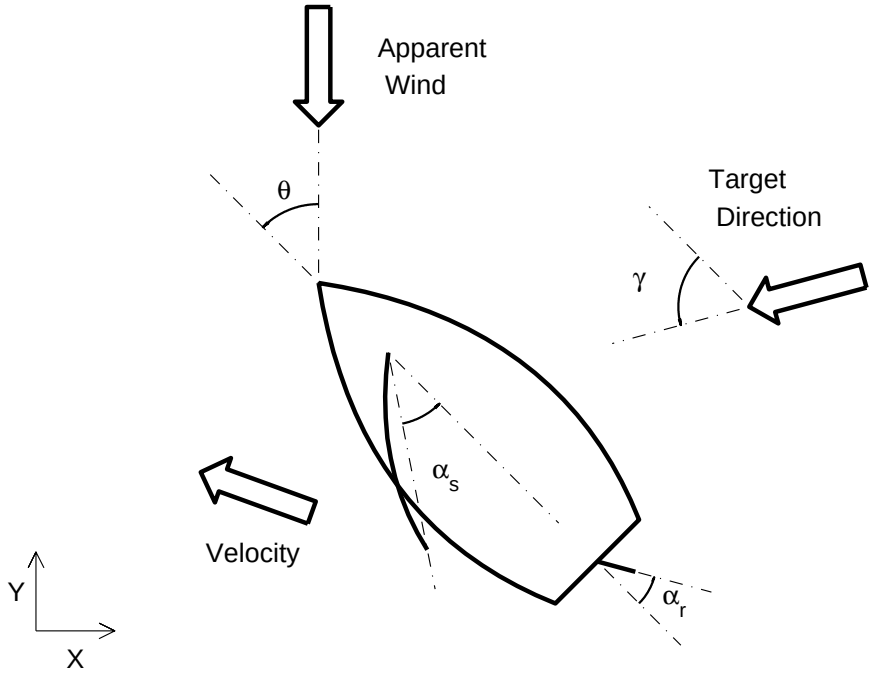


Figure 3.7: The OneSail model, note that the measuring of α_s and α_r have changed. (The keel is not shown in this diagram)

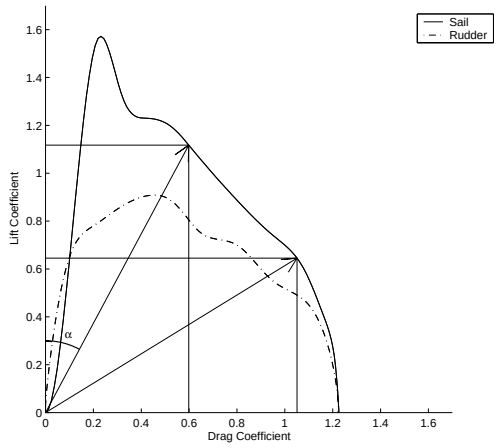


Figure 3.8: Converting the angle of attack into forces

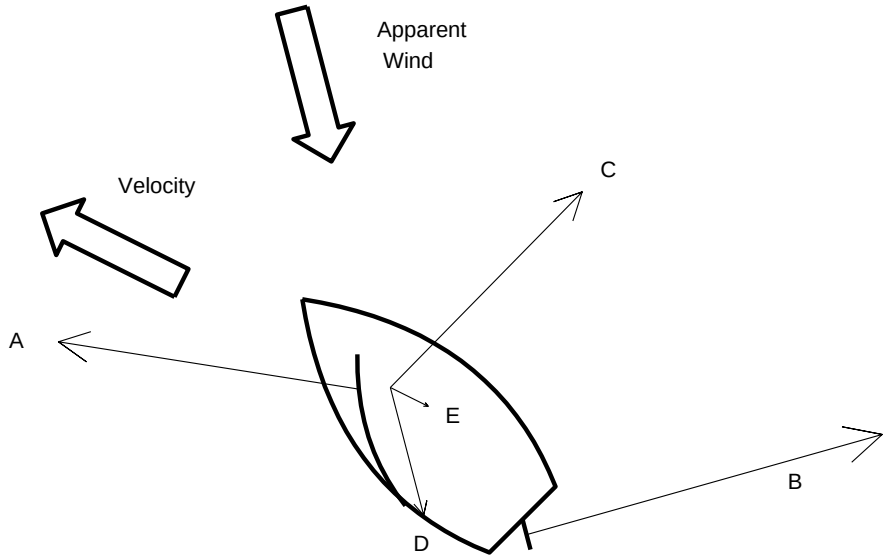


Figure 3.9: Forces involved in the OneSail model

velocity is given as forwards velocity and lateral velocity (since the wind is blowing from the side all boats drift sideways by a small amount, the keel tries to prevent this sideways drift). Knowing the forwards velocity is essential to smooth control since the rudder should be used less at high speeds. The relative heading (θ) is the heading of the boat with respect to the apparent wind. The target error (γ) is the angle from the boat's heading to the target direction. Note that the rudder (α_r) and sail (α_s) are specified relative to the boat, this is in contrast to Timin's model, where the angles are specified relative to the velocity of the medium in which they are travelling.

This model concerns the interaction of five forces (see figure 3.9):

- The sail, which provides the driving force (A).
- The rudder, which provides the steering force (B).
- The keel, which ensures the boat moves in a straight line (C).
- Wind resistance, which is proportional to the square of the apparent wind (D).

- Water resistance which is proportional to the square of the boat's speed (E).

The last three forces are all assumed to operate on the boat's centre of mass and do not contribute at all to the angular momentum. Thus to maintain a heading a controller must focus on balancing the forces of the rudder and sail. We must also include the angular momentum if we wish to learn how to tack[5] correctly. In some situations we might not have enough angular momentum to complete the tack (which is roughly a 90^o turn, most of which requires angular momentum).

To calculate the boat's movement, the forces involved are calculated as follows:

1. Given the angles for the rudder and sail, calculate the angle of attack with respect to the wind and water.

2. Use the velocity and the angle of attack to calculate the resultant sail and rudder force.

3. The positions of the sail and rudder determine the point of application of these forces.

4. Use the difference between the heading and the velocity as an angle of attack for the keel. This force is applied to the boat's centre of mass.

5. Use the relative wind and the boat's velocity to model aerodynamic and hydrodynamic drag. (Also applied to the boat's centre of mass.)

6. Integrate these forces to obtain the new velocity, position, heading and angular velocity (see section 3.5).

The measurements for this model were based on the measurements of a Laser sailing dinghy :

[5] A 'tack' is the name of the turn required when beating

Measure	Laser Measurement
Boat Length	3.81m
Sail Area	7.1m^2
Rudder Area	0.1m^2
Keel Area	0.187m^2
Boat Mass	56.7kg

3.4.1 Hand-coding the OneSail controller

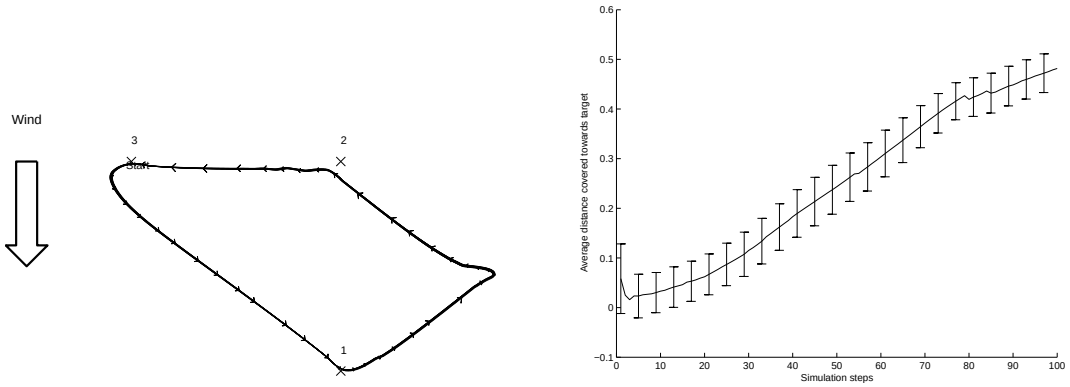
Again as a test of performance we developed a hand-coded controller for this model. This proved somewhat easier as the model is closer to a real sailing task and therefore more intuitive to tweaking. Thus we were able to produce a controller which is near-optimal. In figure 3.10(a) we show its performance on the same course as for the AnnEvolve controller.

The controller calculated its actions as follows:

$$\begin{aligned} \kappa &\Leftarrow \begin{cases} 3 & \text{if } \|\gamma\| > \frac{\pi}{10} \\ 1.5 & \text{otherwise} \end{cases} \\ \alpha_r &\Leftarrow \alpha_r + \gamma + \kappa\dot{\theta} \\ \alpha_s &\Leftarrow \|\theta\| - 0.25 \end{aligned}$$

Note that the update rule for the rudder depends on the previous value of the rudder. In a sense this is cheating as the learner algorithms won't have access to this information. We also found it necessary to include the angular momentum ($\dot{\theta}$) term to prevent the boat from over-steering. In practise over-steering was the hardest thing to control and we will see this in later chapters as well.

If one examines the average reward for the hand-coded controller it initially seems to have a strange shape. This is due to the simulation time; in general it takes about 200 simulation steps to achieve full speed. We did experiment with larger simulated time steps, however this introduced potential instability in our model. This instability occurred if the model could change sufficiently for the drag forces to be in the wrong direction, and cause acceleration. This increased speed would mean increased drag



(a) The course sailed by the hand-coded controller

(b) The average performance of the hand-coded controller

Figure 3.10: The hand-coded performance for the OneSail model.

which could be used for further acceleration, and the model would break.

Note the improved beating in this model when moving from buoy 1 to buoy 2. In part this is because we found our original model was unable to sail upwind. To remedy this we rotated the force generated by the sail 36^o forwards (i.e. towards the front of the boat). This had the effect of making the sail more efficient as it has less drag and more lift. With this newer model we were able to sail upwind quite efficiently.

3.5 Mathematical updates of the dynamics

The update is based on a second order Adams-Bashforth update (see page 295 in [Burden and Faires, 1997]). If trying to find a solution to the problem:

$$\dot{v}(x) = f(x)$$

An approximate solution can be given in the form:

$$v(x_t) = v(x_{t-\Delta t}) + \Delta t(1.5 f(x_t) - 0.5 f(x_{t-\Delta t}))$$

Since force is proportional to acceleration we can use this formula to calculate the velocity. We can also use this formula to calculate the updated position from the velocity.

The update equations are as follows[6]:

$$\begin{aligned} x_{i+1} &= x_i + (1.5\dot{x}_i - 0.5\dot{x}_{i-1})\Delta t \\ \dot{x}_{i+1} &= \dot{x}_i + (1.5\ddot{x}_i - 0.5\ddot{x}_{i-1})\Delta t \\ \theta_{i+1} &= \theta_i + (1.5\dot{\theta}_i - 0.5\dot{\theta}_{i-1})\Delta t \\ \dot{\theta}_{i+1} &= \dot{\theta}_i + (1.5\ddot{\theta}_i - 0.5\ddot{\theta}_{i-1})\Delta t \end{aligned}$$

with x_i, $\dot{x}_i$ and $\ddot{x}_i$ representing the position, velocity and acceleration respectively. To calculate the acceleration the force vectors are simply summed ($\ddot{x} = \frac{1}{m}\sum_j f_j$). However to calculate the torque ($\ddot{\theta}$) from several forces each having a different point of application we sum the cross-products of the vectors from the centre of mass to the point of application (r_j) and the force vector (f_j) from [Goldstein, 1950]:

$$\ddot{\theta} = \sum_j r_j \times f_j$$

explicitly for a single force:

$$\begin{aligned} \ddot{\theta}_1 &= r_2 f_3 - r_3 f_2 \\ \ddot{\theta}_2 &= r_3 f_1 - r_1 f_3 \\ \ddot{\theta}_3 &= r_1 f_2 - r_2 f_1 \end{aligned}$$

However since the forces are only acting in a plane r_3 and f_3 are zero. This means that $\ddot{\theta}_1$ and $\ddot{\theta}_2$ are zero. The value for $\ddot{\theta}_3$ can then be used in the above equation to calculate the new angular velocity and new heading.

3.5.1 The reward function

At first glance the simplest and most intuitive reward function would be to give a reward proportional to the distance covered in the target direction, i.e. if T is the target position:

$$R_i = (x_i - x_{i-1})^T \frac{(x_i - T)}{\|x_i - T\|}$$

[6]For Timin's model there is no explicit heading so the last two equations are not used.

If we are going to use the change in position with respect to a target direction then the reward signal is only dependent on the current state. However in the mathematical update formulas the position is updated using the old velocities. Thus the reward received would be the same for all actions in a given state but the next state would receive different rewards. This is a violation of the Markov property, which is an integral assumption in reinforcement learning.

To avoid this we instead chose to use the directional velocity as a reward signal i.e.

$$R_i = (x_i')^T \frac{(x_i - T)}{\|x_i - T\|}$$

where:

- x_i is the position at time i.
- T is the target position at time i.
- x_i' is the velocity at time i.

Which still gives us a useful reward signal, with the advantage of having immediate feedback.

3.6 Conclusion

In this chapter we have examined the basics of sailing and presented two conceptually different models of a sailing boat. The force calculations required for both models were also presented. Hand-coded controllers were implemented for both models which can successfully sail a simple course. We finally showed how the forces calculated can be transferred into position and velocity updates.

Chapter 4

Discrete Methods

4.1 Introduction

In the chapter the normal method of dividing up the state space is examined. We introduce the concept of a spatio-temporal eligibility trace which will be used later in this thesis. It is shown that these naïve approaches result in an extremely slow learning rate and are rather wasteful. The trace does improve the performance noticeably but it is still a poor method. A dynamic programming approach is used to find what is arguably the best performance that these methods will obtain.

4.2 The Naïve Method

Due to the additional complexity involved in a continuous reinforcement learning framework when researchers are faced with a naturally continuous problem many simply convert it to a discrete problem. However the easiest approach is seldom the best approach and while the problem is now simplified the performance can be severely hampered. This can be as a result of many things:

- *Too coarse a discretisation.* If the continuous variables are sectioned too coarsely then it is possible that many conceptually distinct states which require separate actions are now mapped to the same discrete state. This will result in poor learning as there will be a large variation in the rewards received for each action.

- *Too fine a quantisation.* However if the variables are sectioned too finely then there is a problem with the 'Curse of Dimensionality'. Moreover it is highly unlikely that each state will be experienced a sufficient number of times as the size of the state space explodes exponentially.

- *Varying complexity.* For some problems the complexity of the reward function can vary tremendously. The sailing model is a good example of this. As the boat sails closer to the wind a greater accuracy is required (if the boat sails too close to the wind, it rapidly loses speed, too far from the wind and it doesn't make any progress). However when the boat has to sail downwind, the task is a lot easier. Ideally a method should take advantage of this and allocate storage proportionally to the local complexity of the reward function.

- *The wrong offset.* Even if the discretisation has partitioned the state space into correctly sized units there is no guarantee that these states will be nicely aligned with significant changes in the reward function. For example: if a robot is learning to navigate in a cluttered environment then in some cases the robot should turn left, while in others right. If the boundary condition lies in the middle of a discrete state then that state would receive conflicting reinforcement. Again this is minimised if the states are sectioned as finely as possible, but this is a trade-off against learning time and storage space.

4.2.1 Experimental Setup

To learn the Annevolve task the state,action space was sectioned as follows:

- 12 states for the relative wind equally spaced in $[-\pi, \pi]$.

- 12 states to encode the directional error equally spaced in $[-\pi, \pi]$.

- 5 states for the speed ($[0.05, 0.5]$).

- 5 states for the rudder action ($[-\frac{\pi}{4}, \frac{\pi}{4}]$).

- 5 states for the sail action in ($[-\frac{\pi}{2}, \frac{\pi}{2}]$).

This resulted in a total of 18000 state-action pairs. We did try these methods on the OneSail model, however the number of state-action pairs was prohibitive ($5^4 \times 10^2 \times 4 = 250000$!). This is testament to the poor scaling properties of these algorithms. Standard temporal difference learning (TD(λ)) was implemented with the parameters as follows:

Parameter	Value
λ	0.9
γ	0.9
ε	$0.9 \times 0.9995^t + 0.05$
α	$0.5 \times 0.9995^t + 0.3$

where ε represented the probability that an exploratory action would be taken, and α represented the weighting that was given to new experience. These parameters have been shown to be fairly robust in terms of performance so we did not attempt to optimise them extensively. The decaying formulas used for α and ε will be used throughout this thesis, so it is worth describing their form in some detail now.

The decaying formula is of the form:

$$f(t) = a * b^t + c$$

This formula has the following properties:

$$\begin{aligned} f(0) &= a + c \\ f(\infty) &= c \end{aligned}$$

If we know in advance how many simulation steps there will be in learning a task then we can set b so that there is a graceful transition from the initial value to the final value. This allows us to have a large update rate in the beginning of the trial, but near the end of the trial the update rate can be small to increase the accuracy.

Throughout this thesis, sailing performance is evaluated in two ways, one quantitative and one qualitative. In the quantitative approach the model is randomly initialised with the boat far away from a target, facing in a random direction and with random velocity. Random exploratory moves are turned off. The performance is recorded for a

small number of steps and then the process is repeated. The rewards received are then averaged, and plotted as a function of time.

By sailing towards a target a better idea of the performance is obtained, since the target direction changes slightly if the boat is sailing off course. However this is less of a problem if the boat is able to correct for this deviation later.

The qualitative evaluation is to observe the boat's ability to sail downwind (i.e. running), reaching and beating. The behaviour observed is then compared to the hand-coded controller and any particular quirks are noted.

4.3 Spatio-temporal traces

Here we examine the use of a spatio-temporal eligibility trace [Thompson, 2002]. The standard temporal trace awards eligibility to states which have recently been experienced, while a spatial trace awards eligibility to states that are similar to the experienced state. A spatio-temporal trace simply awards eligibility to states similar to those recently experienced (see figure 4.1, which is very similar to the figure in [Thompson, 2002]). In this way we can hope to gather a broad overview of the task, early on, without having to experience every state. As the trial progresses however we will want to narrow the spatial aspect of the trace to enable a higher resolution of the task.

The trace implemented follows the suggestions made in [Thompson, 2002] with one exception. To ensure the eligibility traces never exceed one Thompson applies a sigmoidal function to the eligibilities.

$$elig(i,j) \Leftarrow 1 - e^{elig(i,j)}$$

The eligibility trace implemented in our case is:

$$elig(i,j) \Leftarrow max(neighbour(i,j), \lambda\gamma elig(i,j)) \tag{4.1}$$

We implemented the eligibility traces in this manner as we felt this was more faithful to the original theory of replacing traces (see [Sutton and Barto, 1998]). Our method has the advantage of degenerating into a standard replacing trace as the spatial eligibilty

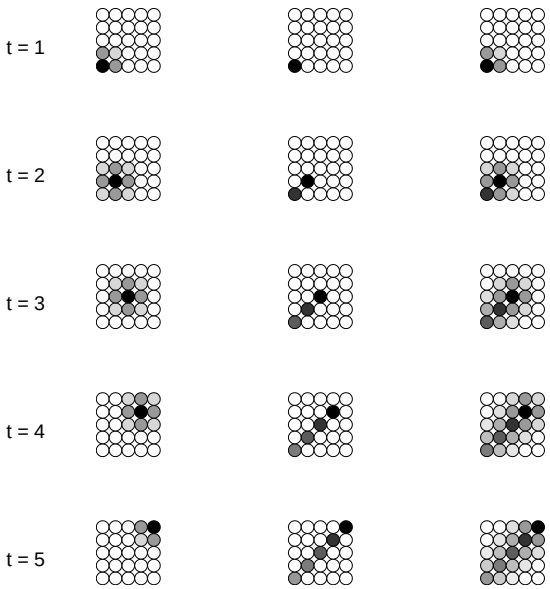


Figure 4.1: Three different types of eligibility traces, from left to right: a spatial trace, a temporal trace and a spatio-temporal trace. Eligibility is represented through shading; darker states are more eligible (after [Thompson, 2002])

shrinks. The *neighbour* function was a Gaussian curve centred on the closest discrete state.

The spread of the Gaussian was determined by specifying the value of an immediate neighbour. This was chosen as it is a fairly intuitive measure. If the immediate neighbour's value is γ then continuing in the same direction the eligibility will be $\{\gamma^{2^2}, \gamma^{3^2}, \gamma^{4^2} ...\}$. Thus one can quickly work out how large an area will be affected by the update, for example if $\gamma = 0.5$ then the eligibilities decrease as follows: {1, 0.5, 0.0625, 0.002...} so the updates will affect states only two neighbours away, after that the eligibility is negligible.

In practice one might want to specify a different variance for each dimension, which could enable a higher precision in critical states earlier on and lead to superior performance. However this was not investigated further.

4.3.1 Results

We tested the standard tabular method as well as the method involving spatio-temporal traces on Timin's model. We first evaluated the learning speed for both methods. This was done by training each method for a total of 30000 steps but evaluating the performance after every 1000[1]. (The performance evaluation was the mean reward from 50 episodes; each 50 steps long with no exploratory moves.) This was repeated 15 times and the results are plotted in figure 4.2. From this we can see that the spatio-temporal trace has increased the learning rate greatly; the performance plateaus after roughly 15000 steps. [Smith, 2001a] reports that neighbourhood learning increased the rate of convergence by a factor of six, from this diagram it seems that spatio-temporal learning has an even faster rate; the performance after 1000 steps is roughly equivalent to the final performance of the standard method after 30000. However if the experiment was run for a sufficiently long time then the standard method would eventually increase its performance to the same level.

The final performance was also examined in more detail. To quantitatively evaluate the performance of both methods exploratory moves were turned off and the simulation was run for 100 steps. This was repeated 100 times and the average reinforcement was recorded. The results are plotted in figure 4.3. From this we can see that using a spatio-temporal trace allows an average reward which is consistently greater than 0.2, this is significantly larger than the average reinforcement received by the standard method which peaks at 0.15.

For qualitative evaluation of the spatio-temporal traces the three aspects of sailing behaviour were examined:

- *Reaching.* The controller was able to move towards the target in a reasonably fast and direct manner.
- *Running.* While the controller was reasonably fast, the direction was considerably off from the target, this then required a large adjustment to its course.
- *Beating.* The controller was able to marginally sail into the wind, however the

[1]This experience was broken up into shorter episodes lasting 50 simulation steps, which ensured that most of the state space was experienced. Thus performance was evaluated after every 20 episodes.

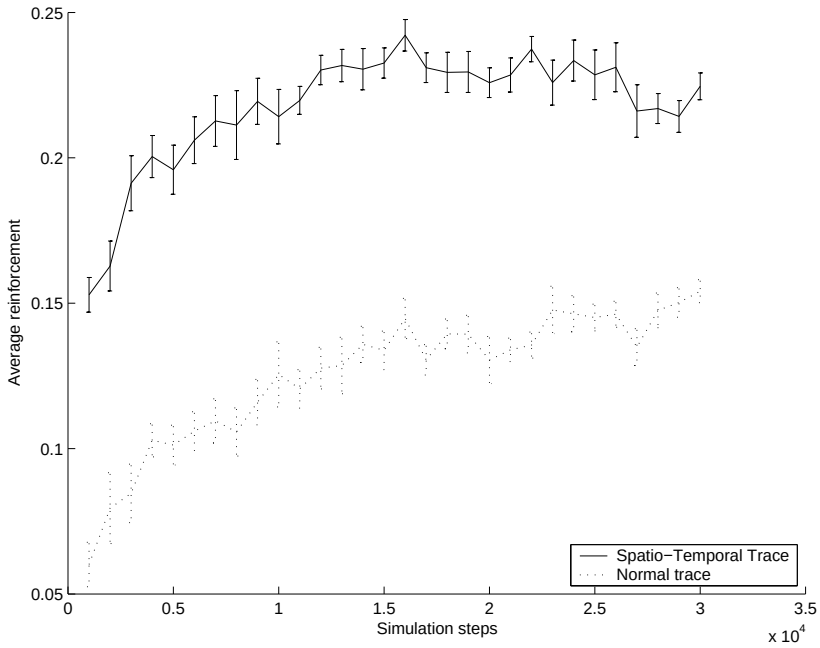


Figure 4.2: The average performance during learning. (Averaged over 15 separate trials.)

controller was unable to tack when the course changed direction. This would result in the boat drifting slowly away from the target

4.4 The dynamic programming approach

We felt that the poor performance, of the above methods might be due to the large number of states, which would require a long simulation time to propagate the information to all states. In light of this another plain discretisation method was investigated. Since our learning task is a simulation it is easy to sample each state-action pair exactly once. A rough model is then built up from sampling each state-action pair and a full dynamic programming update can be performed repeatedly (for more information on Dynamic Programming see [Bertsekas, 1995]).

This ensures that the knowledge from the rewards and the state transitions is propagated in its entirety (until convergence occurs). Arguably this is the upper limit of performance that a plain discrete method could achieve, with some caveats:

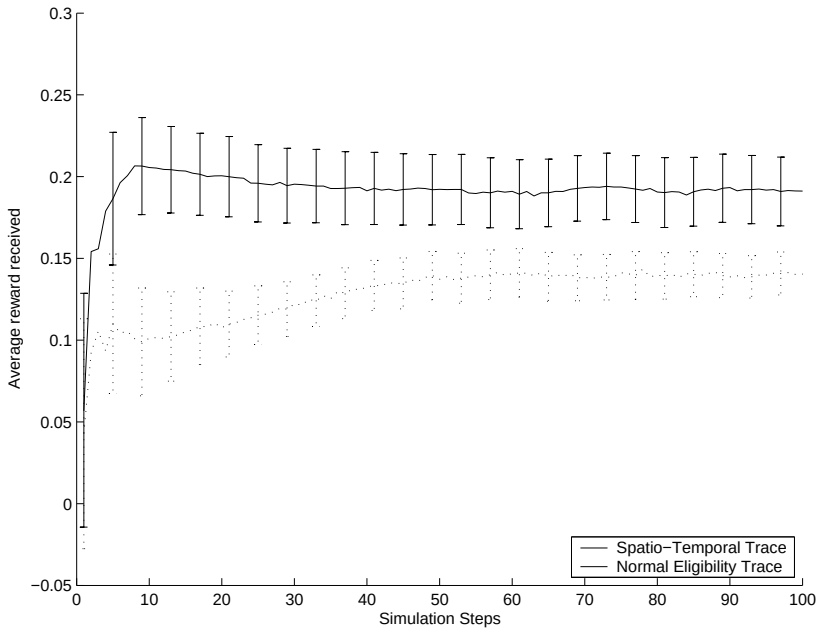


Figure 4.3: A comparison of the typical performance of TD(λ) learning with and without spatio-temporal eligibility traces in Timin's model after learning for 30000 simulation steps

- While sampling every state once, the model is constructed by using the following approximation of the transition probabilities:

$$Pr(s'|s,a) = \begin{cases} 1 & \text{if } s' \text{ is the next state experienced} \\ 0 & \text{otherwise} \end{cases}$$

- This optimisation is carried out without regard to the input space distribution. In the normal course of sailing only a small number of states will be experienced. Performance might be improved if the transition probabilities were better approximated for these states.

- A certain amount of toying with the number of discrete values a state can take is also required. If the number of states is too small then most of the transition probabilities will loop back to the original state as the state won't have changed sufficiently. Of course the algorithm is exponential in the number of states so some care is required.

Once the model has been completely built up then the dynamic programming update:

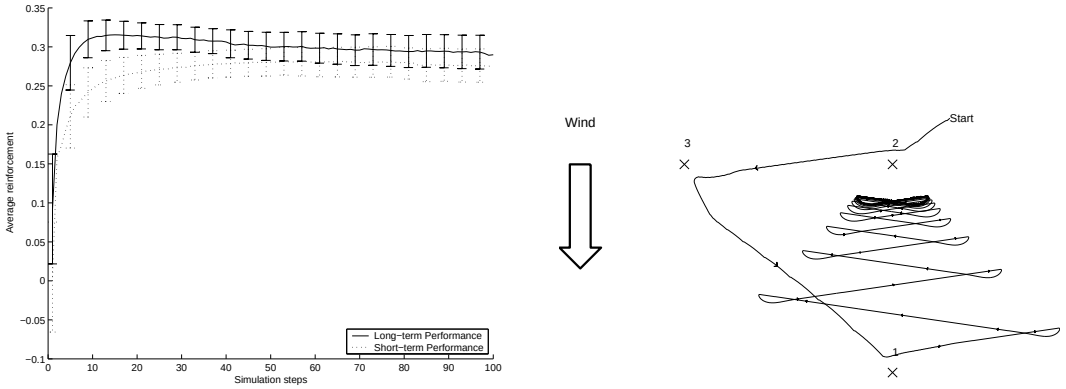
$$Q(s,a) \leftarrow \sum_{s'} Pr(s'|s,a) \left[R^a_{s,s'} + \gamma \max_{a'} Q(s',a') \right]$$

is repeatedly applied until convergence occurs.

It must be pointed out however that in most cases an explicit computer model will not be available (although there are many methods for learning the model for example the Dyna architecture in [Sutton and Barto, 1998]). Moreover even if a model is available but there is noise in the simulation then the state-transition approximation will need to be performed many times before any accuracy is achieved. Thus dynamic programming in general is *not* a useful method but is presented here to show the maximum limits achievable with a static quantisation.

4.4.1 Results

For the updates a value of $\gamma = 0.9$ was used and we performed 100 iterations of the algorithm. The same quantisation as the preceding controllers was used. The performance was evaluated at two stages, the first was just after the model had been



(a) The average reward received.

(b) A trace of the route followed.

Figure 4.4: The performance of the dynamic programming approach

constructed, the second was after the updates had been performed. The quantitative analysis was also evaluated in the same manner as above, and the results are shown in figure 4.4(a). From the figure we can see that the performance after the initial stage is better than the spatio-temporal method. Applying the updates improves the performance further. The qualitative performance is as follows:

- *Reaching.* The controller is able to reach at a fast speed, and it heads directly towards the goal.
- *Running.* The controller moves downwind with good speed and accuracy.
- *Beating.* The controller is able to sail upwind, however it lacks the required accuracy to do it efficiently. As a result it sails far away from the wind and does not make much ground. Another interesting thing to note is that the boat tacks by turning away from the wind, rather than towards it. We are not sure why this is but it could be due to the coarse quantisation which might miss such a manoeuvre. This results in the controller losing ground. As the boat approaches the second buoy the required tacks increase in frequency until the ground lost by tacking is the same as the amount gained by sailing into the wind (this can be seen in fig 4.4(b)).

The boat sails upwind very slowly after the dynamic programming has been applied as a result of the coarseness of the model that has been constructed. When we examined the model we found that roughly 44% of the state-action pairs indicated that they transitioned to themselves. Also in 19% of the states, the state transition was the same, no matter which action was tried. This shows that the quantisation is too coarse to usefully represent all the states possible in the model. However we may improve things by only focussing on the regions which are likely to occur during normal sailing. This leads to adaptive quantisation.

4.5 Conclusion

In this chapter we examined several naïve quantisation methods as applied to Timin's model. We did not experiment using these approaches on the OneSail model as they scale very badly. The results obtained were rather poor, with none of the methods being able to sail upwind. The best approach was found to use dynamic programming and construct an approximate model beforehand, however this approach is seldom possible in real-world problems.

Chapter 5

Smith's Method

5.1 Introduction

In the previous chapter static quantisation of the state-action space was explored. In this chapter we present a method of dynamic quantisation taken from [Smith, 2001b]. We use the spatio-eligibility trace presented in the previous chapter and apply it to Smith's method.

5.2 Smith's model

In his PhD thesis Smith explores a few simple models, such as controlling a Khepara robot and shows that this method is able to learn an adequate controller for such a task. The sailing task is considerably more complex, and represents a good test of the scaling properties of the algorithm.

In [Smith, 2001b] the tasks attempted needed only a one-step return. However since this task is more complex the spatio-temporal trace from chapter 4 is used. In this way neighbouring state-action pairs can share the reward.

The algorithm uses two Kohonen Self Organising Feature Maps (SOFM's). The first SOFM is used to map actions and the second is inputs. A Q-table is maintained as a link between states and actions. The algorithm is as follows (taken from [Smith, 2001a]):

1. Initialise the *input map* and *action map* to small random values.

2. Present the *input map* with the current state vector and identify the winning unit in the *input map* (s_j).

3. Identify a unit in the *action map* as follows:

$$action = \begin{cases} argmax_a(Q(s_j,a)) & \text{with } \Pr(1-\varepsilon) \\ \text{Random Action} & \text{with } \Pr(\varepsilon) \end{cases}$$

4. The action chosen is calculated by adding some exploratory noise to the weights of the winning node of the *action map*.

5. Receive reinforcement r and next state s' from the environment.

6. If $r + \gamma \max_i Q(s'_j, a_i) > Q(s_j, a_j)$, then the perturbed action appears to be an improvement over the proposed action, so update the *action map* towards the perturbed action.

7. Update *all* Q-values towards the corrected return proportionallly to the Q-learning rate and the product of the two neighbourhoods (of the two SOFM's) :

$$elig(s,a) \leftarrow \max(N_{state}(s)N_{action}(a), \gamma\lambda elig(s,a)) \tag{5.1}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha(t) elig(s,a) \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

8. Update the *input map* towards the state just seen according to the usual SOFM update rule.

9. Goto instruction 2

Comparing the update equation 5.1 with equation 4.1 on page 36 then one notices how the product of the neighbourhood functions of the state SOFM and the action SOFM, forms the spatio-temporal trace. In [Smith, 2001a] this is called neighbourhood Q-learning, however the temporal aspect of the trace is not included in that article.

5.2.1 Adaptive Discretisation

Smith's method is attractive for several reasons:

- *Adaptivity* The most important reason is that the discretisation is adaptive in both the state and the action spaces. This is important in learning to control a sailing model as it is not clear which states will be experienced, in using the model. It also avoids many of the difficulties discussed in the beginning of this section.
- *Simplicity* The second reason for choosing such a method is for its conceptual simplicity. It is easy to understand and implement.
- *Distribution sensitive* The SOFM honours the input distribution, this allows a higher resolution in state-space regions which are experienced more often. While there is no guarantee that this is where the resolution is needed it does seem reasonable to apportion resources this way.

5.3 Representation of the state

An advantage of using self organising maps is their dimensionality reduction. They automatically map onto any manifold present in the state space, which allows us to use a better representation of the state space. Rather than using the angles directly we are now able to take the sine and cosine of the angles. Using this representation angles slightly less than π appear close to angles slightly greater than $-\pi$, if one uses the plain angle then these angles appear far apart. While the size of the the state representation has increased, there is no change in the intrinsic dimensionality. Respecting any manifold present in the state space could speed up learning significantly. We decided to investigate whether this would be the case here.

To test this we ran Smith's algorithm with the following parameters:

Parameter	AnnEvolve	OneSail
Input map size	12×12	$6\times6\times6$
Action map size	5×5	5×5
Iterations	30000	50000
Input map neighbourhood[1]	$0.5f(t)+0.25$	$0.5f(t)+0.25$
Action map neighbourhood	$0.5f(t)+0.25$	$0.5f(t)+0.25$
$\alpha(t)$	$0.5f(t)+0.3$	$0.5f(t)+0.3$
Learning rate of input map	$0.5f(t)+0.25$	$0.5f(t)+0.25$
Learning rate of action map	$0.5f(t)+0.25$	$0.5f(t)+0.25$
Probability of exploration	$0.9f(t)+0.01$	$0.9f(t)+0.01$
Standard deviation of noise	$0.5f(t)+0.05$	$0.5f(t)+0.05$
Annealing schedule ($f(t)$)	0.9997^t	0.9999^t

In order to ensure the SOFM's covered the entire state space they were trained in short episodes (50 steps long) and we randomly reinitialised the model afterwards. This ensured that large parts of the SOFM did not converge to cover a single episode, which would occur if many similar states were shown to the SOFM. The experiment was repeated 15 times and the averaged results are shown in figure 5.1.

After the training the performance was evaluated by disabling all random effects (the exploration as well as the fine-tuning noise). What is impressive is that the method has learnt reasonable control of Timin's model with one fifth of the number of states as in the discrete case. However the performance is still not as good as the dynamic programming approach. We did experiment with using larger state space maps, however the performance did not improve significantly and the training times were noticeably longer.

From figure 5.1 we can see that there is no increase in the performance when using the sine and cosine of the angles rather than the angles themselves. This is due to the fact that the best place to introduce a discontinuity would be between π and $-\pi$ since this is the angle at which the best direction to turn changes. By breaking the angle in the best place there is nothing to be gained from using a sine and cosine representation, but it is reassuring that there is nothing to be lost either. For the rest of this chapter we

[1]The neighbourhood functions for both state and action SOFM's are Gaussians, specifi ed in the same manner as in section 4.3

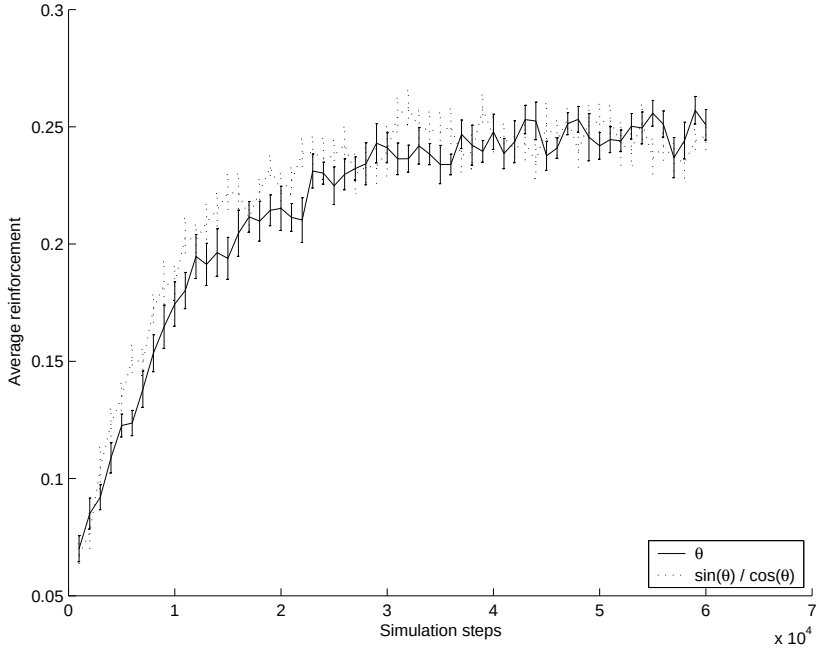


Figure 5.1: A comparison of different state representations on the learning performance (Timin's model)

use the sine and cosine representation, although the results are similar if plain angles are used.

Qualitatively the performance is identical for both representations. The boat has learnt to sail downwind very well, however it battles to sail even slightly above the wind. The controller's actions are somewhat jerky and this definitely reduces its effectiveness. While there are ways to interpolate outputs in a SOFM, [Aupetit et al., 2000], it is not clear how to apply reinforcement learning to such interpolated results. It is quite easy to think of scenarios where such interpolation could lead to disastrous results. For example if one is trying to learn obstacle avoidance in a mobile robot then averaging a left turn and a right turn could result in a collision. [Smith, 2001b] does mention interpolation briefly in section 6.6. It is not investigated further in this thesis, but left as a possible future research direction.

5.4 Learning from a teacher

One of the problems observed while learning is that the exploration tended to disrupt the boat's speed and heading. While this was not a great problem when reaching and running, it became a problem during beating as a consistent heading is required to build up speed. As a result the boat's movements were particularly jerky as the boat moved at a low speed. Smith's algorithm learnt that large actions would ensure the boat stayed roughly on course. The boat seemed to alternate between heading away from the wind to ensure that it maintained sufficient speed, and then heading closer to the wind to obtain a larger reward. As the boat slowed however, the algorithm knew to head away or else the random noise might turn the boat around completely and it would receive large penalties while trying to turn the boat towards the target again.

Our hypothesis was that the exploration had changed the input distribution sufficiently to prevent the system learning a good solution. To investigate this further we generated experience from the hand-coded controller from which to learn. Since the controller's actions are near linear functions of the state we found that successive actions would be very similar to each other. This meant that the winning action node would very likely be the previous winning node. As the actions changed they could slowly drag the winning unit very far from its original position, which meant that the action space was not being effectively quantised. To overcome this we generated the entire 30000 simulation steps beforehand and then showed the experience in a random order. This improved the performance and the final results are shown in figure 5.2.

A comparison of the resulting action SOFM's is interesting and is shown in figure 5.3. In figure 5.3(b) we can see that a large portion of the example experience had the rudder action close to zero (which helps achieve fast speeds). As a result the action SOFM has many units covering this line. In comparison the learnt action SOFM (figure 5.3(a)) has its units more evenly spread throughout the action space. This helps explain the different shapes of the average reinforcement received in figure 5.2. In the taught case the boat is steered more with the sail, so it is slower to turn around, but once travelling in the correct direction it is able to travel faster as it uses less rudder. The learnt controller uses more rudder and is able to head in the correct direction sooner, but it corrects any deviation from the course using the rudder which limits its speed.

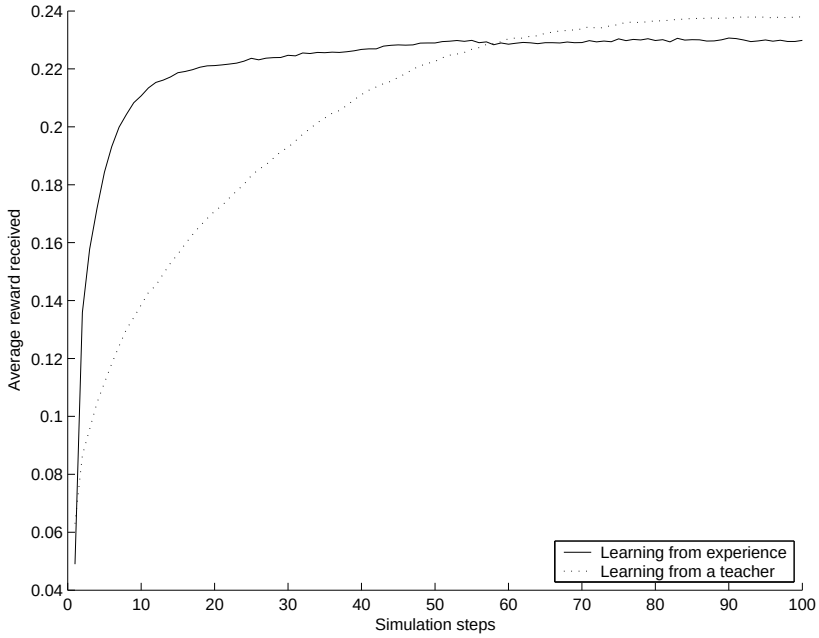
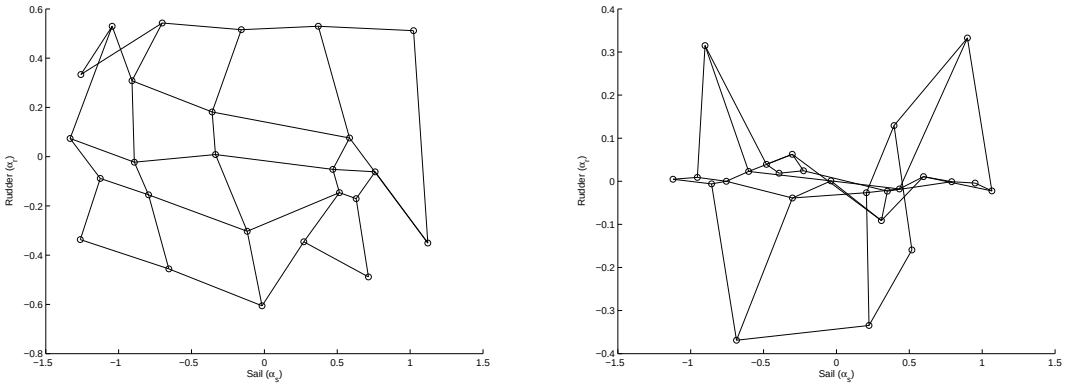


Figure 5.2: A comparison of the performance between learnt behaviour and taught behaviour. (Averaged over 15 trials)



(a) The learnt action SOFM.

(b) The action SOFM after being taught.

Figure 5.3: Differing action SOFM's

However neither method is able to able to sail very effectively into the wind. This seems to be a limitation inherent in using discrete actions and state-space representations for a naturally continuous problem.

5.5 Changing the reward

The solution which was obtained by Smith's method for the OneSail task also had extremely poor performance. After examining the behaviour this could be attributed to two things:

- *An inability to tack.* Tacking in the OneSail model required a large amount of angular momentum; this could only be achieved through the use of the rudder. Unfortunately the boat has to be moving fast for the rudder to become an effective control. This meant the boat would have to sail away for many simulation steps to build up speed before having enough speed to make the rudder effective enough to tack, however sailing away to build up speed resulted in a penalty that was too large, instead the controllers would try to turn into the wind slowly, lose speed and then drift slowly away from the wind.

- *Poor heading control.* In most cases the boat would be able to steer towards the correct direction, however it would build up too much angular momentum in doing so and over-steer. However since the boat would be moving in the correct direction it would still receive positive reward. Only several steps later would the effect of the keel have changed the velocity of the boat sufficiently to receive penalties.

We felt the performance might be improved in this case by changing the reward function, so that the reward is more explicitly a function of the heading. In figure 5.4 we show the three different rewards we tried. To calculate the reward we multiply the speed of the boat ($\|v\|$) by the reward coefficient calculated as a function of the difference in angles between the heading of the boat and the target direction of the boat (i.e. γ) from figure 3.7).

- *Cosine(γ)* This reward is very similar to the original reward, however it rewards motion while facing the target direction rather than motion towards the target direction. It is calculated:

$$\mathrm{R}_{cos}(\gamma) = \cos(\gamma)$$

- *Linear* This function is calculated as :

$$\mathrm{R}_{lin}(\gamma) = (1 - 2\frac{abs(\gamma)}{\pi})$$

- *Squared* This function is calculated in terms of the linear reward as follows:

$$\mathrm{R}_{sq}(\gamma) = sign(\mathrm{R}_{lin})\,[\mathrm{R}_{lin}]^2$$

We chose these functions to emphasize maintaining the correct heading while moving. However we initially found it necessary to modify the reward functions slightly as in some cases the boat would sail backwards briefly, while facing the target. To avoid this we monitor the boat's velocity and if it is moving backwards then the reward is always negative.

After that modification the controllers found another loophole in that they would receive no penalty if they drifted backwards perpendicularly to the target for the reward functions as the reward coefficient was zero. In some cases this was preferable to trying to build up speed (away from the target) and tacking. So the reward was modified to give a penalty equal to the negative of their speed, excluding the reward coefficient when moving backwards.

Another difficulty was that the controllers were unable to tack. To try and overcome this the reward was modified in a different manner. At low speeds the rudder is not very good at steering the boat, and one should concentrate on building up speed rather than any directional control. However once the boat is moving sufficiently fast then one can concentrate on the direction. We tried two new functions which modified a previous reward function in the following manner:

$$k = exp(-10\|v_{new}\|^2)$$

$$\mathrm{R}'(\gamma) = k(\|v_{new}\| - \|v_{old}\|) + (1-k)\mathrm{R}$$

This formula is thus a weighted average of the increase in velocity and one of the previous rewards. At low speeds the reward function would give reward for increasing the velocity even if this direction is away from the target direction. As the speed increased however reward for travelling in the target direction would be given more weight. In this way we hoped the boat would be able to sail away from the target to build up sufficient speed and then the rudder would be effective enough to tack. We tried modifying the original reward and the reward R_{lin} in this manner. The results are also shown in figure 5.5.

To test the effects of the different rewards we trained 10 controllers on each reward function. The performance was evaluated every 1000 steps, by calculating the average reward received for the controller's performance in 50 episodes lasting 50 simulation steps. The results are plotted in figure 5.5. Unfortunately all the reward functions resulted in poor performance, with no clear best reward function.

When examining the behaviour exhibited by the controllers we could see that the different reward functions did have an effect, in particular the squared function gave rise to controllers which tried very hard to keep their heading exactly the same as the target heading. However to achieve this accuracy they used a lot more rudder (in most cases they would alternate a hard left rudder action with a hard right rudder action). Since a rudder acts as a brake every time it is used this meant that these controllers were very slow.

Unfortunately none of the different rewards show an increase in performance. Our interpretation of these results is that the OneSail task requires smoothly-varying control and not just the step-like actions which are produced by a discretisation method. This suggests that a solution of the OneSail model should be naturally continuous. One such method is explored in the next chapter.

5.6 Conclusion

In this chapter we explored an adaptive discretisation method. While it managed to achieve reasonable results with much fewer states than the static quantisation methods the performance was not better. Different representations of the state space were ex-

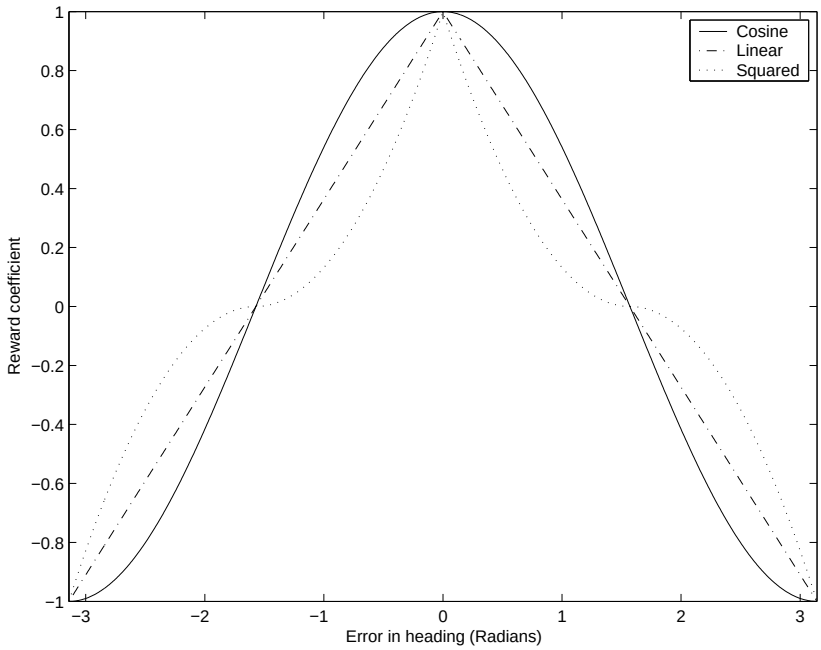


Figure 5.4: Different reward functions

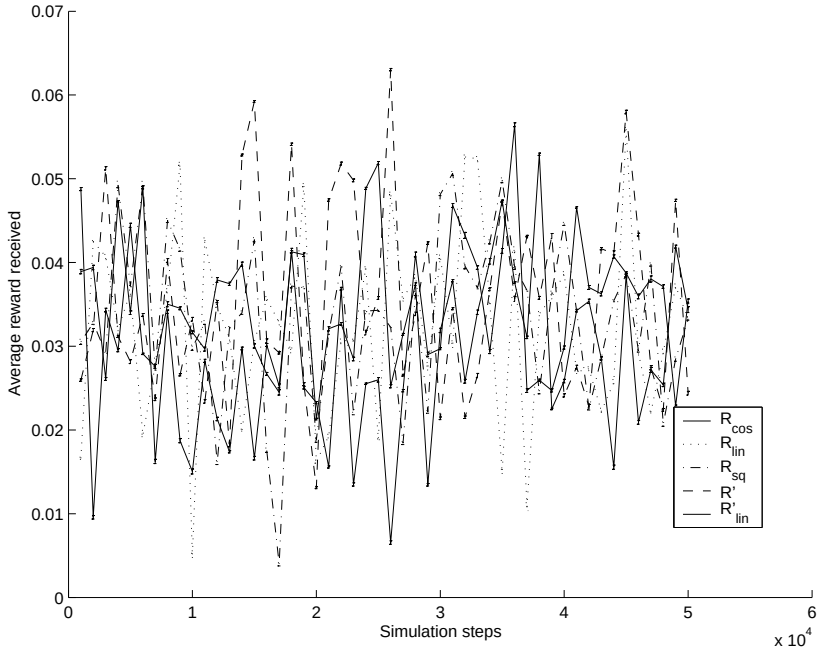


Figure 5.5: Learning performance for the different reward function for the onesail task

amined and it was found that Smith's method was robust to different representations. Unfortunately the performance for the OneSail model was rather poor, different reward functions were examined and were unable to improve the performance of this method.

Chapter 6

Continuous Methods

6.1 Introduction

In this chapter we present the final method examined for sailing control. Wire-fitting [Baird and Klopf, 1993], uses a function approximation scheme and an interpolation function to approximate the value function. We present the architecture of wire-fitting and the complementary method of advantage learning.

6.2 Wire-fitting

Wire-fitting consists of several control actions(u_i's) and control values(y_i's). A control action is a potential action for a given state. Each control value is an estimate of the reward received for taking the corresponding control action. These control values and actions are the output from a suitable function approximator such as a neural network. The neural network maps the given state into estimated control values and control actions. These control actions and the control values are fed into the interpolation function:

$$f(u) = \lim_{\varepsilon \to 0} \frac{\sum_i y_i \times [\|u - u_i\| + c(\max_k y_k - y_i) + \varepsilon]^{-1}}{\sum_i [\|u - u_i\| + c(\max_k y_k - y_i) + \varepsilon]^{-1}} \tag{6.1}$$

by using this interpolation function we are able to estimate the reward received from actions other than the control actions. Equation 6.1 has some useful properties for reinforcement learning:

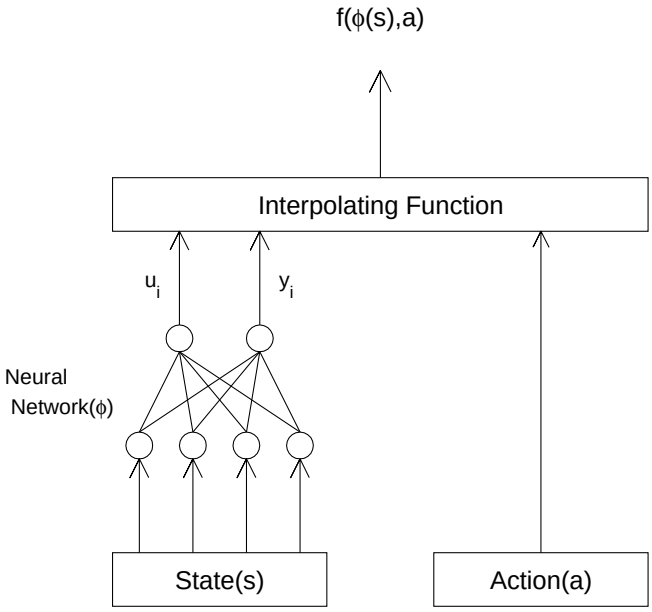


Figure 6.1: The wire-fitting architecture (after [Baird and Klopf, 1993]).

- *No wild predictions.* If the point encountered is far away from known points then the expected value is simply the average of the control values. This prevents any unrealistic extrapolation.

- *Analytical Maximum.* Since any point away from the control actions is a weighted average of the control values, the maximum of the function must occur at one of the actions. Moreover the form of the interpolation guarantees that that maximum will pass through the maximum control value. This means that finding the maximum of the function does not require any evaluations of the function, one can simply use the maximum control value.

- *Differentiability.* This function is differentiable, which allows the gradient of the error to be calculated and passed back to the function approximation scheme. This gradient can then be used to calculate new targets to train the function approximation scheme on. Unfortunately the gradient of equation 6.1 has a very complicated form, it is not technically difficult to derive however and so we do not present it here. The interested reader is advised to read [Gaskett et al., 1999a] for more details.

Since the control values and actions are the output of a function approximator they are smoothly changing functions of the state. Since the policy is simply the action with the maximum control value we can see that the policy action is also a smoothly changing function of the state. However taking the maximum introduces discontinuities into the policy action at decision boundaries where the action with the maximum value changes. This is a good class of policies, as the crisp decision boundaries avoid problems with averaging intermediate values, for example in a collision avoidance task either a left turn or a right turn might be acceptable but averaging them results in moving forwards which could lead to a collision.

A potential disadvantage is that the formula relies on the norm. For high dimensional spaces the norm behaves non-intuitively ([Bishop, 1995] page 29[1]). This would have the result that the interpolated function would be near the mean for most input values, rendering the interpolated function useless for very high-dimensional surfaces.

Added flexibility comes from the fact that architecture does not place any restrictions on the type of function approximation scheme. For this task we used a back-propagation neural network [Bishop, 1995], but other types are possible such as lazy learning or radial basis functions.

An outline of the algorithm we followed:

1. Initialise the neural network weights to small random values.

2. Calculate control actions and control values by presenting the current state to the neural network.

3. Get the action

$$action = \begin{cases} u_i \text{ where } i = argmax_j(y_j) & \text{with } \Pr(1-\varepsilon) \\ \text{Random Action} & \text{with } \Pr(\varepsilon) \end{cases}$$

4. Add some exploratory noise to the action.

5. Take action and observe reward r and next state s'.

[1] Bishop gives an exercise showing that the proportion of volume that a d-dimensional sphere contained in a d-dimensional cube tends to zero as d increases. As a control action only has a spherical region of influence (from the norm) this shows that it too suffers from problems of dimensionality although to a far lesser extent than most other methods.

6. Calculate the reward predicted by equation 6.1.

7. Calculate the gradient of the error and use this to update the network weights.

8. Train the network on the updated control actions and values.

9. Goto step 2.

We also found it necessary to bound the targets for the control actions and action or else they would sometimes diverge. This is in part due to the gradient calculation updating all the control actions when it really only needs to update the nearest few. As an example if the sampled action has a lower reward than predicted it will push all the control actions which are above it (i.e. have a greater control value) away and lower them slightly. If they are already at the bounds of the action space it makes sense to restrict them, since they will not contribute to the surface otherwise.

6.3 Advantage learning

Since wire-fitting is designed for continuous state and action methods, it is natural to include advantage updating which can speed the convergence of near-continuous time learning by orders of magnitude [Baird, 1993]. If a problem requires continuous states and actions it is normally because fine-grained control is required. Advantage updating emphasizes the differences between states by a factor proportional to Δt (see page 10 for more details).

In the original paper on advantage updating, it was proposed that one learn both a value function and an advantage function which is then normalised so that the maximum value for the advantage is zero in all states. However it is possible to modify the formula so that one does not need to model both the value and advantage functions [Baird, 1995]:

$$A(x,u) \leftarrow (1-\alpha)A(x,u)+$$
$$\alpha\left[\frac{1}{\Delta t}(R+\gamma^{\Delta t}\max_{u_{t+1}}A(x_{t+1},u_{t+1}))+(1-\frac{1}{\Delta t})\max_{u_t}A(x_t,u_t)\right] \tag{6.2}$$

This is the advantage learning update (as opposed to advantage updating).

6.4 Experimental Setup

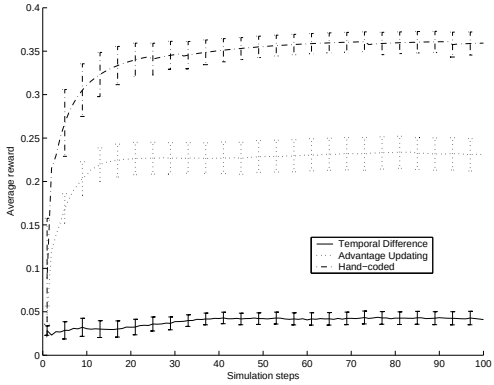
Since we are using a neural network for learning, we must be careful of forgetting previous knowledge. This interference can occur if we only focus on updating the network for the most recent experience. If we perform an update which is greedy with respect to the new experience we can change the network in other parts of the state-space for the worse. Interference is also a problem for Kohonen's Self-Organising Feature maps, which led us to use short training episodes in chapter 5. There are many ways to avoid interference, we take the simplest approach of performing training updates in batches. Other methods do exist, one method followed in [Gaskett et al., 1999a] is to maintain a buffer of experience which is constantly updated and retrained. We found generating new experience was very fast, and did not keep any data, once it had been trained on.

We found that increasing the size of the batches lead to increased performance. This is probably as a result of minimising the possible interference. However we found that initially only small batches were necessary to increase the performance. This led us to starting with a small batch size and incrementally increasing it after every batch update. We also found that the advantage learning update had slow convergence which meant that we needed more than double the previous number of iterations to achieve maximum performance.

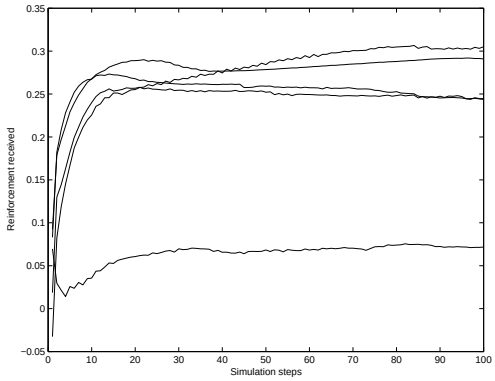
The parameters for both tasks are given:

Parameter	AnnEvolve	OneSail
Initial Batch size	400	400
Increment	100	250
Number of updates	30	30
Number of control wires	25	35
Number of hidden units	12	20

The neural network had a single hidden layer of `tansig` units, with a linear output layer. We wanted to confirm the results of [Gaskett et al., 1999a] which showed a noticeable difference in performance between the standard temporal difference update and the update proposed by advantage learning. To test this we ran 15 trials of each update. The results are shown in figure 6.2(a). It is interesting to note the differences in the surfaces predicted by both methods. We plot two typical surfaces in figure 6.3,



(a) Average performance of temporal difference learning and advantage learning.



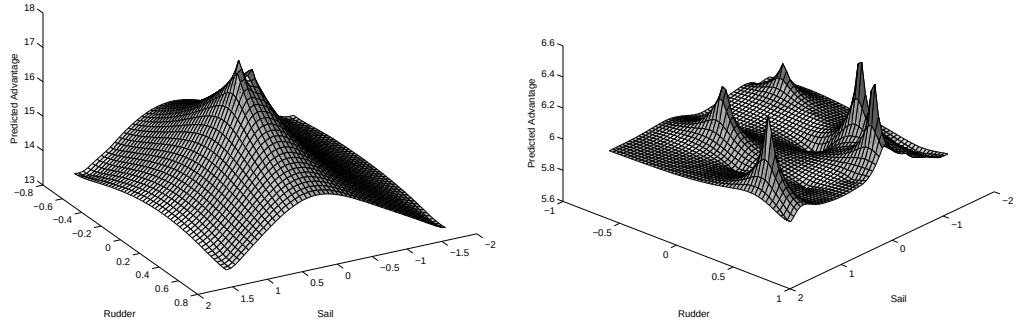
(b) Individual performance of 5 trials (using advantage learning).

Figure 6.2: Wire-fitting for Timin's task.

note the increased scale in the z-axis for the advantage updating method, this increased scale gives the neural network an easier surface to fit, which accounts for the improved performance.

Advantage learning did particularly well on learning Timin's model. In fact it found several solutions which were better than the hand-coded controller (see figure 3.6(a)). However since the algorithm is only guaranteed to converge on a local minimum not all trials had equally good results. In figure 6.2(b) we plotted the performance of five individual trials. From this we can see that four of the trials converged to solutions which had a similar performance to the dynamic programming method, however these solutions were able to sail into the wind much more efficiently. Unfortunately most of the time the controllers were able to sail very efficiently into the wind on one side, but not efficiently on the other. Taking advantage of the symmetry in this task would improve their performance further.

The OneSail task also had its best performance here, although its final performance was not as good as its hand-coded controller. This performance was for the the linear reward ($R_l in$) mentioned in the previous chapter, although the other reward functions from the previous chapter were also examined. The frequency of good solutions found was found was much less (only roughly 1 in 5 would have the performance given in



(a) An advantage updating surface (b) A standard temporal difference surface

Figure 6.3: A comparison of the wire-fitting surfaces (note the different scales in the Z-axis)

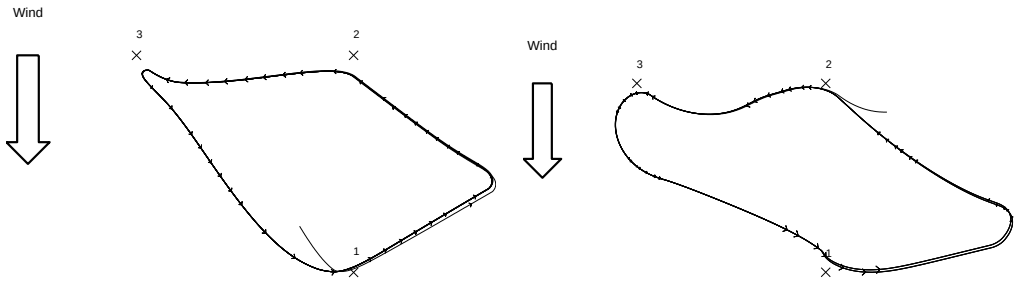


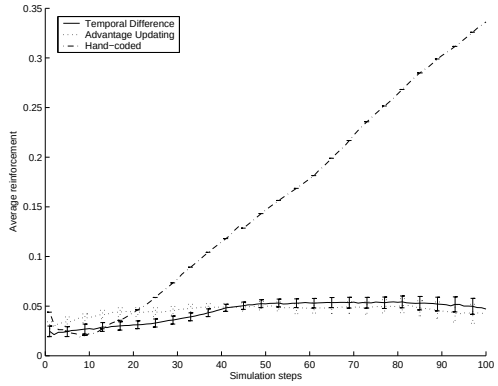
Figure 6.4: Two examples of solutions for Timin's model found using wire-fitting. (Compare these with figure 3.6(a))

figure 6.5(b)). Moreover none of the solutions had the accuracy to successfully sail the entire course.

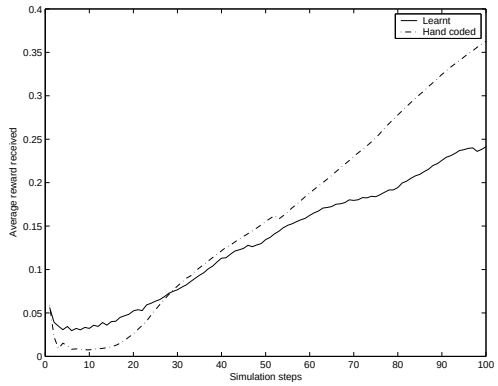
An interesting side-effect occurred when we examined the squared reward function (which has a reward gradient equal to zero when the boat is perpendicular to the target). As a result the fine-tuning noise did not help in determining which direction to change the control actions. If the boat was initially facing away from the target it would steer towards the target, but stop as it approached perpendicularity. In some cases it would sail perpendicularly for some while before accidentally heading close enough to the target to aim directly. This lead us to changing the reward function slightly as detailed in the previous chapter.

6.5 Conclusion

In this chapter we presented a fully continuous reinforcement learning method which has the best performance out of all the previous methods. The convergence of the method is comparatively slow, and unfortunately there is no guarantee of a good solution after the trial has been completed.



(a) The average performance of advantage learning and temporal difference learning



(b) A good solution found.

Figure 6.5: Wire-fitting for the OneSail task.

Chapter 7

Discussion

7.1 Introduction

In this chapter we provide an overview of the the thesis. Possible future research directions are also presented, along with a discussion of the applicability of this work.

7.2 Overview

The thesis explored the application of continuous reinforcement learning to an interesting control problem: sailing. We presented two different models of a sailing boat: Timin's model and the OneSail model. By exploring the conventional method of quantising the state-space we showed that the performance achieved could never be very good. This was in spite of using a spatio-temporal trace which sped the learning rate up considerably.

The dynamic programming approach constructed an approximate model of the task and then repeatedly applied the dynamic programming update. This proved to be the best discrete method to use, however in practise one will not be able to sample each possible state and action pair to construct the approximate model.

This failure of the standard approach lead us to consider an adaptive quantisation approach. Smith's method used two Kohonen Self-Organising Maps to cluster the state space and action space respectively. This allowed us to use one-fifth of the resources

required by the traditional methods and still achieve comparable results in Timin's model. However the performance for the OneSail model was extremely poor. This seemed to be inherent to the discrete approach as the OneSail task seemed to require smoothly varying outputs for a slowly changing state.

In the preceding chapter we then explored a wholly continuous reinforcement learning method: wire-fitting. This method uses a combination of an interpolating function and several example actions to estimate the reward from following an action. These example actions are themselves output from a neural network. Since the interpolating function is differentiable we are able to perform a gradient descent method to adjust the weights of the neural network.

Wire-fitting managed to achieve the best performance for both models. In particular some solutions found for Timin's model were better than the hand-coded solution. However the method only converges to a local minimum and not all solutions were equally good. For the OneSail task reasonable solutions were found only an estimated 30

7.3 Discussion

This thesis originally had the intent of examining the claim made in [Adriaans, 2003] that reinforcement learning when applied to sailing had an incredibly slow convergence rate. While our slowest method (wire-fitting) experienced 240 000 simulations steps before only a mediocre performance was obtained for the OneSail model it should be pointed out that in [van Aartrijk and Samoocha, 2003] data-mining is performed on a database containing twelve million entries. Given this amount of experience we feel it would be possible to achieve good performance using reinforcement learning for several of the behaviours in Adriaans' agent based decomposition.

In both Smith's method and wire-fitting there was a small amount of fine-tuning noise which was added to each action. In a real system one would not be able to simply add random noise, but one could systematically fine-tune the action with a coherent exploration strategy. This would enable the autopilot to slowly adapt to each boat, an advantage which is not present in the current architecture of Adriaans.

This forms the suggested direction of future research; to apply reinforcement learning to other behaviours in the behaviour-based decomposition of Adriaans in order to determine whether or not such a compositional approach to reinforcement learning is viable. However several other research directions arose which also appear promising. These are listed below.

7.3.1 Beating

Beating was the hardest subtask to learn in both models. Sailing into the wind required fine control which was not present in Smith's method. We were disappointed that so few trials found the proper way to tack in the OneSail model; a 90^o turn through the wind rather than a 270^o turn away from the wind. It could be that the random exploration simply did not occur frequently enough at the right time to find it. However we could have been a bit too optimistic considering that the turn required a fair amount of forward speed and a hard rudder command at the correct time; considering the short episode length there would be only a few chances to find this action.

Given extra time it would be worthwhile seeing if better beating could be learnt simply through biasing the training to include more of it.

7.3.2 Interpolation

In Timin's model we feel Smith's method performed extremely well considering its limited resources. We feel it might have performed even better had there been an appropriate form of interpolation. One such heuristic could consider not just the winning node in the state SOFM, but also its nearest neighbour. If the actions recommended by both nodes are reasonably similar then the average of the actions could be chosen. If the actions are quite distinct then only one action should be chosen.

This would avoid trying to average a left turn and a right turn and end up moving forward into an obstacle etc. However this approach seems to introduce even more parameters into Smith's method which already has too many. In what way might one define similar a similar action - the most natural way would be to determine the distance from each action node in terms of the number of nodes separating them. However

this is left as a possible future research direction.

7.3.3 Shaping

We found it necessary to rotate the sail force forwards in the OneSail model. This increased the efficiency of the sail and made the sailing task easier. It would be interesting to see if wire-fitting and Smith's method would have performed better if the task was shaped. In the beginning of the trial the force of the sail could be rotated forwards by a large amount. This would make the task easier to learn, and they could learn an initial policy rapidly. However as the trial progresses the sail force is gradually brought back to its original form. Since the learning methods are adaptive this slow adaption to the changing task should be easy.

7.3.4 Symmetry

Since the task is symmetrical (i.e a left turn is equivalent to a right turn when the boat is reflected about the direction of the true wind) learning could be sped up if one takes advantage of this symmetry. Unfortunately it is unlikely that the performance will improve as we ran all the learning methods until they showed no increase in performance.

7.4 Implications of this work

While wire-fitting's performance was not consistent we still feel that continuous reinforcement learning is a promising research area. In particular the ability to generalise in a reinforcement learning system will be invaluable in increasing the learning rate for real world problems.

However while a neural network representation of the expected reward is conceptually pleasing, in practise the results achieved are poor. It may turn out that the generalisation ability for reinforcement learning will have to come from a local approximation scheme such as receptive field weighted regression which is relatively free from problems of interference.

Bibliography

[Adriaans, 2003] Adriaans, P. (2003). From knowledge-based to skill-based systems: Sailing as a machine-learning challenge. In *Machine Learning: ECML 2003, 14th European Conference on Machine Learning*, volume 2837 of *Lecture Notes in Computer Science*. Springer.

[Albus, 1975] Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, 97:220–227.

[Atkeson et al., 1997a] Atkeson, C. G., Moore, A. W., and Schaal, S. (1997a). Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73.

[Atkeson et al., 1997b] Atkeson, C. G., Moore, A. W., and Schaal, S. (1997b). Locally weighted learning for control. *Artificial Intelligence Review*, 11(1-5):75–113.

[Aupetit et al., 2000] Aupetit, M., Couturier, P., and Massotte, P. (2000). Function approximation with continuous self-organizing maps using neighboring influence interpolation. In Bothe, H. and Rojas, R., editors, *Proceedings of the ICSC Symposia on Neural Computation (NC'2000) May 23-26, 2000 in Berlin, Germany*. ICSC Academic Press.

[Bagnell and Schneider, 2001] Bagnell, J. and Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the International Conference on Robotics and Automation*.

[Baird, 1993] Baird, L. (1993). Advantage updating. Technical Report WL-TR-93-1146, Wright-Patterson Air Force Base Ohio: Wright Laboratory.

[Baird, 1999] Baird, L. (1999). *Reinforcement Learning Through Gradient Descent*. PhD thesis, Carnegie Mellon University.

[Baird and Klopf, 1993] Baird, L. and Klopf, A. (1993). Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-1147, Wright-Patterson Air Force Base Ohio: Wright Laboratory.

[Baird, 1995] Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, pages 30–37.

[Bertsekas, 1995] Bertsekas, D. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific.

[Bishop, 1995] Bishop, C. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

[Boyan and Moore, 1995] Boyan, J. A. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA. The MIT Press.

[Burden and Faires, 1997] Burden, R. L. and Faires, J. D. (1997). *Numerical Analysis*. International Thompson Publishing, 6th edition.

[Gaskett et al., 1999a] Gaskett, C., Wettergreen, D., and Zelinsky, A. (1999a). Q-Learning in continuous state and action spaces. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence*. Springer-Verlag.

[Gaskett et al., 1999b] Gaskett, C., Wettergreen, D., and Zelinsky, A. (1999b). Reinforcement learning applied to the control of an autonomous underwater vehicle. In *Proceedings of the Australian Conference on Robotics and Automation (AuCRA99)*.

[Goldstein, 1950] Goldstein, H. (1950). *Classical Mechanics*. Addison-Wesley Publishing.

[Gullapalli, 1992] Gullapalli, V. (1992). *Reinforcement learning and its application to control*. PhD thesis, University of Massachusetts.

[Isaac and Sammut, 2003] Isaac, A. and Sammut, C. (2003). Goal-directed learning to fly. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Marchaj, 1964] Marchaj, C. A. (1964). *Sailing theory and practice*. Granada Publishing Limited.

[Ross, 1975] Ross, W. (1975). *Sail Power*. Granada Publishing Limited.

[Santamaria et al., 1998] Santamaria, J., Sutton, R., and Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces.

[Schaal and Atkeson, 1997] Schaal, S. and Atkeson, C. (1997). Receptive field weighted regression.

[Smart and Kaelbling, 2000] Smart, W. D. and Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proc. 17th International Conf. on Machine Learning*, pages 903–910. Morgan Kaufmann, San Francisco, CA.

[Smart and Kaelbling, 2002] Smart, W. D. and Kaelbling, L. P. (2002). Effective reinforcement learning for mobile robots.

[Smith, 2001a] Smith, A. (2001a). Applications of the self-organising map to reinforcement learning. Technical report, University of Edinburgh.

[Smith, 2001b] Smith, A. (2001b). *Dynamic generalisation of Continuous Action Spaces in Reinforcement Learning: A Neurally Inspired Approach*. PhD thesis, University of Edinburgh.

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (1998). *Reinforcement Learning : an introduction*. Adaptive Computation and machine learning. MIT Press.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3).

[Thompson, 2002] Thompson, D. R. (2002). Scaling up spatial credit assignment through modularity. Master's thesis, University of Edinburgh.

[Thrun and Schwartz, 1993] Thrun, S. and Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates.

[Timin, 2004] Timin, M. (2004). AnnEvolve website : `http://www.annevolve.sourceforge.net`.

[Touzet et al., 1997] Touzet, C., Giambiasi, N., and Sehad, S. (1997). Neural reinforcement learning for behavior synthesis. *Robotics and Autonomous Systems, Special issue on Learning Robots: the New Wave*, 22:251–281.

[van Aartrijk and Samoocha, 2003] van Aartrijk, M. and Samoocha, J. (2003). Learning to sail. In *Proceedings of European Symposium on Intelligent Technologies, Hybrid Systems and their implementation on Smart Adaptive Systems*.

[van Aartrijk et al., 2002] van Aartrijk, M., Tagliola, C., and Adriaans, P. (2002). AI on the ocean: the robosail project. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002*.