Reinforcement learning in continuous state- and action-space

Barry D. Nichols

Faculty of Science and Technology

Reinforcement Learning in Continuous State- and Action-Space

Barry D. Nichols

A thesis submitted in partial fulfilment of the requirements of the University of Westminster for the degree of Doctor of Philosophy

September 2014

Abstract

Reinforcement learning in the continuous state-space poses the problem of the inability to store the values of all state-action pairs in a lookup table, due to both storage limitations and the inability to visit all states sufficiently often to learn the correct values. This can be overcome with the use of function approximation techniques with generalisation capability, such as artificial neural networks, to store the value function. When this is applied we can select the optimal action by comparing the values of each possible action; however, when the action-space is continuous this is not possible.

In this thesis we investigate methods to select the optimal action when artificial neural networks are used to approximate the value function, through the application of numerical optimization techniques. Although it has been stated in the literature that gradient-ascent methods can be applied to the action selection [47], it is also stated that solving this problem would be infeasible, and therefore, is claimed that it is necessary to utilise a second artificial neural network to approximate the policy function [21, 55].

The major contributions of this thesis include the investigation of the applicability of action selection by numerical optimization methods, including gradient-ascent along with other derivative-based and derivative-free numerical optimization methods, and the proposal of two novel algorithms which are based on the application of two alternative action selection methods: NM-SARSA [40] and NelderMead-SARSA.

We empirically compare the proposed methods to state-of-the-art methods from the literature on three continuous state- and action-space control benchmark problems from the literature: minimum-time full swing-up of the Acrobot; Cart-Pole balancing problem; and a double pole variant. We also present novel results from the application of the existing direct policy search method genetic programming to the Acrobot benchmark problem [12, 14].

Contents

List of Figures

List of Tables

Declaration

I declare that all the material contained in this thesis is my own work.

Signed: Barry D. Nichols Date: 30 September 2014

Chapter 1

Introduction

1.1 Problem Description

Reinforcement Learning (RL) is the process of learning to perform a given task well, learning only from experimental interactions with the environment, in a similar manner to the way we train pets through rewards and punishment. The term RL can be used either to describe problems of this form, where the only training information is in the form of a reward (or punishment) signal, or to describe a class of solutions applied to these problems.

As RL agents are able to learn the expected long term reward and good policies through experimentation with the environment, without an explicit teacher, it has many application areas which may be very difficult, or impossible, to solve through alternative methods, such as:

- difficult games, e.g. an agent reaching high skill level at Backgammon merely by playing the game [54]
- control problems, e.g. an agent learning to fly a helicopter [39]
- resource allocation problems [45]

RL has had much success in the small discrete state- and action-space setting, and through function approximation has been successfully applied to problems with large, or continuous, state-spaces. There are also some methods which enable the application of RL to problems with continuous action-space; however, the two most commonly applied of these are the actor-critic architecture, which requires the training of an extra function approximator, and direct policy search, which takes a large number of training episodes in order to optimize the policy function using derivative-based or derivative-free methods.

1.2 Aims of This Work

Reinforcement learning in small, discrete state- and action-space can be achieved by storing, in a lookup table, the expected sum of long-term discounted rewards we will receive from being in any given state and taking any of the available actions from that state. When the state-space is either very large or continuous we must utilise function approximation to allow us to generalize between different states as we will be unable to physically store the values of so many states, and we would be unable to visit all states sufficiently often to learn the correct values. But we are still able to compare the expected values of each possible action from the current state to select the one believed to maximize the long-term reward. However, when the action space is large, or continuous, selecting the action which will lead to the highest expected long-term reward also poses a problem. This research investigates different methods of solving this optimization problem to allow fast action selection using the same implicit policy methods which is the preferred approach when the action space is small [21].

In order to achieve this we investigate the current state-of-the-art approaches for solving continuous state- and action-space reinforcement learning problems and the various function approximation techniques which are used to store the expected sum of rewards, as a function of the state and action, and/or the policy function, which stores the action to take from the specified state.

We then attempt to develop novel approaches which are an improvement on the current performance of RL algorithms by overcoming the problems of applying RL in the continuous state- and action-space. These novel approaches will be verified against the state-of-the-art approaches on some well known control benchmark problems from the literature.

1.3 Contributions

The main contributions to knowledge made by this thesis are:

- Description of two novel approaches to action selection through application of optimization methods, which provide:
    - faster action selection than gradient ascent, which is stated as an approach in the literature [47]
    - no discretization of the state- or action-space
    - training on continuous problems without a separate policy function
- Comparison of performance of different optimization methods to the action selection

- Novel results of these and the existing approaches of genetic programming on the Acrobot benchmark problem from the literature

1.4 Thesis Outline

The rest of this thesis is presented as follows. Firstly, the following three chapters present the necessary background material required to appreciate the rest of the thesis. This consists of the background of: reinforcement learning (Chapter 2) including the notation used throughout the thesis and the basic algorithms of RL; artificial neural networks (ANN) (Chapter 3) describing the architectures and training methods of several ANNs which are commonly applied to approximating the value function and/or policy function in RL; and finally evolutionary algorithms (Chapter 4) describing both genetic algorithms and genetic programming, which are two approaches based on evolution in nature applied to evolving candidates which optimize a fitness function, in RL this fitness function is the sum of rewards received in an episode and these methods are applied to direct policy search.

This is followed by a more in-depth description of the algorithms used to apply RL to continuous action-space problems, which make use of all of the techniques covered in the background. This includes both a review of the state-of-the-art continuous action-space RL algorithms (Chapter 5) and then presents the details of some novel implicit policy methods which take advantage of optimization methods to select the action rather than discretizeing the action-space (Chapter 6).

Thirdly we present the application of several of the algorithms described to three difficult continuous state- and action-space control benchmark problems from the literature to show how the novel algorithms compare to the state-of-the-art empirically (Chapter 7). The control problems are: the Acrobot (Section 7.1) which also includes novel results from existing techniques, the Cart-Pole (Section 7.2) and the double Cart-Pole (Section 7.3) which are both compared to the CACLA algorithm as it has been shown to outperform many other RL approaches in the literature [47]. The novel approaches compare favourably to the state-of-the-art on all three benchmarks.

And finally Chapter 8 presents a brief summary of the contents of the thesis; a discussion of the results of this research; and directions for future work.

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) [30, 45, 53] is a machine learning technique whereby an agent learns, through interaction with the environment, a policy of which actions to take from any given state. The environment encompasses all information relating to the application domain, including details unobservable by the agent, such as a model of the dynamics of a control problem or the locations of all cards in a poker game. The state is a representation of the information the agent should reasonably be made aware of in order to act in the current environment, e.g. the cards the agent has in a poker game, the betting pot and the chips each player has. The state signal is passed to the agent from the environment, and the action is the signal the agent sends to the environment which then affects the environment and results in the transition from the current state to another state. In the poker example a reasonable set of actions may be {bet, check, raise, fold}, which would not all be available from every state, e.g. if the previous player has placed a bet the agent cannot check, but must either bet, raise or fold.

Here *agent* refers to a computer program with the ability to interact in some way with the environment and to autonomously select actions, based on the information it observes about the current state of the environment, and can update this policy based on positive or negative reinforcement signals, which are received from the environment.

In order for the agent to learn which actions are better than others from a given state, there must be some measure of how good it is to be in a particular state in the environment the agent is placed in. This takes the form of a reward which is the feedback the agent receives from the environment as a measure of the quality of the transition from the current state to the resulting state when applying the selected action.

Thus, the task of a RL agent is to learn a policy which maximises this reward signal, in order to perform optimally in the environment, this of course assumes the rewards are correctly aligned with the goal to be achieved (see Section 2.1 for details

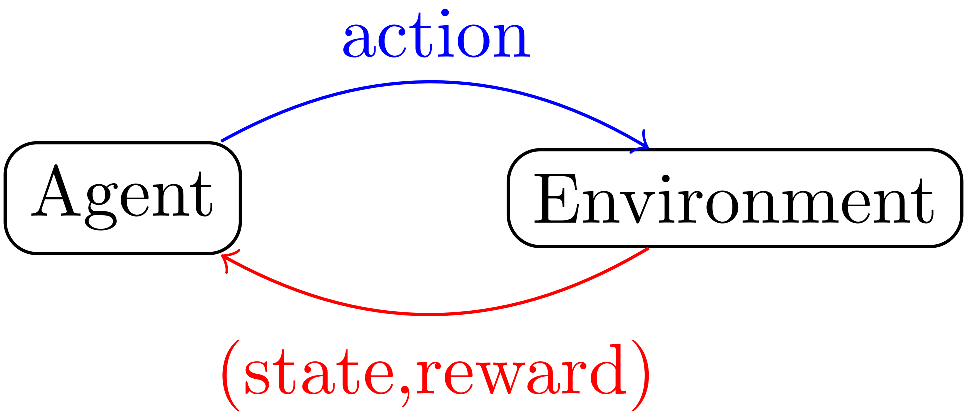


Figure 2.1: The interaction between the RL agent and the environment. At each time-step the agent selects an action based on the current state, which it applies to the environment after which the environment notifies the agent of the next state and immediate reward resulting from the state transition.

on how the RL problem is formulated). The way in which the agent interacts with the environment through these signals can be seen in Fig. 2.1.

It is important to note that the goal of RL is to maximise the long-term reward rather than the immediate reward at each time-step which may involve taking, seemingly, suboptimal actions in order to achieve greater rewards in the future. For example at certain times of the day it may be quicker to take a longer route rather than queuing up to take the short-cut with everyone else.

In order to explain how the agent learns to optimize the rewards received, we first describe the notation used to model the problem to be solved using RL in Section 2.1, then in Section 2.2 we describe the basics and the algorithms of dynamic programming, which is a method of solving RL problems when a full model of the environment is available. Then we expand this to general reinforcement learning in which a model of the environment is not assumed (Section 2.3). And in Section 2.4, Section 2.5 and Section 2.6 we discuss the main obstacles involved in applying RL, including the exploration exploitation trade off, slow leaning and applying algorithms in continuous state- and action-spaces, along with some commonly applied approaches to overcoming such difficulties. Finally Section 2.7 contains a summary of the material covered in this chapter.

2.1 Markov Decision Process

A problem which is to be solved by RL is first formulated as a *markov decision process* (MDP). An MDP is a mathematical model of a decision process which comprises:

- A set of possible states $\mathcal{S}$

- A set of possible actions $\mathcal{A}$
- A state transition function: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$
- A reward function: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$

each of which is described in more detail in the following sections.

Another key factor of formulating a task as an MDP is that it must satisfy the markov property, which means the information contained in one state is sufficient to optimize future actions, without knowledge of previous states.

2.1.1 State-Space

The state-space is the set of all possible states $\mathcal{S}$, it is important to define the state-space such that the *markov property* holds. The state-space may be discrete or continuous, vector or scalar valued. In some cases a good state representation may be obvious from the application in other cases it may be necessary to construct a, somewhat, artificial state representation in order to utilise symmetries in the state-space, or in an attempt to construct a more hierarchical representation.

An example of a state-space representation which satisfies the markov property is the current layout of a Chess board: the sequence of moves leading to the current board layout is not required for a player to select his next move; thus, the raw board layout satisfies the markov property and could be used. Of course with such a large state-space it would be useful to identify other features to augment the state representation and, therefore, assist in the learning process, but identifying such features may require expert knowledge of the application domain.

Although it is possible to apply RL to problems where the markov property does not hold: partially observable MDPs (POMDPs) [49] where, e.g. due to noisy sensors, the state signal may not be distinguishable from other states. Such difficulties are not considered here.

It is also possible, in some applications, to use the afterstate [53] (also known as post-decision state [45]). The afterstate is the state with the action applied to it, but before the state transition function has been applied. One obvious example of this is in a board game, where the afterstate is the board layout after the agents move, but before the opponents move. This fully utilises all the information available to the agent, and because many state-action pairs may lead to the same afterstate can accelerate learning [53], and can also be utilised in applications with much higher dimensional action-spaces [45].

2.1.2 Action-Space

The action-space $\mathcal{A}$ is the set of all possible actions which may be taken from the current state. In some problems, the available actions may vary according to the current state, and therefore, the action-space is often expressed as $\mathcal{A}(s)$.

As with the state-space, the action-space may be discrete or continuous, scalar or vector valued. The action representation also depends on the application, and on the solution method: some approaches involve discretization of continuous action-space in order to avoid the optimization problem for action selection [24, 47].

2.1.3 State Transition Function

The state transition function maps the probability of making the transition from $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ when applying $a \in \mathcal{A}(s)$. We denote this probability with the notation:

$$\Pr(s'|s,a) = \Pr(s_{t+1} = s'|s_t = s, a_t = a). \tag{2.1}$$

Again this function is very problem specific: some may be deterministic, others stochastic. It may also depend on how the task is formulated.

2.1.4 Reward Function

The reward function gives the expected reward received after taking a in s and as a result transitioning to s':

$$r(s,a,s') = \mathbb{E}\left\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{2.2}$$

Although we include s, a and s' in this notation, in practice this function may be defined in terms of just s and a or just s', it is possible that the reward could be deterministic, this is largely dependent on the problem.

Although some applications may have an inherent reward function: e.g. portfolio management [45], for many others this function will have to be defined by the designer to indicate the goal of the task. For a disaster avoidance task, one example of which is balancing a pole on a cart (Section 7.2), the reward may simply be 0 at all time-steps until failure, when a negative reward is received. In problems where a minimum time solution is required the reward may be a negative value at all time-steps until the agent is successful, e.g. the minimum-time swing-up of the acrobot (Section 7.1).

In other cases a more sophisticated reward function may be applied to include several features of desired performance. It is important to ensure the reward function only defines the desired outcome and not the method of achieving the desired outcome. In the chess example if the defined reward function punishes the agent for having pieces taken by the opponent, this may result in the agent maximizing the number of remaining pieces even at the expense of the game [53].

2.2 Dynamic Programming

Dynamic programming (DP) [5] is the predecessor to RL, which seeks to optimize the long term sum of a reward received at each discrete time-step of a task. This is achieved by first separating the sum of all rewards into the immediate reward and the sum of all future rewards in order to apply a recursive solution which maximizes the sum of the immediate reward and the sum of all future rewards.

RL extends DP to be applicable to problems where a model of the environment is not available and/or the state- and action-space are too large to evaluate every possible action from every possible state until the value function and/or policy has converged. Thus DP can be seen as a special case of RL when a model of the environment is available, and is often viewed as such by the RL community [10, 28, 30, 53].

Using the notation described in Section 2.1, DP seeks to select the action at each state to maximize the sum of discounted rewards:

$$\sum_{t=0}^{T-1} \gamma^t r_{t+1} \tag{2.3}$$

with the undiscounted setting included as a special case where $\gamma = 1$; however, typically γ is set to values such as 0.8 or 0.9. In certain applications, such as financial applications where the value of money changes with time, the value of γ may take a special meaning in the application domain. But, in general, it both ensures the sum remains finite and controls how much preference is given to receiving rewards sooner.

The mechanism employed by DP to separating the sum of rewards into a recursive problem is the value function (Section 2.2.1), after describing the value function we go on to describe the main algorithms of DP: policy iteration (Section 2.2.2) and value iteration (Section 2.2.3).

2.2.1 Value Function

The principal component of DP is the value function, which divides the maximization of the sum of long term discounted rewards into a recursive problem of maximizing the sum of immediate rewards and the sum of all future rewards, where the sum of all future rewards can be approximated by the value function at the following time step.

The value function is the expected sum of discounted rewards:

$$V(s_t) = \mathbb{E}\left\{ \sum_{k=0}^{T-(t+1)} \gamma^k r_{t+k+1} \right\} \tag{2.4}$$

which is split into the immediate and all future rewards:

$$V(s_t) = \mathbb{E}\left\{ r_{t+1} \right\} + \gamma\, \mathbb{E}\left\{ \sum_{k=0}^{T-([t+1]+1)} \gamma^k r_{[t+1]+k+1} \right\} \tag{2.5}$$

and the value function at the next time-step is substituted for the sum of all future rewards:

$$V(s_t) = \mathbb{E}\left\{ r_{t+1} \right\} + \gamma V(s_{t+1}) \tag{2.6}$$

The equation which optimizes the value function $V(s)$ is known as the Bellman optimality equation:

$$V^*(s) = \max_a \mathbb{E}\left\{ r(s,a,s') + \gamma V^*(s') \right\} \tag{2.7}$$

which is the maximum sum of discounted rewards. DP methods and some classes of RL methods attempt to find approximate solutions to this equation in order to find an optimal policy. The two main approaches to DP: *policy iteration* and *value iteration* are described below.

2.2.2 Policy Iteration

Policy iteration relies on iteratively calculating the value function $V(s), \quad \forall s \in \mathcal{S}$ based on following the current policy and then calculating an improved policy by setting it to the action which optimizes $V(s)$ assuming the value of selecting all future actions according to the current learnt policy:

$$\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}(s)} \sum_{s'} \Pr(s'|s,a) \left[r(s,a,s') + \gamma V(s') \right], \quad \forall s \in \mathcal{S} \tag{2.8}$$

until there are no changes to the policy.

2.2.3 Value Iteration

Value iteration, on the other hand, calculates $V(s)$ by selecting the maximum action rather than from following a policy using Equation (2.9), until the changes in the value function are very small.

$$V(s) \leftarrow \max_a \sum_{s'} \Pr(s'|s,a)[r(s,a,s') + \gamma V(s')], \quad \forall s \in \mathcal{S} \tag{2.9}$$

Thus, approximately solving $V^*(s)$, after which the policy which maximizes the value function can be calculated using (2.10) if required.

$$\pi(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s'|s,a)[r(s,a,s') + \gamma V(s')], \quad \forall s \in \mathcal{S} \tag{2.10}$$

One of the largest problems with DP is that every state and action must be visited in order to solve the approximation of the value function. This is especially important as the number of possible states and actions increases exponentially with an increase in the dimensionality of the state- or action-space, this is known as the *curse of dimensionality.* The curse of dimensionality is one of the problems RL methods attempt to overcome by updating only the value function for states which are visited by the agent.

2.3 Reinforcement Learning Methods

There are three main approaches to the learning of policies with RL: *Temporal Difference* (TD) learning, *Actor-Critic* (AC) methods and *direct policy search* (DPS). TD and AC both store estimates of the value function, but TD calculates the policy from the value function whereas AC also explicitly stores the policy function. DPS only stores the policy function. Here each of these methods will be briefly discussed individually.

2.3.1 Temporal Difference Learning

Initially the value functions are unknown and must be learnt through experimentation with the environment. Temporal difference (TD) learning [52] is one method of updating the value function. TD learning updates the policy at each time-step, and therefore takes advantage of the updated estimate when deciding future actions within the same episode. TD learning is done by, at each time-step, calculating the TD error δ (2.11), which is the difference between the estimate of $V(s_t)$ made at the previous time-step and the sum of the immediate reward and the discounted estimate of the sum of all future rewards: $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$. $V(s)$ is then updated to reduce the TD error:

$$\delta = r(s,a,s') + \gamma V(s') - V(s) \tag{2.11}$$

by applying:

$$V(s) \leftarrow V(s) + \alpha\delta \tag{2.12}$$

This method clearly relies on the estimate at the next time-step in order to update the estimate at the current time-step, and therefore several episodes will be required to establish accurate value estimates of states occurring towards the beginning of the trajectory which take into account rewards received later in the trajectory. This can be improved by eligibility traces (Section 2.5).

It is not always possible to select actions based on $V(s)$, which requires a model of the transition function. In such cases the state-action value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is applied instead. The TD error is defined in a similar way, however, as there is no estimate of $V(s)$ two different approaches exist for estimating the sum of future rewards using the Q function. One option is to base the update on the action selected at time-step $t + 1$, which leads to the SARSA algorithm:

$$\delta = r(s, a, s') + \gamma Q(s', a') - Q(s, a) \tag{2.13}$$

The alternative is to use the value of the greedy action at time-step $t + 1$, which results in the Q-learning algorithm:

$$\delta = r(s, a, s') + \gamma \max_{\hat{a} \in \mathcal{A}(s')} Q(s', \hat{a}) - Q(s, a) \tag{2.14}$$

The difference, in RL terminology, is that SARSA uses on-policy updates, and Q-learning uses off-policy updates. Which means SARSA applies the learnt policy to select actions when interacting with the environment (on-policy), whereas Q-learning uses a different policy for action selection and training (off-policy): updates assume the greedy action will be taken but actually exploratory actions may be taken. This can lead to some differences in learnt functions and learnt policies: SARSA takes exploration into consideration and therefore learns a safer policy if exploration leads to poor rewards, whereas Q-learning ignores the values of such exploratory actions and therefore may learn a policy in which the long-term expected reward is maximized in episodes where the greedy policy is followed; however any exploratory actions, or noise, may have devastating consequences, as is illustrated by the cliff walking example in [53, p. 149].

2.3.2 Actor-Critic

Actor-Critic (AC) based methods were among the earliest approaches to solving RL problems [3]; however, despite the fact that generally, when it is possible to do so, preference is given to implicit policy methods [21] (also referred to as critic-only methods), AC methods remain popular largely due to their applicability to continuous action-space without searching for the optimal action directly [53], particularly as it is often stated that searching of the action which maximizes the value function is not feasible [21, 55]. Another reason for the survival of AC methods is their ability to learn a stochastic policy, which is particularly applicable in non-Markov problems [53, p.153].

2.3.3 Direct Policy Search

Direct policy search (DPS) methods (sometimes referred to as actor-only methods), including evolutionary algorithm based methods and policy gradient methods, attempt to search for optimal policies without learning the value function. They typically attempt to solve a slight variation of the RL problem:

$$\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}\left\{ \sum_{t=0}^{T} \gamma^t r_t \;\middle|\; \boldsymbol{\theta} \right\} \tag{2.15}$$

which is the expected sum of discounted rewards using the policy parameter vector $\boldsymbol{\theta}$. Thus, DPS seeks to optimize the policy by adjusting the parameters of the policy without attempting to solve $V(s)$. One exception to this is the genetic programming approach (discussed in more detail in Section 5.2.3) which not only adjusts the parameters of the policy, but the policy function also has no fixed structure and is updated through evolutionary operations.

2.4 Exploration vs Exploitation

During learning, particularly at the earlier stages when an arbitrary policy is used, it is important to explore the environment as well as exploiting the current policy in order to find action choices which may lead to state trajectories which are an improvement over the current policy. There are several methods of achieving this, the most popular of which are ε-greedy in the discrete action-space setting, and Gaussian exploration in the continuous action-space.

2.4.1 ε-Greedy

In the ε-greedy exploration method an exploration parameter ε controls the exploration exploitation trade off. With probability ε an exploratory action is selected at random, and with probability $1 - \varepsilon$ the greedy action is taken, to exploit the learnt policy. This parameter may be constant throughout learning, or may start high when there is no knowledge about the optimal value function and reduced as learning progresses.

2.4.2 Softmax Exploration

An alternate exploration technique, in the discrete action-space setting, is the softmax method which, by applying the Boltzmann distribution, selects actions according to probability based on their expected value rather than selecting from all exploratory actions with equal probability, as is the case with ε-greedy [53]. The distribution includes a temperature parameter τ, which controls the extent to which the action selection probabilities are affected by the difference in Q values.

$$\Pr(a_t = a | s_t = s) = \frac{\exp\left[Q(s,a)/\tau\right]}{\sum_{a_j \in \mathcal{A}(s)} \exp\left[Q(s,a_j)/\tau\right]} \tag{2.16}$$

For simplicity in (2.16) we assumed the use of $Q(s,a)$, of course, if $V(s)$ is being used, it is possible to define the same distribution in terms of $V(s')$, where s' is calculated using the state transition probabilities (2.17).

$$\Pr(a_t = a | s_t = s) = \frac{\exp\left[\sum_{s'} \Pr(s'|s,a) V(s')/\tau\right]}{\sum_{a_j \in \mathcal{A}(s)} \exp\left[\sum_{s'} \Pr(s'|s,a_j) V(s')/\tau\right]} \tag{2.17}$$

Despite the reduced danger of selecting disastrous exploratory actions ε-greedy is still the more commonly applied of the two methods, possibly due to the difficulty involved in selecting the reduction schedule of τ [53, p.31]. Also, bad exploratory actions may be less important is some applications, particularly when learning is performed on computer simulations.

2.4.3 Gaussian Exploration

Gaussian exploration does not select random actions uniformly, as is the case with ε-greedy. Instead a random value is selected from a zero mean Gaussian distribution, where the standard deviation is an adjustable parameter to control the exploration. This random value is then added to the selected action, i.e.:

$$a \leftarrow a + \sim \mathcal{N}(0, \sigma^2) \tag{2.18}$$

As with the other exploration methods discussed, the exploration may be reduced as learning progresses by reducing the standard deviation.

2.5 Eligibility Traces

In TD learning only the estimate of the value function for the current state is updated at each time-step, which can lead to slow learning, particularly with delayed rewards e.g. a board game where the reward is 1 if the agent wins and -1 if the agent loses.

An extension to TD in order to improve learning efficiency is to not only update the current state, but also to update, to a lesser extent, all previous states in the trajectory. This is known as the eligibility trace.

This technique can be applied to many TD based RL algorithms, and the version of such algorithms is typically denoted by appending λ to their names, e.g. TD(λ), SARSA(λ), Q(λ). Named after the eligibility decay parameter λ, which controls to what degree states at previous time-steps are eligible for the immediate reward received at the current time-step.

Eligibility traces are implemented by making the following slight adjustments to the value update equation:

$$\begin{aligned} &e(s_t) \leftarrow e(s_t) + 1 \\ &V(s_k) \leftarrow V(s_k) + \alpha\delta e(s_k), \quad e(s_k) \leftarrow \gamma\lambda e(s_k), \quad k = t, t-1, \ldots, 0 \end{aligned} \tag{2.19}$$

where $e(s)$ is the eligibility value of the given s and the other parameters are as standard TD learning (Section 2.3.1).

The update (2.19) could be performed on every state visited since the start of the episode, however, this may be very time consuming for long trajectories; moreover, states visited several time-steps ago will not be significantly affected due to the eligibility decay parameter λ and, thus, may be excluded from the update in order to minimize the increase in update time [53]. For SARSA or Q-learning the update equations use $Q(s, a)$ and $e(s, a)$ rather than $V(s)$ and $e(s)$ respectively.

There are two methods of applying eligibility trace: *accumulating traces* and *replacing traces*. In (2.19) we utilised an accumulating trace, which takes its name from the fact that the eligibility of the current state s is $e(s) \leftarrow e(s) + 1$, i.e. the eligibility keeps accumulating. It is also possible to apply a replacing trace, where we use $e(s) \leftarrow 1$, i.e. the eligibility is replaced with the value of one. This method can be used to overcome the problem of assigning a disproportionally large amount of credit to states which are visited often, but did not significantly contribute to the reward [53].

2.6 Continuous State- and Action-Space

When the state- and action-space are both discrete and sufficiently small the value function $Q(s, a)$ may be stored in a lookup table, but as the state space grows this becomes infeasible due to both storage size and the fact that states may only be visited once; hence, values of unvisited states must be generalized from the values of visited states.

2.6.1 Continuous State-Space

The most common solution is to apply function approximation to the storage of the value function $V(s)$. This may solve both problems, depending on what function approximation technique is used, by both allowing the storage of any value function with continuous state-space and also making it possible to generalise values of one state based on values of others.

The optimal action may be selected quickly and easily if the action space is discrete and relatively small, by simply evaluating:

$$Q(s,a), \quad \forall a \in \mathcal{A}(s) \tag{2.20}$$

or the resulting $V(s)$ of applying all possible actions if the transition function is known or afterstates are being used and selecting the action which maximizes the function.

2.6.2 Continuous Action-Space

However, for applications with a larger action-space this becomes impractical for moderately large action-spaces, and impossible when the action-space is very large or continuous. The problem is not due to the storage or the training of the value function: a global function approximator, such as an MLP, can be trained to represent any function with arbitrary accuracy [25], it is the selection of the optimal action which leads to difficulty.

Some approaches apply discretization to the action-space to reduce it to a practical size, e.g. using one-step-search [47], or applying k-nearest neighbours based techniques to the storage of the value function [24]. However, the majority of approaches to the continuous action-space avoid this obstacle completely by employing function approximation to the explicit storage of the policy either, in addition to the value function, in the form of actor-critic [48, 56] or, without storing the value function at all, through direct policy search [14, 15, 46]. A more complete discussion of these, and other, approaches to the continuous state- and action-space problem can be found in Chapter 5.

2.6.3 Applying Eligibility Traces with Function Approximation

When linear function approximation is applied the values are not directly stored for each state, but for a collection of features which represent the state, e.g. the vector output of the function $\boldsymbol{\phi}$:

$$V(s) = \boldsymbol{\phi}(s)\boldsymbol{\theta} \tag{2.21}$$

in this case the eligibility trace can be stored as a vector of the same length as the feature vector and applied as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\boldsymbol{e} \tag{2.22}$$

where the TD error, as usual (for SARSA) is calculated using:

$$\delta \leftarrow r(s, a, s') + \gamma Q(s', a') - Q(s, a) \tag{2.23}$$

and the eligibility trace $\boldsymbol{e}$ is calculated using:

$$\boldsymbol{e} \leftarrow \gamma\lambda\boldsymbol{e} + \nabla_{\boldsymbol{\theta}} Q(s, a). \tag{2.24}$$

as is described in [53].

When certain non-linear function approximation techniques (e.g. MLP) are applied each of the weight vectors, or matrices, must have its own eligibility value of equal dimensions, which can be updated similarly.

2.7 Chapter Summary

In this chapter we have introduced the basics of RL including the notation which will be used throughout this thesis. We then went on to describe briefly some of the difficulties which exist within RL including the exploration exploitation trade off and the problems that arise when applying RL to applications with continuous state- and action-space, along with some of the techniques commonly adopted in order to overcome them. This thesis focusses on overcoming the difficulties associated with applying RL to continuous state- and action-space problems; hence, this problem is revisited in Chapter 5 where we discuss the state-of-the-art continuous RL algorithms in detail.

Chapter 3

Artificial Neural Networks

In this chapter we will introduce a selection of artificial neural networks (ANNs) [18, 25] which are commonly applied to value and/or policy function approximation in RL. ANNs are inspired by the way biological neural networks, i.e. the brains of animals are believed to function.

Although all ANNs take their inspiration from the same set of ideas, there are many alternative architectures which have differences resulting in different advantages and disadvantages when being applied to solving a given problem. There are three classes of problems to which ANNs are typically applied: clustering, classification and function approximation. Some ANN architectures are particularly well suited to solving one of these classes of problems, but not others. Here we limit our presentation to those suitable for function approximation, focussing particularly on the architectures commonly applied to value and policy function approximation in continuous state- and action-space RL.

Firstly, in Section 3.1 we describe the most basic ANN architecture: the perceptron, which is a single layer neural network capable only of approximating linear functions. After which we extend this to the multilayer perceptron in Section 3.2, which, by incorporating a hidden layer of nodes with non-linear activation functions, is able to approximate non-linear functions. Then we examine alternative architectures in the form of Radial Basis Functions (Section 3.3) and CMAC (Section 3.4). After presenting these main ANN architectures, we summarise the material presented in this chapter and compare the architectures for applicability as function approximation methods for continuous state- and action-space RL in Section 3.5.

3.1 Perceptron

The Perceptron comprises a layer of inputs and a single layer of neurons, which here we will restrict to a single output node as we are only interested in approximating scalar

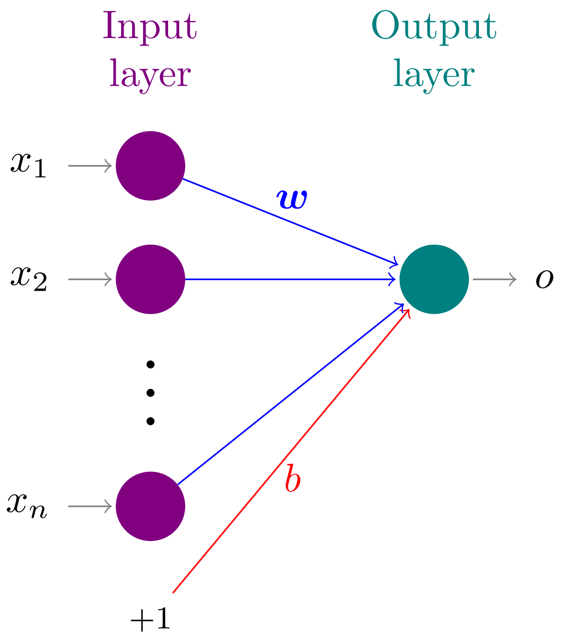


Figure 3.1: Diagram of the perceptron artificial neural network. Weights are shown in blue and bias in red.

functions; however, any number of nodes may be used in the output layer, the number of which will be determined by the problem to be solved, e.g. if the ANN is being applied to approximating a policy function the quantity of output nodes will match the dimensionality of the action-space.

The structure of the perceptron can be seen in Fig. 3.1, the perceptron's response to an input signal $\boldsymbol{x}$ is generated by first calculating the input to the output node o_in by sending the input signal $\boldsymbol{x}$ to the output node via the synaptic weights $\boldsymbol{w}$ where the sum of these values and the bias are calculated:

$$o\_in = b + \sum_{i=1}^{n} x_i w_i \tag{3.1}$$

then the final output o is generated by applying the activation function f to the sum of inputs to the node:

$$o = f(o\_in) \tag{3.2}$$

The activation function is a mathematical function, applied to the sum of inputs to a neuron in order to produce an output signal limited to the desired range, and, in the case of MLPs, to allow non-linear functions to be approximated by the ANN.

The Activation function is normally applied to limit the output of the neuron to a given range, however, sometimes it may limit the output to binary values. Possible activation functions include linear, sigmoid, bipolar sigmoid, hyperbolic tangent. For binary outputs it is also possible to use step functions, such as:

$$f(x) = \begin{cases} 1, & \text{if} \quad x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

however, such activation functions will not be discussed further as they are not applicable when the ANN is utilised to the approximation of a value function, or the approximation of a policy function in continuous action-space.

3.1.1 Training

Training the perceptron for continuous outputs can be accomplished using the *delta rule* where the contribution of each weight to the total error of the output is calculated. There are alternative training algorithms, such as the Hebb rule, which may be employed when the output is limited to binary or bipolar values by a step activation function, however, as we are only interested in continuous outputs and therefore such methods will not be discussed here.

The error we will assume we are attempting to minimize here is the mean squared error (MSE):

$$E = \frac{1}{p} \sum_{i=1}^{p} (t_i - o_i)^2 \tag{3.4}$$

where p is the total number of patterns we have to train the ANN with and o_i is the network response and t_i is the desired response to the input vector i.

Here we focus on the iterative, rather than batch update, and therefore, apply the error $e = (t - o)^2$ where t and o are the target and output values for the current input vector $\boldsymbol{x}$. Therefore for each pattern we seek to minimize the error given by:

$$e = \frac{1}{2} (t - o)^2 \tag{3.5}$$

where the coefficient $\frac{1}{2}$ is included to cancel out multiplication by 2 which will result from calculating the derivative of the error w.r.t. $\boldsymbol{w}$, which is used to obtain the contribution of each weight value to the error, and the final weight update will be:

$$\Delta \boldsymbol{w} = \eta \cdot -\frac{\partial e}{\partial \boldsymbol{w}} \tag{3.6}$$

We update the weights in the negative direction of the gradient $\nabla_w e$, for which each element is calculated as follows:

$$\begin{aligned}\frac{\partial e}{\partial w_i} &= \frac{\partial e}{\partial o}\frac{\partial o}{\partial o\_in}\frac{\partial o\_in}{\partial w_i} \\ &= \frac{\partial}{\partial o}\left[\frac{1}{2}(t-o)^2\right]\frac{\partial}{\partial o\_in}\left[f(o\_in)\right]\frac{\partial}{\partial w_i}\left[b+\sum_{j=1}^{n} x_j w_j\right] \\ &= -(t-o)f'(o\_in)x_i\end{aligned} \tag{3.7}$$

where o_in is the input to the output node.

Often, in order to simplify the update equations, δ is defined as:

$$\delta = -\frac{\partial e}{\partial o\_in} \tag{3.8}$$

this also leads to more uniform update equations when we expand this to the multilayer perceptron. By applying this variable and substituting (3.7) into (3.6) we arrive at the weight update vector:

$$\Delta w_i = \eta \delta x_i \tag{3.9}$$

and the bias update value is:

$$\Delta b = \eta \delta \tag{3.10}$$

due to the fact the input to the bias is always 1.

These update values are then added to the current weight and bias values to arrive at the values used at the next iteration:

$$\begin{aligned}\boldsymbol{w} &\leftarrow \boldsymbol{w} + \boldsymbol{\Delta w} \\ b &\leftarrow b + \Delta b\end{aligned} \tag{3.11}$$

which are then applied to generating future output values which may, if training is still taking place, be used to generate $\Delta \boldsymbol{w}$ at the following iteration.

3.2 Multilayer Perceptron

The multilayer perceptron (MLP) [18, 25] is an extension on the single layer perceptron (Section 3.1) which includes one or more *hidden layers* which have a non-linear activation function, which allows the approximation of non-linear functions. Although it is possible to have any number of hidden layers, it has been proven that a single hidden layer is sufficient to approximate any function, providing sufficiently many nodes are in the hidden layer [18, 25]; thus, a single hidden layer is often used. However, in some cases it may be beneficial to include a second hidden layer [18].

The MLP is a global function approximator and can be trained to approximate any function to arbitrary accuracy [25]. As can be seen in Fig. 3.2 on the following page, the MLP takes a vector input $\boldsymbol{x}$, of n elements, at the input layer. These inputs are sent to the hidden layer via the weight matrix $\boldsymbol{V}$ and the summation of all inputs to each hidden node are computed, including the bias values $\boldsymbol{b}_1$:

$$h\_in_j = b_{1_j} + \sum_{i=1}^{n} x_i v_{i,j} \tag{3.12}$$

and then the non-linear activation function f is applied to generate the output value from each of the m hidden node:

$$h_j = f(h\_in_j) \tag{3.13}$$

The outputs from the hidden layer are then multiplied by the weights vector $\boldsymbol{w}$ as they are sent to the output node o where the sum of these values and the bias b_2 are computed:

$$o\_in = b_2 + \sum_{j=1}^{m} h_j w_j \tag{3.14}$$

and finally the output activation is applied resulting in the final output value:

$$o = g(o\_in) \tag{3.15}$$

The activation at the output node g may also be non-linear, to scale the output to a desired range, e.g. $[-1, 1]$; however, it could also be linear in which case the function g may simply output the value of its input, i.e. $o = o\_in$.

3.2.1 Training

Training the MLP can be achieved in an on-line fashion similarly to the delta rule of the perceptron 3.1, by applying the *generalised delta rule*, which is an extension of the delta rule in order to allow the updating of the weights leading to the hidden layer(s). This type of learning is also known as *backpropagation* as the error from each layer, starting from the output layer, is *backpropagated* to calculate the error at the previous layer.

We begin in a similar fashion to the single layer perceptron, by calculating $\nabla_{\boldsymbol{w}} e$:

$$\begin{aligned}\frac{\partial e}{\partial w_j} &= \frac{\partial e}{\partial o}\frac{\partial o}{\partial o\_in}\frac{\partial o\_in}{\partial w_j} \\ &= \frac{\partial}{\partial o}\left[\frac{1}{2}(t-o)^2\right]\frac{\partial}{\partial o\_in}[g(o\_in)]\frac{\partial}{\partial w_j}\left[b_2 + \sum_{k=1}^{m} h_k w_k\right] \\ &= -(t-o)g'(o\_in)h_j\end{aligned} \tag{3.16}$$

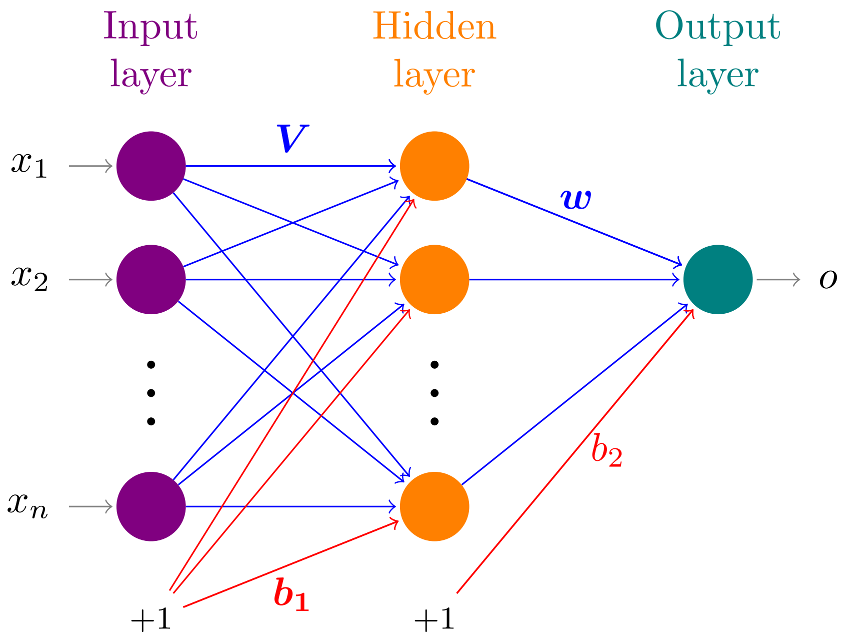


Figure 3.2: Diagram of the multilayer perceptron artificial neural network. Weights are shown in blue and bias in red.

and as with the single layer perceptron, we substitute:

$$\delta = -\frac{\partial e}{\partial o\_in} \tag{3.17}$$

and arrive at:

$$\frac{\partial e}{\partial w_j} = -\delta h_j \tag{3.18}$$

allowing us to compute the weight update values at the output layer:

$$\Delta w_j = \eta \delta h_j \tag{3.19}$$

with the bias being:

$$\Delta b_2 = \eta \delta \tag{3.20}$$

However, we cannot apply these updates before calculating the update values of the first layer weights $\boldsymbol{V}$.

As we do not have targets for the outputs of the hidden nodes we are unable to compute their update in the same way, and therefore we require backpropagation. We propagate the error terms from the output δ back along the respective weights w to generate an error term for each of the hidden nodes:

$$\delta_j = \delta w_j f'(h\_in_j) \tag{3.21}$$

from which, along with $\partial h\_in_j / \partial v_{i,j} = x_i$, we can compute the weight updates for the first layer weights $\boldsymbol{V}$:

$$\Delta v_{i,j} = \eta \delta_j x_i \tag{3.22}$$

with the first layer bias updates:

$$\Delta b_{1_j} = \eta \delta_j \tag{3.23}$$

and finally all updates can be applied to the weights:

$$\begin{aligned} \boldsymbol{V} &\leftarrow \boldsymbol{V} + \boldsymbol{\Delta V} \\ \boldsymbol{b_1} &\leftarrow \boldsymbol{b_1} + \boldsymbol{\Delta b_1} \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + \boldsymbol{\Delta w} \\ b_2 &\leftarrow b_2 + \Delta b_2 \end{aligned} \tag{3.24}$$

3.2.2 Momentum

As the MLP is known to have problems of slow convergence and the weights are susceptible to becoming 'stuck' in local minima of the error. A common extension to the backpropagation training method described above is to include a momentum term. The momentum parameter μ controls the amount of weight given to the momentum in the update.

The idea behind the use of momentum is that by including the previous $\Delta\boldsymbol{w}$ when calculating the current $\Delta\boldsymbol{w}$ the update applied will be larger if the current and previous updates are in the same direction. This allows the weights to more quickly converge to the minimum values and, more importantly, skip past local minima in the error function. The weight updates with momentum are:

$$\Delta v_{i,j} \leftarrow \eta \delta_j x_i + \mu \Delta v_{i,j} \tag{3.25}$$

and

$$\Delta w_j \leftarrow \eta \delta h_j + \mu \Delta w_j \tag{3.26}$$

There exist other variations and extensions to strive for better performance with the MLP architecture, such as variable learning rates [18]; however, as we do not apply such extensions in this work they are not discussed further here.

3.3 Radial Basis Function

The radial basis function (RBF) neural network [25] is different to the perceptron and the MLP in that the first layer of weights $\boldsymbol{V}$ are not coefficients to multiply the inputs by, but are a set of points in the input space serving as centres for the basis functions

(BFs), therefore the closer the input is to one of these centres the larger the output from the associated hidden node will be.

Fig. 3.3 presents a visual representation of the activation of the RBF nodes for 1-dimensional input. For inputs with 2-dimensions the Gaussian function will be a 3-dimensional bell shape, for higher-dimensions the graphic representation is no longer possible. However, in any dimension the first layer weights can be seen as being the centres c of the BFs and it is the distance from this c which is used in calculating the nodes output.

Each hidden node has a different centre c and therefore will have a different output from the same input signal. The nodes may or may not have different values of σ, which controls the width of the BF. The BFs may be evenly spaced throughout the input space; randomly positioned; or the centres may be selected by some form of clustering analysis. As the data is not available beforehand in RL problems evenly spaced basis functions is the most widely applied approach [10, 35].

The output of the RBF is the sum of the outputs of the hidden nodes multiplied by the output weight vector, as with the perceptron and MLP. The structure of the RBF ANN can be seen in Fig. 3.4, and although similar in appearance to the MLP diagram the outputs are calculated in a very different way. The first layer of weights $\boldsymbol{V}$ of the RBF represents the centres of the basis functions of their respective nodes; thus, instead of computing the dot product of the inputs and the first layer weights, as with the MLP, the values of $\boldsymbol{V}$ are used in calculating the activation of the hidden nodes:

$$\phi_i(\boldsymbol{x}) = \exp\left(-\frac{1}{2\sigma^2}\|\boldsymbol{v}_i - \boldsymbol{x}\|^2\right) \tag{3.27}$$

resulting in the outputs of each hidden node $\phi_i(\boldsymbol{x})$. We then calculate the output using:

$$o = \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{w} \tag{3.28}$$

therefore, only the weights in the vector $\boldsymbol{w}$ connected to nodes close to the input will have significant impact on the final output.

3.3.1 Training

In general the training of the RBF consists of two parts: firstly selecting the c and σ for each hidden node, and secondly training the output layer weights either in batch or in an on-line manner. When utilised for value function approximation for RL, the first phase is generally a case of selecting the quantity of BFs to use and then the c and σ are calculated to spread them evenly throughout the input-space, which are decided through experimentation on the given problem [10]. Thus, the training of the RBF is only the updating of the output weights, which, when combined with TD learning, is

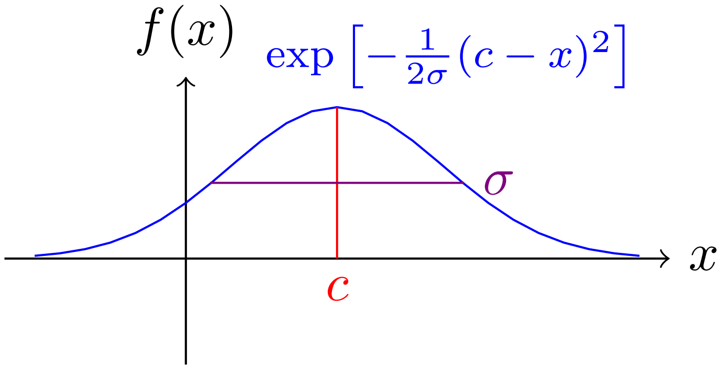


Figure 3.3: Diagram of the RBF activation function, for one-dimensional input. The Gaussian function is shown in blue; the centre variable c in red; and the width variable σ in purple.

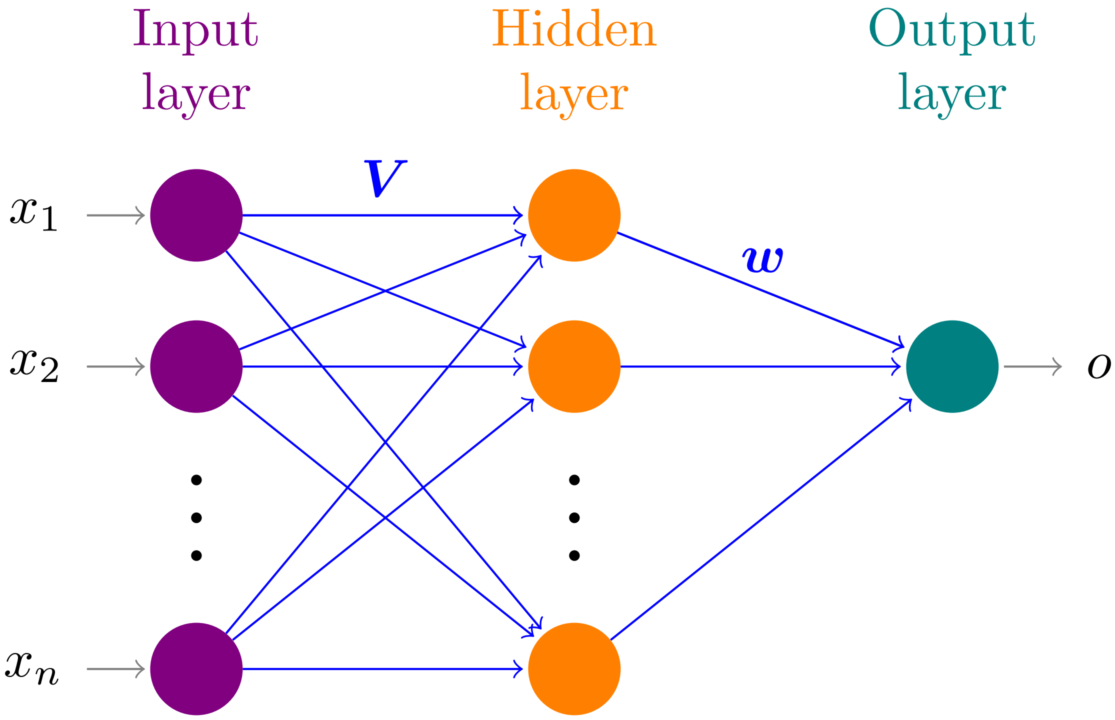


Figure 3.4: Diagram of the RBF neural network.

updated using gradient descent, in a similar manner to the training of the perceptron.

Thus training takes the form of minimising the error e by calculating $\nabla_{\boldsymbol{w}} e$, the elements of which are computed as follows:

$$\begin{aligned}\frac{\partial e}{\partial w_i} &= \frac{\partial e}{\partial o}\frac{\partial o}{\partial w_i} \\ &= \frac{\partial}{\partial o}\left[\frac{1}{2}(t-o)^2\right]\frac{\partial}{\partial w_i}\left[\sum_{j=1}^{n}\phi_j(\boldsymbol{x})w_j\right] \\ &= -(t-o)\phi_i(\boldsymbol{x})\end{aligned} \tag{3.29}$$

and updating the weights in the negative direction of this gradient:

$$\begin{aligned}\Delta w_i &= \eta\left(-\frac{\partial e}{\partial w_i}\right) \\ &= \eta(t-o)\phi_i(\boldsymbol{x})\end{aligned} \tag{3.30}$$

and, as with the other architectures, the weight update vector $\Delta\boldsymbol{w}$ is added to the current weight vector $\boldsymbol{w}$ to generate the weight vector for the next iteration:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w} \tag{3.31}$$

3.4 CMAC

The Cerebellar Model Articular Controller (CMAC) originally proposed by [1] uses a different approach from other ANNs such as MLP (Section 3.2) and RBF (Section 3.3) in that the non-linearity does not come from passing the inputs through non-linear activation functions, such as tanh. The active tile in each layer is calculated from the input vector (Fig. 3.5), and then a hash function is applied to the activated tile from each layer. This produces a set of active weight indices, one for each tiling layer used. Which is why the CMAC is often referred to as *tile-coding* among the RL community.

The designer selects the quantity of layers of tiles to be used, each of which is initialized with a random offset value in each dimension of the input space, which remain constant after initialization. For any given input the associated tile from each of the layers of tiles is activated. The layer and coordinates of the active tile in that layer are passed through a hash function to produce an active weight index. The summation of the active weights is calculated and is the CMAC's output value, training is performed by updating only the values of the active weights to move the output vector closer to the target.

It can be seen from Equation (3.32) that the output o of the network is obtained by computing the dot product on the weight vector $\boldsymbol{w}$ and the input vector $\boldsymbol{x}$ after it

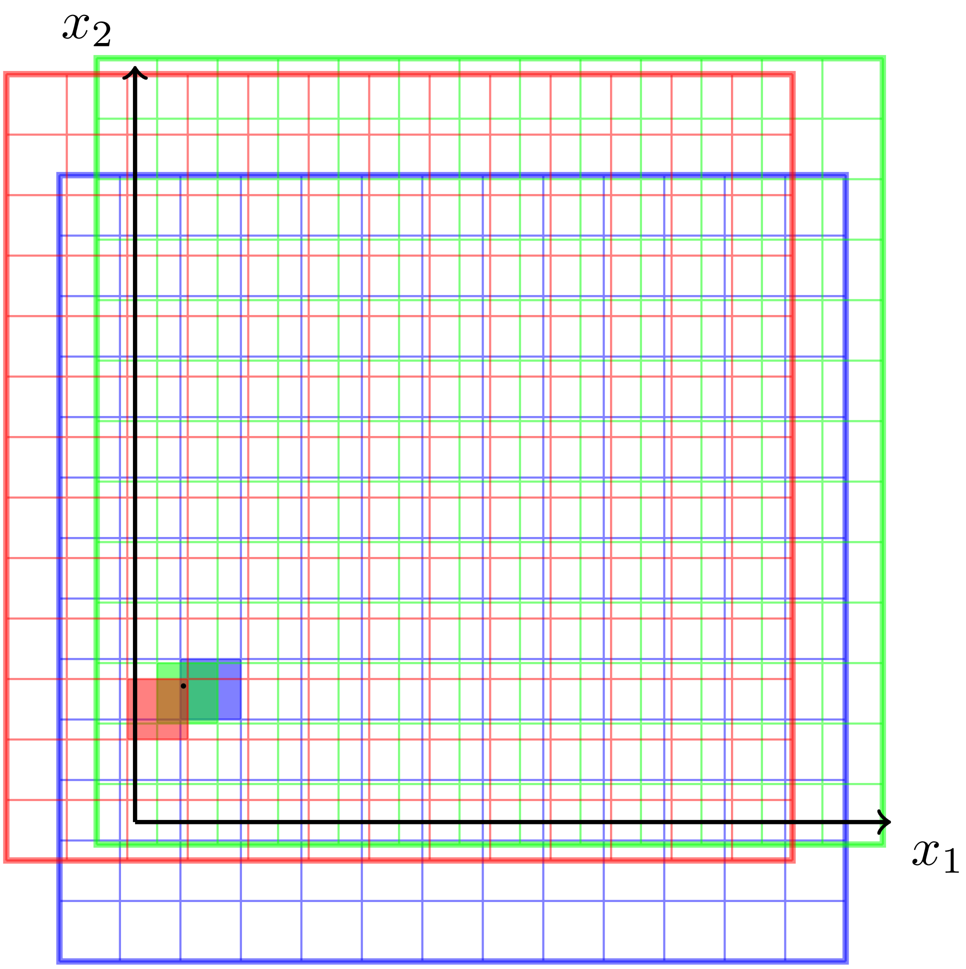


Figure 3.5: Diagram of the tiling of the CMAC with three layers of tiling shown in red, green and blue. The 2-dimensional input to the CMAC is shown as a black dot and the activated tiles of each layer are filled with their respective colour.

has been passed through a non-linear mapping function $\boldsymbol{\phi}(\cdot)$.

$$o = \boldsymbol{\phi}(\boldsymbol{x})\boldsymbol{w} \tag{3.32}$$

This is the same as Equation (3.28) for the RBF (Section 3.3), with the difference being solely in the function $\boldsymbol{\phi}$. In the CMAC approach the output of the function $\boldsymbol{\phi}$ is a binary vector consisting mostly of zeros. The activated indices have a value of one, the number of which is the same as the quantity of layers of tiles used. Thus, this method may have a large number of weights, but only a small number will be used in calculating the response to any particular input.

The number of active weights are equal to the number of layers tiling used, each layer produces a integral vector, or the grid coordinates of the active tile, which is then supplied to the hash function, along with the tiling number, to produce a positive index $\in [0, n-1]$, where n is the number of weights being used. Thus the quantity of tiling layers is not equal to the number of weights used, but is the number of active weights. If we store the output of the hash from each tiling layer in $\boldsymbol{a}$ we can calculate the output by applying:

$$o = \sum_{i=0}^{m} w_{a_i} \tag{3.33}$$

where there are m layers of tiling being used.

3.4.1 Training

Training can be performed similarly with the perceptron and the RBF, by updating the active weights in the direction of the gradient in order to reduce the error: e.g. the squared difference between the output and the desired output.

The main differences between the training of these architectures is that the output is simply the sum of the active weights of the CMAC (3.32), therefore the calculation of the elements of $\nabla_{\boldsymbol{w}} e$ is:

$$\begin{aligned} \frac{\partial e}{\partial w_i} &= \frac{\partial e}{\partial o}\frac{\partial o}{\partial w_i} \\ &= \frac{\partial}{\partial o}\left[\frac{1}{2}(t-o)^2\right]\frac{\partial}{\partial w_i}\left[\sum_{j=0}^{n}\phi_j(\boldsymbol{x})w_j\right] \\ &= \begin{cases} -(t-o), & \text{if } \phi_i(\boldsymbol{x}) = 1 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{3.34}$$

therefore only these m weights must be updated and all other elements of $\nabla_{\boldsymbol{w}} e$ will be zero.

The weight update values are:

$$\Delta w_i = \eta(t-o), \quad i = a_1, a_2, \ldots, a_m \tag{3.35}$$

and only those weights need to be updated:

$$w_i = w_i + \Delta w_i, \quad i = a_1, a_2, \ldots, a_m \tag{3.36}$$

making this a far less computationally intensive network to update than the others, particularly the RBF.

3.5 Chapter Summary

In this chapter we have presented a brief introduction to the most commonly applied ANN architectures applied to function approximation in RL, and how they can be trained iteratively using the gradient. It is also possible to train these neural networks in other ways, such as using batch algorithms, which is more suitable when working with a fixed dataset. With RL the data is received at each time-step in the form of the reward and thus iterative update methods are more suitable. Alternative methods of updating the weights will be discussed in Chapter 5, where we consider the policy gradient and GA approaches to direct policy search.

Each of the architectures presented in this chapter have advantages and disadvantages when applied to function approximation in RL, and we will consider these and compare the different ANNs below.

The Perceptron is not as widely used in approximating the value function in RL as the other approaches mentioned here due to the fact that it is only able to approximate linear functions of its input vector $\boldsymbol{x}$. One approach which can be taken to overcome this is to perform a non-linear mapping of the input before applying it to the ANN, e.g. input $\boldsymbol{x}$ is mapped to the ANN input $\tilde{\boldsymbol{x}} \leftarrow [x_0, x_1, x_0x_1, x_0^2, x_1^2]^\top$ which is then supplied as the ANN input vector; however, as the dimensionality of the input space grows, this method becomes difficult to apply unless there is some prior knowledge about the function which will be approximated, which is not normally the case in RL. An approach of this nature was taken by [35] where it was stated that achieving acceptable performance with a linear architecture requires tweaking.

The MLP is an obvious choice to overcome this problem, which is capable of approximating the widest range of functions of all the architectures presented here, as a global function approximator it is capable of approximating any function given sufficient hidden nodes [18, 25]. However, due to the fact that the MLP is more difficult to train, due to the weights becoming stuck in local minima, some members of the RL community prefer ANNs which are linear in the parameters such as RBF and CMAC. Despite this, RL utilising the MLP has been shown to achieve good results [48, 54, 56].

The RBF is a more widely applied approach as, the first layer performs a non-linear mapping and, once the BF centres are fixed, only the output weights are trained which leads to a method which is linear in the parameters and therefore is simpler to train than the MLP. However, as the input dimensionality grows the number of BFs grows exponentially, and as each of these must be evaluated to generate the output the RBF becomes computationally demanding and doesn't scale well. In order to apply RBF with larger input dimension it may be necessary to space the BFs further apart reducing the complexity of the functions which can be approximated.

The CMAC is similar to the RBF in that the tile coding and hash function performs a non-linear mapping to a set of active weights, the linear combination of which produces the output. However, the CMAC does not suffer from the scalability problems of the RBF, as only the number of tiling layers of weights are used for each input, which also results in a less computationally intensive approach. For these reasons the CMAC is widely applied in RL problems. However, the CMAC does perform some discretization at the tile coding stage and therefore is unable to achieve the same level of continuous smooth function approximation as other methods, e.g. RBF and MLP. It is shown that due to this the CMAC is unable to match the performance of the RBF on the Mountain Car benchmark problem [33].

Chapter 4

Evolutionary Algorithms

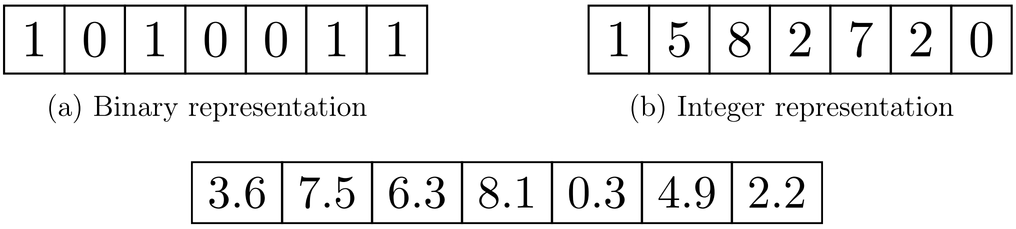
Evolutionary algorithms (EA) [17] is an approach to solving problems through the application of a process which is inspired by evolution and natural selection in nature. This is achieved by maintaining a population of individuals and at each generation some of these individuals are selected based on their fitness level as *parents* to be recombined, by the process of crossover, producing *children* which will be placed in the next generation.

Here we briefly introduce two of the main classes of evolutionary algorithms: genetic algorithms (GA) and genetic programming (GP), both of which are applied to solving RL problems through direct policy search. But they are very different in how they represent individuals and therefore how they can be applied to solving problems. Whilst GA represents individuals as vectors of numbers, and therefore the problem has to be modelled in such a way as to apply these vectors as solutions, GP produces programs, or mathematical functions, and therefore can be more directly applied to solving problems and the designer is not required to specify, so strictly, the form of the solution.

Firstly, we describe the details of genetic algorithms in Section 4.1; then we describe genetic programming in Section 4.2; and finally, in Section 4.3, we summarize the material presented in this chapter.

4.1 Genetic Algorithms

In genetic algorithms [17] the population of solutions at the first generation are initialised with random values within the limits of the problem and then, at each generation, individuals are selected based on their fitness and are with probability P_c recombined to produce children to place in the population at the next generation or with probability P_r are reproduced as exact copies in the next generation. There is also a probability of mutation P_m that a random change will be applied to individuals before they are added to the next generation. This process is repeated either for a



1	0	1	0	0	1	1

(a) Binary representation

1	5	8	2	7	2	0

(b) Integer representation

3.6	7.5	6.3	8.1	0.3	4.9	2.2

(c) Real number representation

Figure 4.1: Diagram of chromosome representations in genetic algorithms

specified number of generations or, if information about the desired performance is available, when we reach the optimal or desired solution.

Each of these processes is described in more detail in the following sections. Firstly we describe how the individuals are represented; followed by a description of the fitness function; the parent selection methods; then the details of how individuals are recombined through crossover and randomly changed through mutation; and finally implementation of elitism, which ensures the best individuals of the current population are included in the next generation.

4.1.1 Representation of Solutions

The population is comprised of multiple solutions to the problem, each of which is encoded as a chromosome. These chromosomes are represented as vectors of numbers, which may be either bit-strings; vectors of integers; or vectors of real numbers. However, in all cases they are a fixed length numeric representation of a solution to the problem to be solved by GA. Some examples of possible representations can be seen in Fig. 4.1.

Although the individuals are represented in this way, and therefore the problem must be modelled as such, this does not limit the applicability of GA to numerical optimization of a function taking a vector as the input, as non-numerical optimization problems can also be formulated in such a way as to be solved by GA.

4.1.2 Fitness Function

The fitness function is an critical element of constructing any EA solution as the fitness, along with the parent selection method, guide the evolution process, such that future generations include individuals with improved performance on the desired task.

The fitness function calculates a numeric value representing the performance of a particular individual on the problem to be solved as a function of the chromosome values.

4.1.3 Parent Selection

In order to guide the recombination of individuals through their fitness on the task the members of the population to be recombined, or reproduced, in the next generation are selected based on their fitness. This results in fitter individuals having a higher probability of either surviving to the next generation or reproducing children to populate the next generation. There are several methods of achieving this, here we will discuss two popular methods: *roulette wheel selection*, which is a fitness proportionate method, and *tournament selection*, in which a group of randomly selected individuals compete in a tournament for selection.

Roulette Wheel Selection

Roulette wheel selection, as the name suggests, uses a simulation of a roulette wheel to perform parent selection, where the wheel is divided into sections each of which represents an individual from the population. The size of these sections are proportional to the fitness of the corresponding individual, and therefore, when the wheel is spun the probability of the ball landing on any particular individual is proportional to the fitness of that individual.

Before the selection is performed the sum of the fitness of all individuals in the current population, of size M, is calculated: $sf = \sum_{i=1}^{M} f$. To 'spin the wheel' a random number rf is selected uniformly in the range $[0, sf]$ and the fitness of each member of the population is added to the cumulative fitness cf until $cf \geq rf$, and the last individual whose fitness was added is the selected individual.

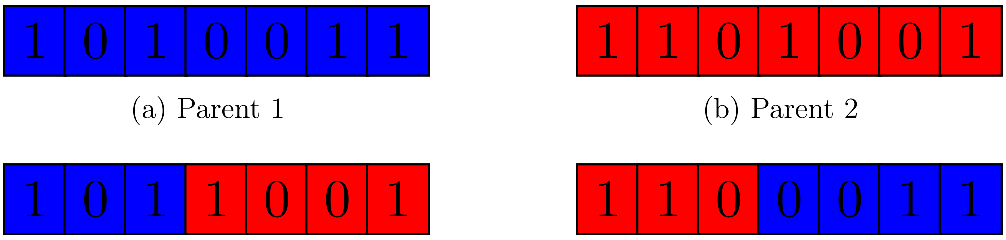
Tournament Selection

An alternative to the roulette wheel method is the tournament selection method, which does not require the fitness of all individuals in order to perform the selection. This is achieved by the uniform random selection of k individuals from the population to compete in the tournament. The winner is the individual in the tournament with the highest fitness.

4.1.4 Crossover

Crossover is applied to the two individuals which were selected, based on their fitness, to be combined to produce children, or to be directly reproduced, in order to populate the next generation of individuals.

There are several possible ways to apply crossover to the parents which may depend on the representation of the individuals. A simple example when bit-string encoding is employed is to select a random number α in the range $(1, L)$, where L is the length



(a) Parent 1

(b) Parent 2

(c) Child 1

(d) Child 2

Figure 4.2: Diagram of genetic algorithm one-point crossover, where the randomly selected crossover point is between the third and fourth nodes. Blue and red nodes represent genetic material from parent 1 and parent 2 respectively.

of the bit-string representation used, and to take the binary bits from the start until α from parent 1, and the bits from α to the end of the bit-string from parent 2, to create child 1 and the first section of parent 2 and the last section of parent 1 to create child 2, as is shown in Fig. 4.2. This method is one-point crossover, it is also possible to apply n-point crossover, for $n > 1$.

Alternative possible methods of crossover could be applied when real number encoding is used for chromosomes, one example is arithmetic recombination where the two parents, represented by vectors $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, are recombined to create the two children $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, by applying the following:

$$\begin{aligned} \boldsymbol{c}_1 &= \alpha\boldsymbol{p}_1 + (1-\alpha)\boldsymbol{p}_2 \\ \boldsymbol{c}_2 &= \alpha\boldsymbol{p}_2 + (1-\alpha)\boldsymbol{p}_1 \end{aligned} \tag{4.1}$$

where $\alpha \in (0,1)$ is a parameter controlling the weight given to each parent in the recombination, but in this method if $\alpha = 0.5$ the two children $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ will be identical. It is also possible to perform a similar arithmetic crossover at only certain indices of the individuals.

There are many other methods of crossover, obviously too many to discuss here; however, the above are given as some examples, further examples can be found in [17].

4.1.5 Mutation

Along with crossover, mutation is applied to alter individuals as they enter the next generation; however, mutation is applied to modifying one individual rather than combining two *parents*. As with crossover, the exact method of mutation will be somewhat dependant on the representation of the chromosomes. Mutation may be required in GA to restore areas of the search space which have been lost due to the parent selection method, or may not have been present in the initial random population. A simple example of this, when using a binary representation, is if the ideal solution is $[1,1,1,1,1]$, and the population contains $[1,1,0,0,1]$, $[1,1,1,0,1]$, $[1,0,1,0,1]$ and

$[0, 1, 1, 0, 1]$. Any recombination of the individuals in the population will not achieve the optimal solution, as no individual has a 1 in the second to last index. However, by allowing bits to be randomly flipped it may be possible to achieve the desired solution.

As mutation indiscriminately alters individuals at random, it may destroy good solutions rather than restoring necessary material that has been lost from the population. For this reason the probability of mutation P_m is generally very low.

The above example introduced a simple mutation operator applicable to bit-string chromosome representations: flipping a bit. In this method every bit of every individual will have probability P_m of being flipped.

When an integer or real valued representation is used, it is not possible to simply flip the value as there will be more than two possible values. In this case it is possible to randomly reinitialise the value at the specified index of the chromosome with a uniformly selected value.

Alternative mutation operators are also possible such as permutations of the individual or swapping the values of two indices in a given chromosome.

4.1.6 Elitism

When randomly performing crossover and mutation on the population there is a possibility that good individuals will not survive to the next generation, either because they were not selected as parents; crossover was performed and the feature of the solution which provided its high fitness was lost; or mutation was applied to the individual. To ensure the survival of the fittest individuals in the population elitism may be applied, which transfers exact copies of the individual(s) with the highest fitness to the next generation.

4.2 Genetic Programming

Genetic programming [32, 44] also takes its inspiration from genetics and evolution. Unlike GA, however, solutions in GP comprise the structure as well as the parameter values. The solutions can be thought of as a program, or as a function, an illustration is given in Fig. 4.3 on page 37, and also in Fig. 4.4 on page 40 which illustrates how individuals are combined during the genetic recombination process to produce the next generation of solutions.

The initial population of trees are randomly generated, and then the population at each subsequent generation is generated by with probability P_c performing crossover on two individuals from the current population; with probability P_r selecting two individuals to reproduce exactly from the current population; and with probability P_m applying mutation to an individual from the current population.

In this section we introduce the structure of GP individuals; initialization methods; crossover and mutation operations; and the problem of 'bloat' along with some approaches to overcome it.

4.2.1 Representation of Individuals

In GP individuals in the population are commonly represented as a tree of *function nodes* and *terminal nodes*. Function nodes have branches to a number of nodes, corresponding to the quantity of arguments of the particular function. These nodes may also be function nodes, or terminal nodes. Terminal nodes, as the name suggests are terminals and have no branches.

In order to apply GP to a problem we must specify the set of possible function nodes $\mathcal{FS}$ and the terminal set $\mathcal{TS}$, when doing this, care must be taken to include all required functions and terminals, but to limit unnecessary function and terminals as they will expand the solution search space and increase the difficulty of evolving optimal solutions.

Function Nodes

The function set may include mathematical functions such as:

$$\{+, -, \times, \div, \surd, \sin, \tanh, |\cdot|\}$$

where $|\cdot|$ is the absolute value. However, care must be taken when applying functions such as $\div$ which are undefined for certain values, i.e. when the second argument is zero, in such cases it is common to apply a 'protected' form of the function, such as the protected divide function [32], ensuring the function is defined for all possible argument values. The protected divide used here is:

$$\begin{cases} arg_1 \div arg_2, & \text{if } arg_2 \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

It is also possible to apply functions such as if-less-than to provide a branching structure in the solutions:

$$\begin{cases} arg_3, & \text{if} \quad arg_1 < arg_2 \\ arg_4, & \text{otherwise} \end{cases} \tag{4.3}$$

From these examples it is clear that functions may take various numbers of arguments, each of which could either be another function node, or a terminal node.

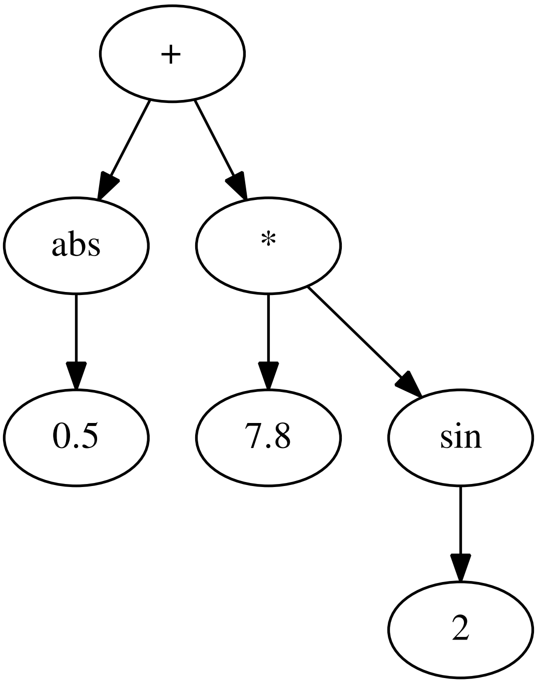


Figure 4.3: An example of a tree representation of an individual used in GP

Terminal Nodes

Terminal nodes are numerical values and could be numeric constants, e.g.:

$$\{\pi, e, \varphi\}$$

which may be included in the terminal set if they are considered to be helpful in solving the problem. Alternative terminals include the current values of the state of the problem, e.g. the current angle of a pole which is to be balanced, or a randomly generated value $R \in [R_{min}, R_{max}]$, which is generated at initialization and remains constant thereafter. This makes potentially useful values available without requiring the inclusion of them in the terminal set at design time

4.2.2 Initialization Methods

When initializing the population in GP, trees may be built by selecting from the set of function nodes, which obviously results in further layer(s) following the current node, or selecting from the set of terminal nodes, which will limit the depth of the tree to that of the current node. Therefore, some rules may be applied to determine the depth and shape of the initial population.

Here we describe three standard initialization methods applied in GP: *full*, *grow* and *ramped half-and-half*. For all of these methods a parameter d_i is set to restrict the maximum initial depth of the trees.

Full

When the 'full' method is applied the maximum depth d_i is set and all nodes at depth less than d_i are selected only from the set of function nodes $\mathcal{FS}$, and nodes at depth equal to d_i are only selected from the set of terminal nodes $\mathcal{TS}$. This results in all trees being initialised to the full size permitted by the d_i parameter.

Grow

With the 'grow' initialization method we do not restrict the nodes at depth less than d_i to $\mathcal{FS}$, instead permitting any nodes from $\mathcal{FS} \cup \mathcal{TS}$, and therefore allow trees, or sections of trees, to have a depth less than d_i. However, nodes at depth d_i are still restricted to $\mathcal{TS}$ to limit the maximum initial depth.

Ramped Half-and-Half

The 'ramped half-and-half' method combines both of the previously described methods: 'full' and 'grow', and also varies the maximum depths of the trees created. This is achieved by generating half of the initial population using the grow method and half by the full method. Also, to vary the depths of the trees in the initial population, an equal proportion of the trees are initialised with a maximum depth of $2, 3, \ldots, d_i$. Thus, the GP process begins with an initial population varying in both depth and shape.

4.2.3 Parent Selection Methods

As with GA we must select individuals from the current population to apply crossover and reproduction in order to populate the next generation.

As each individual has a fitness value we can apply the same fitness proportionate or tournament based methods described for GA in Section 4.1.3. There is also another commonly applied parent selection technique applied to GP: *fitness proportionate over-selection*, which is described below.

Fitness Proportionate over-selection

Fitness proportionate over-selection [32] is a method to place a higher probability of selecting the fittest individuals in the population. This is implemented by first calculating the normalised fitness of all the individuals in the current population and then sorting the population based on their normalised fitness. In order to calculate the normalised fitness we first calculate the adjusted fitness, which exaggerates the difference between the best performing individuals:

$$f_a(i) = \frac{1}{1 + f(i)} \tag{4.4}$$

where, $f(i)$ is the fitness of the i^{th} individual of the population. Then we calculate the normalised fitness:

$$f_n(i) = \frac{f_a(i)}{\sum_{j=1}^{M} f_a(j)} \tag{4.5}$$

where M is the number of individuals in the population.

Once the normalised fitness has been calculated and the population has been sorted it is divided into two groups: one containing the individuals of highest fitness and the other containing the rest of the population, over-selection can then be performed.

A cumulative percentage parameter c controls the percentage of the total fitness of the entire population which is included in the fittest group. The fittest individuals are added to this group until the cumulative fitness of them reaches c, the remaining individuals in the population are placed in the other group.

Individuals are selected for crossover 80% of the time from the group of fittest individuals and 20% of the time from the other group. When over selection is performed, the individuals are still selected proportionately to their normalised fitness from appropriate subset of the population.

4.2.4 Crossover

When crossover is to be performed two individuals are selected from the population, using one of the parent selection methods discussed above, and they are combined to create two new individuals for the next generation. As the individuals are selected based on their fitness it is hoped that the combinations of the individuals will produce individuals with higher fitness.

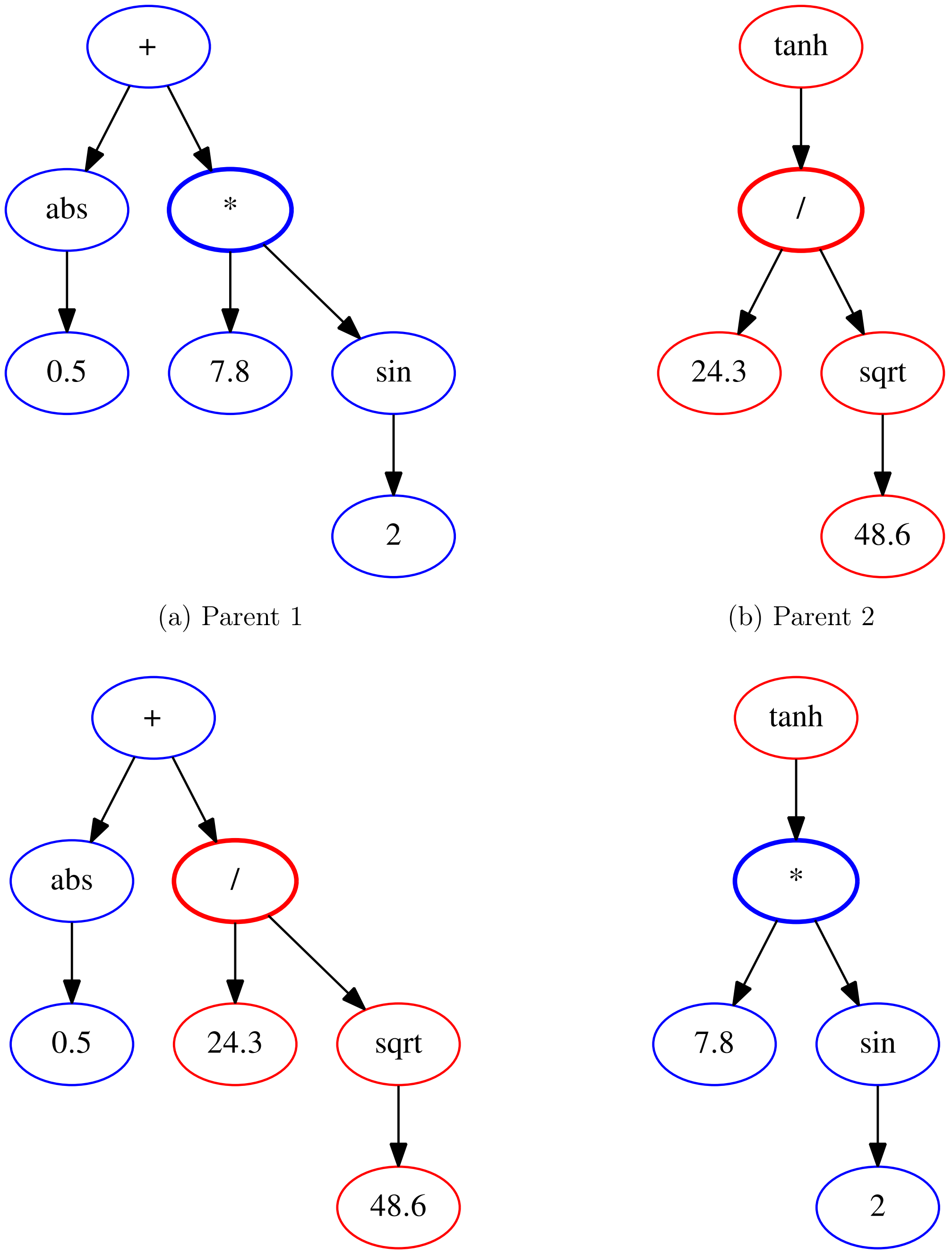
As GP individuals can be viewed as tree structures, the crossover process first selects a random node in the tree of each parent. A node at an *internal point* is selected with probability P_{ip}, which should be high to avoid crossover merely swapping terminal nodes in the two trees [32].

Following the selection of two crossover points, the sub-trees having these nodes at their roots are 'cut' from the parents and swapped to generate the two children, as depicted in Fig. 4.4 on the next page.

4.2.5 Mutation

Mutation in GP allows the random re-initialization of a node or sub-tree to allow the re-introduction of required functions or terminals which are not currently present in the population.

The simplest method of applying mutation in GP is to select a random node in the tree to be mutated and replace the node with a randomly generated sub-tree which, as



(a) Parent 1

(b) Parent 2

(c) Child 1

(d) Child 2

Figure 4.4: Diagram of crossover in genetic programming, the bold nodes show the randomly selected crossover points, and blue and red nodes identify nodes which are taken from parent 1 and parent 2 respectively.

with the initially generated trees, has a maximum depth. However, many alternative mutation operations exist [44].

As both the function and terminal sets tend to be relatively small and repeated many times in the population it is unlikely that any nodes will be missing from the population, and require mutation to be re-introduced into the population [32]. Thus, mutation is not considered as essential in GP as it is in GA and is not always employed [32]. However, there is also evidence that mutation may be beneficial, depending on the problem, and therefore should be tested, with a low probability, when tuning the parameters of the GP model [44].

4.2.6 Bloat Control

When performing crossover, a deep sub-tree may be placed in another tree. Such crossover operations can, within a few generations, lead to massive growth in the depths of the individuals in the population. This can be problematic for two reasons: firstly, the larger trees may increase the resources required; and secondly, more complicated trees may have reduced generalization capabilities. This undesirable growth which does not significantly increase the fitness is known as bloat in the GP community [37, 44].

There are several approaches to controlling bloat in GP, the most simple being *depth limiting*, which places a limit on the maximum permitted depth of trees from genetic operations. Many other approaches, however, exist [37], e.g. adding a *penalty term* to the fitness function which is known as *parsimony pressure*. Both of these approaches are described below.

Depth Limiting

Depth limiting, as used in [32], is a widely applied approach to bloat control, in which a maximum depth parameter D_c controls the maximum depth of individuals due to genetic operations. This is different to the initial depth parameter D_i, and is a larger value to allow some growth of the individuals during the genetic process.

After crossover is applied the depth of the two children are calculated and if one of them exceeds the maximum depth D_c it is replaced by a direct copy of one of the parents. If both of them exceed D_c then both children are replaced by the parents.

Parsimony Pressure

An alternative, slightly more sophisticated, bloat control method is the parsimony pressure method [37]. This method can be applied in many forms but as an example in the linear version individuals are selected proportionally to an augmented fitness function g, which is a linear combination of the fitness f and the depth d of the given solution:

$$g = xf + yd \tag{4.6}$$

This method adds an additional parameter to tune, which specifies the importance of the size of solutions in relation to the fitness function. This parameter is difficult to set and best results may be found by combining this technique with depth limiting [37].

4.2.7 Elitism

The genetic operations applied to the individuals in GP are guided by the fitness, and therefore, will create individuals with higher fitness on the task GP is being applied to. However, there is a chance that the fittest individuals may not be reproduced to the next population and even if they are selected for crossover, the resulting individuals may have much lower fitness.

In order to overcome this problem *elitism* may be applied, which, as with GA, ensures the fittest individual(s) from the current generation are copied to the next generation.

4.3 Chapter Summary

This chapter was a brief introduction to evolutionary algorithms in the forms of GA and GP, both of which are applied to RL problems later in the thesis. There are many variations of these techniques including alternative crossover and mutation operations; however, this limited presentation describes the fundamentals of the techniques.

GA is often applied as a global optimisation method and has been applied to finding the weights of artificial neural networks, such as those introduced in Chapter 3, approach is applied to the solution of an RL problem, such methods can be seen as an example of direct policy search (described in Section 5.2).

GP is also a well known approach to direct policy search [30, 53] and is described in Section 5.2.3 and an example of its application to the Acrobot swing-up problem is presented in Section 7.1.

Chapter 5

Continuous State- and Action-Space State-of-the-art

Here we elaborate on the problems of continuous state- and action-spaces, as was described briefly in Chapter 2, and present the current state-of-the-art methods applied to solving such problems.

When problems have discrete, small state- and action-spaces the value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, giving the expected value of taking a given action from a given state can be stored in a look-up table; however, if $|\mathcal{S}|$ is too large or, in the continuous state-space setting, infinite this is no longer feasible, and leads to two problems:

1. it would be impossible to store all values in a look-up table
2. the agent would be unable to visit all states and actions sufficiently often to find the correct values

When this is the case, but the action-space is discrete and sufficiently small, we can solve both of these problems by utilising function approximation, such as artificial neural networks (Chapter 3), to approximate the state-action value function $Q(s, a)$. Due to the generalisation capabilities of such function approximation methods, we are able to make estimates of values of unvisited states based on the current learnt values of other states. Also the updating of these values will improve the approximation of similar states and actions. Furthermore, as the function is approximated using a small number of parameters it is possible to store in memory. The optimal action $\arg\max_a Q(s, a)$ can then be selected by evaluating:

$$Q(s, a), \quad \forall a \in \mathcal{A} \tag{5.1}$$

which is often the favoured approach when it is practical to do so [21].

However, when the action-space $\mathcal{A}$ is large, or continuous, this approach is no longer directly applicable, as it would be impossible to evaluate so many possible actions at every time-step. There are, however, several approaches which either facilitate the application of this method through discretization or avoid the problem by storing an approximation of the policy function, either alongside the value function; or by directly optimizing the policy function without approximating the value function. The examples of the state-of-the-art approaches we discuss here can be classified as follows:

- actor-critic:
    - adaptive critic [48]
    - CACLA [56]
- direct policy search:
    - genetic algorithm based direct policy search [16]
    - genetic programming based direct policy [12]
    - policy gradient [46]
- implicit policy methods:
    - action-space discretization [47]
    - gradient based action selection [30, 47]
- other methods:
    - k-nearest neighbours [23]
    - wire fitting [2]

In the remainder of this chapter the above approaches will be presented individually, along with the advantages and disadvantages of each, followed by a summary at the end of the chapter.

5.1 Actor-Critic

Actor-Critic (AC) methods [21, 55] require two function approximators: the critic stores the value function (either $Q(s, a)$ or $V(s)$ may be used) whilst the actor stores the policy function $\pi : \mathcal{S} \to \mathcal{A}$, i.e. which action to take given the current state. This leads to very fast selection of actions and allows the full range of continuous actions to be selected. However, it also leads to many more parameters which must be tuned, in the form of the function approximator parameters, and two function approximators which must be trained.

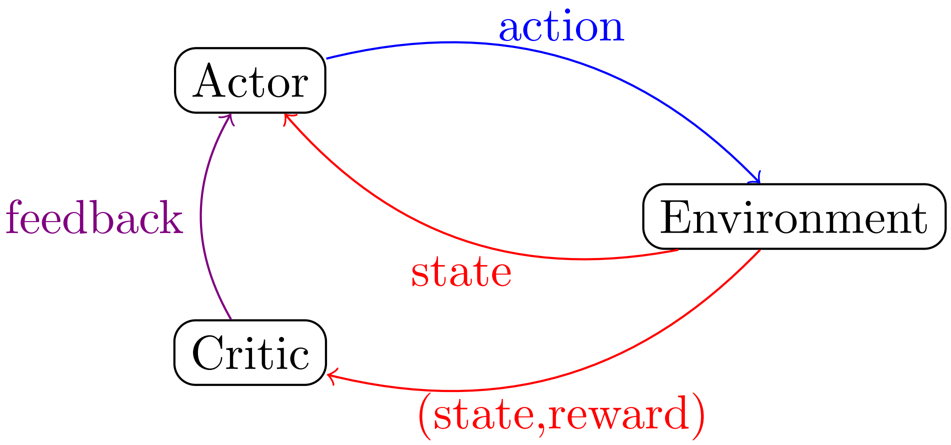


Figure 5.1: Diagram of the Actor-Critic architecture, showing how the actor and critic interact with the environment and each other.

At each time step the actor selects an action and applies it to the environment, after which it receives the next state s'. At the same time the critic receives s' and also r which are used to update the critics estimate of the value function and also to send a feedback value to the actor from which the actor updates the policy function. A graphical representation of this interaction between actor, critic and environment is shown in Fig. 5.1.

Two specific examples of AC implementations which have been applied to continuous state- and action-space problems are the adaptive critic [48] and CACLA [55], each of which is described in more detail below.

5.1.1 Adaptive Critic

The adaptive critic is another name for the actor-critic architecture, here we will use this name to describe the implementation of the AC architecture used in [48].

This implementation utilises two MLPs for the actor and critic. The actor approximates $\pi : \mathcal{S} \to \mathcal{A}$, whilst the critic approximates $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This approach also includes a count of the number of consecutive trials within the current episode which ended within a given number of time-steps of each other, which allows the weights of the MLPs to be reset if training becomes *stuck*.

The critic network $Q(s, a)$ is used to approximate the discounted reward:

$$R(t) = r_{t+1} + \alpha r_{t+2} + \dots \tag{5.2}$$

where α is the discount parameter.

The error of the critic network is calculated as:

$$e_c(t) = \alpha Q(s_t, a_t) - [Q(s_{t-1}, a_{t-1}) - r_t] \tag{5.3}$$

and the weights of the critic network are updated using gradient descent to minimize the square of this error.

The error of the actor network is defined as the difference between the output of the Q function and the desired value of Q. This is possible as the reward is defined to be 0 in the success and is -1 only on failure, thus, the desired value is zero, and the error is:

$$e_a = Q(s_t, a_t) - Q_{desired} \tag{5.4}$$

where gradient descent is applied to updating the policy function to minimize the square of this error.

There are separate learning rates for the action and critic networks which are reduced as time progresses. This implementation also performs several updates on each network with parameters to control the maximum number of updates performed at any time-step N_a and N_c, and also error thresholds for each network to determine if fewer updates is already sufficient T_a and T_c. The weights of the critic were also confined to a small range by, after each set of updates of the critic network, applying[1]:

$$\boldsymbol{\theta}_c \leftarrow \frac{\boldsymbol{\theta}_c}{\|\boldsymbol{\theta}_c\|_{max}} \tag{5.5}$$

This method requires several parameters not used in other applications, such as CACLA, to allow multiple updates and resetting the weights of the neural networks. Moreover, to apply the update to π there must be a desired Q value whilst this is often the case in disaster avoidance tasks, where the reward is zero at all times except failure, at which point it becomes a negative value, this is not necessarily the case for all RL tasks.

5.1.2 CACLA

Another approach is that used in the continuous actor-critic learning-automaton (CACLA) [56], which comprises an actor network and a critic network, which were both implemented using MLPs with 12 hidden nodes in the experiments where CACLA was proposed [56]. CACLA updates the policy by observing the temporal-difference error and if it is positive the action taken is better than expected, so the actor is updated to increase the probability of selecting the action taken. As the policy is only updated when exploratory actions are taken exploration must be applied, using e.g. the Gaussian exploration method [55].

The main unique point of the CACLA algorithm is that only the sign of the TD

[1] In [48] it is stated that $\|\boldsymbol{\theta}_c\|_1$ is used and a corresponding normalisation is applied to the actor weights; however, this description specifies the details of the implementation which was supplied by the authors.

error is used in calculating the actor update, whereas other AC algorithms also use the magnitude. Also, most other AC methods update the policy when the TD error is negative, but this is not the case with the CACLA algorithm based on the intuition that we should not increase the probability of actions which were not taken as no information about the value of such actions is available [55, 56].

The updates to the MLPs are performed as follows, at each time-step the critic network is updated using:

$$\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \alpha\delta\nabla_{\boldsymbol{\theta}_v} V(s) \tag{5.6}$$

where $\boldsymbol{\theta}_v$ is the vector of parameters of the value function $V(s)$; α is a learning rate parameter; and δ is the TD error. Then if $\delta > 0$ the actor network is updated using:

$$\boldsymbol{\theta}_a \leftarrow \boldsymbol{\theta}_a + \alpha(a - \pi(s))\nabla_{\boldsymbol{\theta}_a}\pi(s) \tag{5.7}$$

where $\boldsymbol{\theta}_a$ is the vector of parameters of the policy function $\pi(s)$ and, as with the critic update, α is the learning rate parameter.

In order to speed up the training of the policy function the number of updates applied to the actor network is proportional to the ratio of the TD error and the variance of the TD error. The variance of the TD error is initialized by a parameter at the start of training and is then updated at each time-step using:

$$\text{var} \leftarrow (1 - \beta)\text{var} + \beta\delta^2 \tag{5.8}$$

and the number of updates performed on the policy network is $\left\lceil \frac{\delta}{\sqrt{\text{var}}} \right\rceil$.

Both of these AC methods, and AC methods in general suffer from the fact that two function approximators are required, and the policy function is trained based on the current estimates of the value function. This results in more training and more parameters; however, the explicit policy function does allow very fast action selection.

5.2 Direct Policy Search

Direct policy search (DPS) methods store only the policy function $\pi(s)$, and therefore, as with AC methods, select actions quickly once trained. The policy function is either updated using approximations of the gradient [46]; or using derivative-free optimization techniques [15, 12]; however, regardless of the method used, a very large number of episodes are required to find the values of the objective function in this optimisation, which is generally the sum of rewards. DPS methods do not take advantage of rewards received at each state transition as is utilised by temporal difference methods.

All DPS algorithms seek to maximize the expected sum of rewards received when following the policy π[1]:

$$\mathbb{E}\left\{\sum_{t=0}^{T} r_t \middle| \pi\right\} \tag{5.9}$$

without making any use of intermediate rewards or the estimated values of states, as is used by DP and TD methods.

The various approaches to DPS are described separately in this section: firstly, policy gradient, which seeks to optimise the parameters of the policy function by estimating the gradient; followed by genetic algorithms which is an evolutionary, derivative-free approach to the same problem; and finally genetic programming, which attempts to optimises the structure of the policy function rather than merely finding the parameters which optimise a fixed structure policy function.

5.2.1 Policy Gradient

Policy gradient (PG) methods [46] consist of an approximation of the policy function which is improved through iterative updates based on approximations of the gradient.

As the PG method searches for good policies and does not attempt to learn the value function, sometimes it is referred to as an actor only method [10]. One of the advantages of PG is that learning a good approximation of the value function may be more difficult than finding a good policy directly [4]. PG is also applicable to continuous action problems which are problematic for many other methods; however, PG methods only search for a local maximum of the performance measure of the policy $J(\boldsymbol{\theta})$.

The policy is parameterised on a vector $\boldsymbol{\theta}$, and therefore is denoted as $\pi_{\boldsymbol{\theta}}$ and the performance of the policy is the expected sum of rewards:

$$J(\boldsymbol{\theta}) = \mathbb{E}\left\{\sum_{t=0}^{T} r_t \middle| \pi_{\boldsymbol{\theta}}\right\} \tag{5.10}$$

and this parameter vector is updated in the direction of the gradient using the update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{5.11}$$

where:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2}, \ldots, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n}\right]^{\top} \tag{5.12}$$

There are three widely applied approaches to approximating $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$: the method of finite differences; vanilla gradient; and natural gradient, as described below.

[1] The expected average reward may also be used by dividing this value by the number of time steps T, but if the episodes are of the same length this will be dividing by a constant and therefore will not alter the ranking of the different policies.

Finite Differences

One of the most simple methods of estimating the gradient of $J(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$ is the method of finite differences. The parameter vector is perturbed in each dimension to approximate the partial derivative of $J(\boldsymbol{\theta})$ w.r.t. the given dimension of $\boldsymbol{\theta}$:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_k} \approx \frac{J(\boldsymbol{\theta} + \varepsilon \boldsymbol{U}_k) - J(\boldsymbol{\theta})}{\varepsilon} \tag{5.13}$$

where $\boldsymbol{U}_k$ is a vector with 1 at the k^{th} position and 0 at all other positions. In this approach the policy used is deterministic.

Vanilla Policy Gradient

An alternative method of estimating the gradient is the 'vanilla' policy gradient method which is different from the method of finite differences in that it relies on the use of a stochastic policy. The estimate of the gradient is also calculated from sample trajectories.

One of the problems noted with this method is the high variance of gradient estimates, thus a baseline value is utilised to reduce this [46].

Natural Gradient

An extension to the vanilla policy gradient method is to follow the natural gradient rather than the standard gradient [31], which can overcome the problems associated with the standard gradient becoming trapped in plateaus [6, 31, 43, 46].

This method is also called natural actor-critic when the training process uses the value function $V(s)$ as the baseline, to minimize the variance [6].

5.2.2 Genetic Algorithms

Genetic Algorithms (GA) (as described in Section 4.1) is a successful derivative-free global optimization method, and therefore is a natural alternative to the derivative based methods used in the policy gradient approach.

GA is applied to directly searching for the parameters of a policy function which maximise the sum of rewards, i.e. to find the solution to:

$$\arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{5.14}$$

and thus produce a policy which performs well on an RL task. The policy is often implemented using an ANN, where $\boldsymbol{\theta}$ represents the weights of the ANN which approximates $\pi(s)$ [16] and the GA fitness function is simply $J(\boldsymbol{\theta})$.

This method can be very time consuming as it is common to have many competing solutions in the GA population, all of which must be evaluated for their fitness by running a simulated episode of the control problem, possibly multiple times with different starting states, at every generation[1]. This may be mitigated, to some extent, by computing the fitness of several individuals from a given generation in parallel, as the individuals in each generation are not affected by others in the same generation; however, this is only possible in simulations. Also, due to the nature of the optimization method, whilst some individuals will improve, many individuals will have terrible performance. Thus, even without the computational requirements precluding the direct applicability rather than simulations, the poor individuals may cause damage to hardware if applied to a real control problem.

5.2.3 Genetic Programming

Genetic Programming (GP) has been noted as an approach to RL problems [30, 53] and is commonly applied to finding optimal control for problems where the sum of rewards received at each time-step throughout the episode is applied as the fitness function, albeit not often presented from an RL perspective [12, 13, 14, 32].

Whereas PG and GA methods search for the parameters of the policy function $\boldsymbol{\theta}$ which maximize $J(\boldsymbol{\theta})$, GP searches for the structure of the policy function by combining a given function set and terminal set, which are problem specific and specified by the designer. Thus, GP can be thought of as a technique to search a subset[2] of the space of all policy functions for the π which maximizes the average expected reward:

$$\mathbb{E}\left\{\sum_{t=0}^{T} r_t \middle| \pi\right\} \tag{5.15}$$

rather than seeking to maximise (5.10).

The fact that the structure is not predefined makes GP more generic in the functions it can produce, but also places many other implementation decisions on the designer, most importantly the function and terminal sets; however, there are many more decisions such as bloat control methods, initialisation method, populations size, number of generations, etc. (see Section 4.2). If the function set is too large good solutions may be missed due to the prohibitively large search space, on the other hand if required functions are omitted from the function set the desired solution will not be possible. The other parameters must also be tuned for the specific problem.

[1] It is not necessary to recompute the fitness of any individuals which are reproduced in the following generation without mutation; however, few individuals are reproduced exactly from one generation to the next in GA.

[2] Limited by the function and terminal sets selected at design time.

Moreover, each run of GP is very time consuming due to the large population size and number of generations required, and several runs must be attempted using different randomly initialized populations due to the stochastic nature of GP; thus, tuning the parameters for GP may be a very time consuming process.

5.3 Implicit Policy Methods

Implicit policy methods (IPM) store only the value function $Q(s, a)$ and by evaluating this for possible actions are able to select the action which maximizes this function for the given state. This approach, therefore, does not require the learning of the policy function, and is sometimes referred to as a critic only method.

The two main approaches to applying IPM to continuous action-space problems are discretization of the action-space and applying gradient-based optimization to the value function. When discretization of the action-space is applied the optimal action can be selected by evaluating $Q(s, a)$ for each of the discrete actions, and the action with the highest value is selected, and thus, limits available actions to the discretized action set. When gradient-based optimization is applied we retain the ability to select any action from the continuous action-space at the expense of solving a numerical optimization problem each time the selection of the greedy action is required.

5.3.1 Discretization

The most straight-forward approach to discretizing the action-space is the one-step-search [47], where an interval is selected a_Δ and only the subset of the action-space:

$$\tilde{\mathcal{A}} = \{a_{min}, a_{min} + a_\Delta, \ldots, a_{max} - a_\Delta, a_{max}\} \tag{5.16}$$

is evaluated. As this transforms the continuous action-space problem into a discrete action-space problem it allows the use of implicit policy methods. However, this approach suffers from two large problems: 1) the level of granularity must be specified at design-time, which is difficult; 2) if the level of granularity required to achieve suitable performance on the task is too fine, even $|\tilde{\mathcal{A}}|$ will be too large to evaluate $Q(s, a), \quad \forall a \in \tilde{\mathcal{A}}$.

Both of these problems have been addressed, to some extent, by [42] where the action-space is discretized with a very fine granularity but the action selection only requires the evaluation of two actions $a - a_\Delta$ and $a + a_\Delta$, where a_Δ is also updated to increase or decrease the granularity at each time-step. This may prevent fast changes in action value and also precludes the agent from applying the same action at two consecutive time-steps, due to the minimum value of a_Δ.

5.3.2 Gradient Based

Applying the gradient of $Q(s, a)$ w.r.t. a has been suggested as a method to select the action which maximizes the value function [30, 47]; however, this approach is often dismissed as impractical [21, 55].

The greedy action is selected by initializing a to an estimate and iteratively updating the value of the selected action using the gradient ascent method of optimization:

$$a \leftarrow a + \alpha \nabla_a Q(s, a) \tag{5.17}$$

where α is a step-size parameter.

This method is only applicable if the gradient of the value function is available, and also suffers from the problems of slow convergence rate and a susceptibility to becoming trapped in local optima [41].

5.4 Other Methods

The following two methods do not fit into the other classes of solutions, they are wire-fitting and k-nearest neighbours. Wire-fitting performs function approximation whereby the learnt function outputs several actions and their corresponding values, and due to the nature of the function approximation method used the greedy action will always be included in the set of action-value pairs produced. k-nearest neighbours performs discretization of the state-space and the action-space, resulting in a problem that can be stored in a lookup table as is often applied in the small, discrete state- and action-space setting; however, the selection of continuous actions is made possible by taking a weighted average of the k actions selected from the discretized set.

5.4.1 Wire-Fitting

Wire fitting (WF) is a method of approximating $Q(s, a)$ using n wires, resulting in a value function like a sheet draped over the wires which can be evaluated through interpolation [2].

The algorithm consists of a function approximator which outputs n actions, corresponding to the locations of the n wires, and the associated n state-action values of these actions and the given state, i.e. $Q(s, a_i), \quad i = 1, 2, \ldots, n$.

Interpolation, $f(s, a)$ in Fig. 5.2, can be utilised to find the values of actions between those output by the function approximator; however, this is not required for action selection as the action with the highest value will always one of those given by the function approximator as one of the wires will always be located at the maximum point.

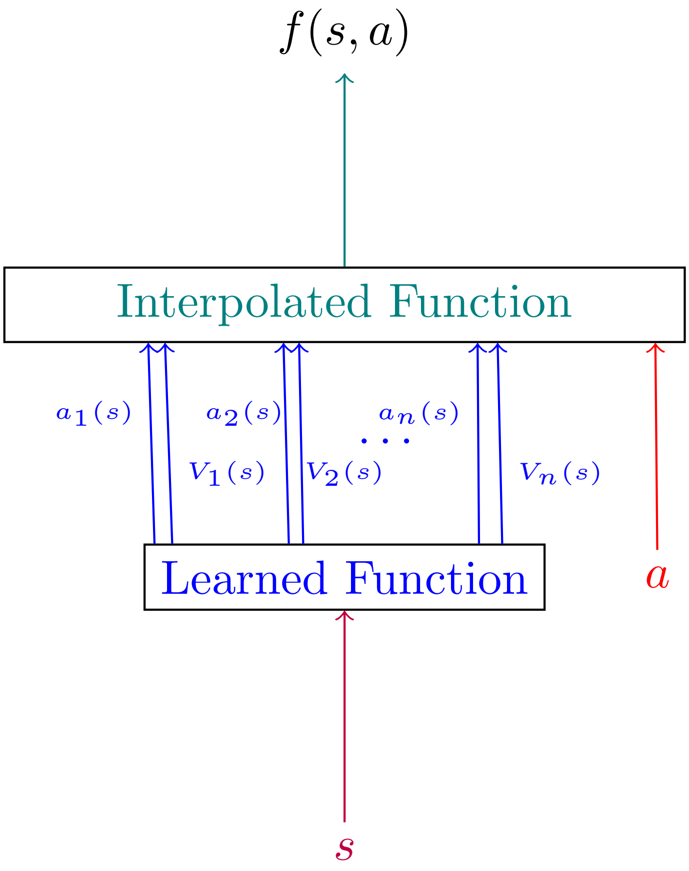


Figure 5.2: Diagram of the wire-fitting architecture.

This method has been shown empirically to be unable to succeed in the Cart-Pole problem when the track is limited [2] and has since been shown to perform worse than other methods [56] on a continuous action control problem. Also, for each of the n wires the learned function must output an action and a value, therefore if a large number of wires is required for the problem this method may become computationally demanding, as is suggested in [56] where it is stated as being the most computationally demanding of the methods applied.

5.4.2 k-NN

The k-nearest neighbour (k-NN) method [23], similarly with the one-step-search, uses a discretized action-space, but the k-NN method is also able to select actions which are not included in the discretized action set. This is achieved by discretizing both the state- and action-spaces, and then taking the weighted average of the k discretized actions that would have been selected from the k discretized states with the smallest euclidean distance from s.

The discretized action space is achieved by taking n uniform discrete actions from the continuous-space, this can be done by setting:

$$a_\Delta = \frac{a_{max} - a_{min}}{n-1} \tag{5.18}$$

and the discretized action-space:

$$\tilde{\mathcal{A}} = \{a_{min} + i \cdot a_\Delta\}_{i=0}^{n-1} \tag{5.19}$$

The weighting of each action $a_i \in \boldsymbol{a}$ is calculated by first calculating a closeness value based on the euclidean distance between each of the k nearest neighbour states $knn_i, \quad i = 1, 2, \ldots, k$ and the state to be evaluated s:

$$w_i = \frac{1}{1 + \|s - knn_i\|} \tag{5.20}$$

normalising so that the weights sum to 1:

$$\overline{w}_i = \frac{w_i}{\sum_{j=1}^{k} w_j} \tag{5.21}$$

finally using these normalised weights to calculate the weighted average of the k actions:

$$a = \overline{\boldsymbol{w}} \cdot \boldsymbol{a} \tag{5.22}$$

The training for this method as presented in [23] includes an eligibility trace and is more computationally demanding as the values of all the states in $\boldsymbol{knn}$ and their respectively selected actions must be included in the calculation. The training method used is an adapted version of SARSA(λ), thus, we assume we have the values of $s, a, r, \boldsymbol{knn}, \boldsymbol{a}, s', a', \boldsymbol{knn'}$ and $\boldsymbol{a}'$.

We calculate the eligibility as the normalised weight of the states in knn for all actions in a:

$$e(s,a) = \begin{cases} \overline{w}_i & \text{if } s = knn_i \text{ and } a \in \boldsymbol{a} \\ 0 & \text{otherwise} \end{cases} \tag{5.23}$$

and δ using:

$$\delta = r + \gamma \overline{Q}(\boldsymbol{knn'}) - \overline{Q}(\boldsymbol{knn'}) \tag{5.24}$$

where $\overline{Q}(\boldsymbol{knn})$ is defined as:

$$\overline{Q}(\boldsymbol{knn}) = \sum_{i=1}^{k} \max_{a \in \tilde{\mathcal{A}}} Q(knn_i, a) \cdot \overline{w}_i \tag{5.25}$$

and the updates are performed by applying:

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + \alpha\delta e(s,a) \\ e(s,a) &\leftarrow \gamma\lambda e(s,a) \end{aligned} \tag{5.26}$$

This method suffers from the same 'level of discretization' problem as the one-step-search, but also applies discretization to the state space which may lead to further aliasing problems.

5.5 Chapter Summary

In this chapter we have presented the main approaches currently being applied to solve continuous state- and action-space RL tasks. Each method applies different techniques to overcoming the problems associated with continuous state- and action-space.

The actor critic methods and direct policy search methods attempt to approximate the policy function, thereby overcoming the problem of selecting the optimal action. They do however have disadvantages: the AC methods require more parameters and two function approximators must be trained, whereas DPS methods require a very high number of episodes of the problem to be carried out in order to estimate the gradient of the policy without taking advantage of the information contained within the value function, which is not stored. Also PG methods search only for a local maximum of the performance measure of the policy $J(\boldsymbol{\theta})$ [43].

When discretization is performed on the action-space in order to apply implicit policy methods or k-nearest neighbours there is a trade off to be made between achieving a good approximation of the value function and the computation required. Also, as these methods transform the continuous action-space problem into a discrete action-space problem, they cannot truly be thought of as solving the continuous action-space problem.

The gradient based approach to applying implicit policy methods allows the solution to the continuous action-space problem, without applying a second function approximator for the policy function; however, it has been suggested that it would be too time consuming to be practical [21, 55].

The wire-fitting method allows the solution of the continuous action-space problem, where the action can be computed in a fast time. However, the number of wires required to approximate the learned function may be too large and therefore require excessive computation to apply this approach, it has been also shown to perform poorly on the Cart-Pole benchmark problem [2, 56].

Also, CACLA has been shown to outperform many other methods from the literature including wire-fitting and an alternative AC method on the continuous action-space Cart-Pole problem [56] and natural actor-critic, and an evolutionary algorithm approach on the double pole variant [55], due to this we apply this method as the state-of-the-art to compare other methods to in Chapter 7.

Chapter 6

Implicit Policy Methods with Optimization

It has been stated that if the function approximation method used to approximate $Q(s,a)$ enables the evaluation of $\nabla_a Q(s,a)$ it would be possible to apply a gradient based method to selecting the greedy action w.r.t. $Q(s,a)$, and thus, apply implicit policy RL methods to problems with a continuous action-space [47]. However, it is also often asserted that directly solving the optimization problem would be prohibitively time-consuming in the justification of the use of an AC method [21, 55]. Here we investigate the possibility of applying optimisation techniques to directly solving:

$$\arg\max_a Q(s,a), \quad \forall a \in \mathcal{A} \tag{6.1}$$

thus eliminating the requirement of an additional function approximator for $\pi(s)$, and therefore, enabling the use of implicit policy RL methods, such as SARSA (Section 2.3), which are often the algorithms of choice when problems have a small, discrete action-space [21].

We use a single MLP to approximate $Q(s,a)$ which is updated using backpropagation with the target values of the updates calculated using the SARSA RL algorithm. We apply momentum to the updates of the ANN, however we do not overcomplicate the algorithms with further learning acceleration techniques in order to retain the simplicity, general applicability and to allow comparison to other algorithms, which are equally eligible for the application of such performance enhancements. Such extensions can, and should, be applied to get the most out of the algorithm on a particular problem, as should any available domain knowledge. However, the algorithms also were not adapted to each problem here in order to maintain the simplicity and general applicability of the proposed methods.

Here, the implicit policy algorithm applied to the updating of the value function $Q(s,a)$ is SARSA, and an MLP is utilised for the value function approximation.

In the rest of this chapter we firstly describe several optimization methods which are considered for the action selection, and compare their properties for this application. After which we discuss how this approach compares to existing approaches, before presenting the detailed description of the algorithms resulting from utilizing the four optimization methods: Genetic Algorithms; Gradient Descent; Newton's Method; and Nelder-Mead to the action selection. Preliminary results from the application of each of the aforementioned algorithms to the Cart-Pole benchmark control problem are presented to compare the performance of the algorithms both in terms of the success rate and action selection time. Followed by a chapter summary.

6.1 Optimization Methods

Several optimization methods were considered for the action selection, here we discuss two derivative-based and two derivative-free methods and compare them in terms of action selection time and final performance on the Cart-Pole problem, which is one of the benchmark control problems which is used in comparing algorithms later in the thesis (Section 7.2).

6.1.1 Derivative Based

The two derivative-based optimization methods we consider are both line-search methods, i.e. the $\boldsymbol{x}$ that we seek which optimizes the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is updated at each time-step t by first selecting the direction and then taking a step in that direction:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \alpha_t \boldsymbol{p}_t \tag{6.2}$$

where α_t is the step-size and $\boldsymbol{p}_t$ is the search direction at time-step t. The difference being how $\boldsymbol{p}_t$ is calculated. Gradient Descent uses the gradient to select the direction, whereas Newton's Method also makes use of the second derivative in calculating the search direction.

Gradient Descent

Gradient Descent [41] is a relatively straightforward optimization method which only requires the first derivative of the objective function $f(\boldsymbol{x})$, i.e. $\nabla f(\boldsymbol{x})$, from which it takes small steps in the direction which reduces the gradient:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{6.3}$$

where α is a small positive step-size parameter.

Although gradient-decent requires little computation due to relying only on the first derivative for the search direction, it can be excruciatingly slow depending on the problem and is also highly susceptible to becoming trapped in local optima [41].

Newton's Method

Newton's Method [41] is widely applied as a second order derivative based optimization technique. By taking into account the second derivative of the objective function when calculating the search direction, Newton's Method can converge much faster than if only the gradient were used [9, 41].

Newton's Method is an iterative algorithm to finding zeros of a function, but is commonly applied to finding the points at which the derivative of the function is zero, i.e. the minimum and maximum points of the function. These stationary points of a function $f(x)$ are found by repeatedly applying (6.4), which updates x to progressively approach a nearby stationary point with each iteration. As the algorithm does not distinguish between maximum and minimum points it must be applied from several different starting points, x_0, in order to find the value of x which maximizes $f(x)$; however, the extra computation required to apply the algorithm from several initial points is offset by the fast convergence of the algorithm.

$$x_n \leftarrow x_{n-1} - \frac{f'(x_{n-1})}{f''(x_{n-1})} \tag{6.4}$$

There are additional considerations when Newton's Method is applied to optimizing an objective function taking an input dimension greater than one, in such cases the Hessian must be positive definite to ensure the search direction is defined and the inverse of the Hessian must be calculated at each iteration. Modified versions of Newton's Method may be applied to overcome these problems such as quasi-Newton methods [9]. As Newton's Method relies on the derivatives it also suffers from a susceptibility to becoming trapped in local optima; however, this is mitigated by necessity to run the algorithm from several initial points.

6.1.2 Derivative Free

Derivative free optimization methods do not require the derivative of the objective function which allows their application to a wider range of function approximators, where it may not be possible, or practical, to evaluate the partial derivatives of $Q(s, a)$ w.r.t. a. Also the fact that they don't rely on the derivative for the search direction may provide an additional benefit, by helping to avoid becoming trapped in local optima, particularly when combined with a diverse set of points [17]. But this comes at the cost of the ability to utilise the derivative to guide the search, which provides fast convergence.

Genetic Algorithms

Genetic Algorithms (GA) [17], as described in Section 4.1, is an evolutionary based technique which can be applied to a wide range of problems by encoding the solution as a genetic representation and specifying a fitness function. It is clearly straightforward to apply this to optimization of a mathematical function where the fitness function is already defined as the objective function $f(\boldsymbol{x})$ and it is trivial to either directly use the input value $\boldsymbol{x}$ as a real-valued genetic representation, or to convert it to another representation, e.g. binary string.

There are many implementation decisions and parameters which must be set and tuned when applying GA: number of generations; population size; probability of crossover; probability of mutation; probability of reproduction; whether elitism is used; type of crossover: one point, n-point, uniform, etc.; parent selection method. Also there may be various other parameters depending on implementation choices, e.g. if the genes are represented by binary strings the length of the binary string must be specified.

Moreover, the number of function evaluations required to apply GA is very high as the whole population must be evaluated at each generation even if we avoid recalculating the fitness of individuals that are reproduced from previous generations, which will not be high unless the probability of mutation and the probability of crossover are both very low, which would adversely affect the GAs ability to search the solution space.

Nelder Mead

The Nelder-Mead optimization technique [38] is a simplex method for minimizing a function $f(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^n$ in which the dimension of the simplex is $n+1$ and each element of the simplex is a point in $\mathbb{R}^n$. The algorithm proceeds by, at each iteration, performing one of four operations: shrink, reflect, expand or contract to update the simplex according to certain conditions, as can be seen in Algorithm 6.1 on the following page adapted from [34]. There are four parameters: ρ, χ, γ and σ, which affect reflection, expansion, contraction and shrinkage respectively. We set these parameters to the values: $\rho = 1$, $\chi = 2$, $\gamma = 0.5$, $\sigma = 0.5$ as given in [34].

As a derivative-free method Nelder-Mead does not require $\nabla_a Q(s,a)$ or $\nabla_a^2 Q(s,a)$ and thus can be applied to any function approximation method. Also, the action selection can be very quick depending only on the number of iterations Nelder-Mead is run and the speed of calculating $Q(s,a)$, which is very fast with most function approximation techniques.

Algorithm 6.1 Nelder Mead optimization algorithm

```
procedure NelderMeadMin($\boldsymbol{x}$, $f(\cdot)$, $\rho$, $\chi$, $\gamma$, $\sigma$)
    INITIALISE_SIMPLEX
    for $i \leftarrow 1, max\_iterations$ do
        Sort simplex such that $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \cdots \leq f(\boldsymbol{x}_{n+1})$
        $\bar{\boldsymbol{x}} \leftarrow \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{x}_j$        ▷ Calculate centroid of $n$ best points
        $\boldsymbol{x}_r \leftarrow \bar{\boldsymbol{x}} + \rho(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$        ▷ reflection
        if $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_r) < f(\boldsymbol{x}_n)$ then
            $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$        ▷ Accept reflection
            next iteration
        end if
        if $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$ then
            $\boldsymbol{x}_e \leftarrow \bar{\boldsymbol{x}} + \chi(\boldsymbol{x}_r - \bar{\boldsymbol{x}})$        ▷ expansion
            if $f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r)$ then
                $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_e$        ▷ Accept expansion
                next iteration
            else if $f(\boldsymbol{x}_e) \geq f(\boldsymbol{x}_r)$ then
                $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$        ▷ Accept reflection
                next iteration
            end if
        end if
        if $f(\boldsymbol{x}_r) \geq f(\boldsymbol{x}_n)$ then        ▷ Contract
            if $f(\boldsymbol{x}_n) \leq f(\boldsymbol{x}_r) < f(\boldsymbol{x}_{n+1})$ then
                $\boldsymbol{x}_c \leftarrow \bar{\boldsymbol{x}} + \gamma(\boldsymbol{x}_r - \bar{\boldsymbol{x}})$        ▷ outside contraction
                if $f(\boldsymbol{x}_c) \leq f(\boldsymbol{x}_r)$ then
                    $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_c$        ▷ Accept contraction
                    next iteration
                end if
            else if $f(\boldsymbol{x}_r) \geq f(\boldsymbol{x}_{n+1})$ then
                $\boldsymbol{x}_c \leftarrow \bar{\boldsymbol{x}} - \gamma(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$        ▷ inside contraction
                if $f(\boldsymbol{x}_c) < f(\boldsymbol{x}_{n+1})$ then
                    $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_c$        ▷ Accept contraction
                    next iteration
                end if
            end if
        end if
        $\boldsymbol{x}_j \leftarrow \boldsymbol{x}_1 + \sigma(\boldsymbol{x}_j - \boldsymbol{x}_1), \quad j = 2, 3, \ldots, n+1$        ▷ Apply shrink
    end for
    return $\boldsymbol{x}_1$
end procedure
```

6.1.3 Comparison

One way to classify the optimization algorithms, for a general comparison, is by whether or not they require the derivatives of the objective function to guide the search. Derivative based methods can utilise the derivatives to select the search direction, which can lead to faster convergence; however, they are more susceptible to getting stuck in the nearest local maxima to the initial point, whereas derivative free methods are not guided by the gradient and therefore are less likely to converge to the nearest local maxima, although they also are clearly not guaranteed to converge to the global maximum.

A simple approach to overcome such difficulties is to apply the gradient based method from several initial points $\boldsymbol{x}_0$ and select the maximum of the local maxima found from each starting point, which was applied to both derivative-based methods here. Another problem with first order derivative based methods is that they can be slow, second order methods, such as Newton's Method, can help overcome this problem [9, 41].

An obvious deciding factor is that as derivative-based methods require the calculation of the derivative; thus, if this is not possible derivative free methods are the only option.

6.2 Relation to Other Approaches

Here we list related approaches from the literature including application of the gradient in action selection [30, 47] and discretization of the action-space [47]. Both of which allow the use of implicit policy RL algorithms in the continuous action-space.

6.2.1 One-Step-Search

The one-step-search [47] is one of the most straight-forward discretization methods in which the actions are limited to the discrete subset of actions:

$$\mathcal{A} = \{a_{min}, a_{min} + \Delta_a, \ldots, a_{max} - \Delta_a, a_{max}\} \tag{6.5}$$

where Δ_a is a tunable parameter to control the granularity of discretization. All of the actions in the discretized subset are evaluated and the one with the highest value is selected as the greedy action.

This method requires the granularity of the discrete actions to be set, but such information about the required level of granularity to obtain a sufficiently close approximation of the value function is rarely available to the implementer; furthermore, if the required level of granularity is too fine $|\mathcal{A}|$ would still be too large to evaluate all possible actions. This will then lead to a trade-off between computational time and quality of the selected actions.

6.2.2 Adaptive Action Modification

The Adaptive Action Modification approach [42] claims to allow action selection over a continuous action-space by limiting the possible actions to only two possible actions:

$$\mathcal{A}(s_t) = \{a_{t-1} \pm \Delta_a\} \tag{6.6}$$

this affects how fast or slow changes in action values can occur, depending on the value of Δ_a, and also precludes the same action being selected at two consecutive time-steps. Moreover, as there are parameters specifying the maximum and minimum values of Δ_a, the overall technique is a form of discretization at some varying level of granularity between these limits.

6.2.3 Gradient

The application of gradient-descent to continuous action selection has been suggested [30, 47]; however, it is not generally applied, possibly due to the slow and poor performance we experienced (Section 6.4).

6.3 Detailed Description

Here we describe the details of the algorithms resulting from applying Genetic Algorithms; Gradient Descent; Newton's Method and Nelder Mead to continuous action selection.

6.3.1 Genetic Algorithms

The application of GA to action selection is presented in Algorithm 6.2 on the next page. The representation used was binary strings and the crossover method was one point crossover. Roulette wheel parent selection was used. A value was added to the calculated fitness in order to ensure the fitness of all individuals was positive to allow the roulette wheel method to be applied, this value is obviously dependent on the possible values of the objective function, $Q(s, a)$. Actions were selected in the range $[-1, 1]$ as the input to the ANN was adjusted to this range. Rescaling of the action to the allowable range for the problem was done after action selection was complete. The initialisation of the population was done by generating random values within the allowable range for the problem, i.e. $[-1, 1]$, to set the initial value of each individual

Algorithm 6.2 Genetic Algorithm Action Selection

```
procedure GetActionGA($s$)
    for all $generation \in \{1, 2, \ldots, MAX\_GENERATIONS\}$ do
        for all $i \in \{0, 1, \ldots, POPULATION\_SIZE - 1\}$ do
            INITIALISE($population_i$)          ▷ Initialise population in valid range
        end for
        for all $i \in \{0, 1, \ldots, POPULATION\_SIZE - 1\}$ do
            $population_i.fitness \leftarrow Q(s, population_i.real)$
        end for
        SORT($\boldsymbol{population}$)
        $new\_population_0 \leftarrow population_0$          ▷ Copy elite individuals
        $new\_population_1 \leftarrow population_1$
        while SIZE($\boldsymbol{new\_population}$) $< POPULATION\_SIZE$ do
            $\boldsymbol{parents} \leftarrow$ ROULETTE_WHEEL($\boldsymbol{population}$)
            if UNIFORM_RAND$(0, 1) < P_c$ then
                $\boldsymbol{children} \leftarrow$ CROSSOVER($\boldsymbol{parents}$)          ▷ Perform crossover
            else
                $\boldsymbol{children} \leftarrow \boldsymbol{parents}$          ▷ Perform reproduction
            end if
            for $i \in \{1, 2\}$ do
                for $j \in \{0, 1, \ldots, L - 1\}$ do
                    if UNIFORM_RAND$(0, 1) < P_m$ then
                        FLIPBIT($children_i, j$)
                    end if
                end for
            end for
            $new\_population.insert(\boldsymbol{children})$
        end while
        $\boldsymbol{population} \leftarrow \boldsymbol{new\_population}$
        CLEAR($new\_population$)
    end for
    return MAX($\boldsymbol{population}$)$.real$
end procedure
```

in the population. The notation $population_i.real$ is used to represent the real number representation of the value of the i^{th} member of the population. The individuals also stored their own fitness value in $population_i.fitness$, for the fitness of the i^{th} member of the population.

The copying of the best individuals is included in the algorithm, however, the use of elitism can be set by a parameter, as was the case in the implementation used. Other tunable parameters in this algorithm include: $MAX\_GENERATIONS$, $POPULATION\_SIZE$, L (length of the binary string), P_c (probability of crossover), P_m (probability of mutation).

Algorithm 6.3 Gradient Descent Action Selection

```
procedure GetActionGD($s$)
    $a_{best} \leftarrow 0$
    for all $a_0 \in \{a_{min}, a_{min} + a_\Delta, \ldots, a_{max} - a_\Delta, a_{max}\}$ do
        $a \leftarrow a_0$
        $a_{previous} \leftarrow a_{max} + 10$
        for $i \leftarrow 1, max\_iterations$ do
            $a \leftarrow \left[a + \frac{\partial Q(s,a)}{\partial a}\right]_{a_{min}}^{a_{max}}$        ▷ Limit $a$ to allowable range
            if $Q(s,a) > Q(s, a_{best})$ then
                $a_{best} \leftarrow a$
            end if
            if $|a - a_{previous}| < \zeta$ then
                break        ▷ If $a$ has converged move on to next $a_0$
            end if
            $a_{previous} \leftarrow a$
        end for
    end for
    return $a_{best}$
end procedure
```

6.3.2 Gradient Descent

The application of gradient descent to action selection we used is presented in Algorithm 6.3. The parameters were set as follows $STEP\_SIZE = 0.1$; $a_\Delta = 0.1$; and $MAX\_ITERATIONS = 20$. The optimization was run from several starting actions between the minimum and maximum actions as the gradient descent optimization method is susceptible to converging to local optima.

6.3.3 Newton's Method

The application of Newton's Method to action selection can be seen in Algorithm 6.4 on the following page. Few iterations of Newton's Method were required to converge to a static point in the experiments carried out (Chapter 7), therefore this algorithm does not introduce a significantly large run-time overhead compared with discretization methods, whilst also allowing any action in the continuous range to be selected. Unlike discretization methods, NM-SARSA is not limited to the discrete actions $a_{min} + n \cdot a_\Delta$, which are merely the starting points to apply Newton's Method from; therefore, it is also possible to apply a far larger a_Δ for this algorithm compared with action-space discretization methods, in the following experiments $a_\Delta = 0.5$ was used when searching for actions in $[-1, 1]$; actions were in this range, regardless of the problem, due to scaling for input to the MLP, which were rescaled to the desired range after action selection was complete, in order to apply them to the simulation.

Algorithm 6.4 Newton's Method Action Selection

```
procedure GetActionNM($s$)
    $a_{best} \leftarrow 0$
    for all $a_0 \in \{a_{min}, a_{min} + a_{\Delta}, \ldots, a_{max} - a_{\Delta}, a_{max}\}$ do
        $a \leftarrow a_0$
        $a_{previous} \leftarrow a_{max} + 10$
        for $i \leftarrow 1, max\_iterations$ do
            $a \leftarrow \left[a - \frac{\partial Q(s,a)/\partial a}{\partial^2 Q(s,a)/\partial a^2}\right]_{a_{min}}^{a_{max}}$          ▷ Limit $a$ to allowable range
            if $Q(s,a) > Q(s,a_{best})$ then
                $a_{best} \leftarrow a$
            end if
            if $|a - a_{previous}| < \zeta$ then
                break          ▷ If $a$ has converged move on to next $a_0$
            end if
            $a_{previous} \leftarrow a$
        end for
    end for
    return $a_{best}$
end procedure
```

The maximum number of iterations of Newton's Method used to produce the results in Table 6.1 on page 68 was 5; however, as it was noted that a often converged in fewer iterations, the previous action was recorded and compared to the current action at each iteration to check for early convergence, if there was no significant change in the action ($|a - a_{previous}| < \zeta$) the inner loop was terminated immediately.

Also, as it was observed that occasionally a reached a point where it was switching between two or more values, a_{best}, which maximizes the value function, was updated at every iteration to ensure the action which maximizes the value function was returned.

Whilst this method does take longer than presenting a policy network with the current state, as is the case with actor-critic or policy gradient methods, it does not take a prohibitively long time, the average time taken to select an action using this method was 0.00003 seconds on the Cart-Pole problem, as is shown in Table 6.1 on page 68.

6.3.4 Nelder Mead

Here it is shown how the Nelder-Mead derivative-free optimization method to the action selection. Many of the details of this approach are the same as with the Newton's Method approach and the same MLP was used for the function approximation. The only difference in the approach was the optimization method used. Although Nelder Mead does not require the ability to calculate derivatives of the value function, we still utilise the same MLP architecture throughout these experiments, which allows direct

Algorithm 6.5 Nelder Mead Action Selection

```
procedure GetActionNelderMead($\rho$, $\chi$, $\gamma$, $\sigma$, $iterations$, $s$, $Q(\cdot,\cdot)$)
    $f(\cdot) \leftarrow -Q(s,\cdot)$
    $x_0 \leftarrow 0$
    $x_1 \leftarrow \text{uniform}(a_{min}, a_{max})$        ▷ uniform random action in $[a_{min}, a_{max}]$
    for $i \leftarrow 1, iterations$ do
        sort($\boldsymbol{x}$,$f(\cdot)$)
        $x_r \leftarrow [x_0 + \rho(x_0 - x_1)]_{a_{min}}^{a_{max}}$        ▷ compute reflection points
        if $f(x_r) < f(x_0)$ then
            $x_e \leftarrow [x_0 + \chi(x_r - x_1)]_{a_{min}}^{a_{max}}$        ▷ expand
            if $f(x_e) < f(x_0)$ then
                $x_1 \leftarrow x_e$        ▷ accept $x_e$
            else
                $x_1 \leftarrow x_r$        ▷ accept $x_r$
            end if
            continue        ▷ next iteration
        else if $f(x_r) \geq f(x_0)$ then
            if $f(x_0) \leq f(x_r) \wedge f(x_r) < f(x_1)$ then
                $x_c \leftarrow [x_0 - \gamma(x_r - x_0)]_{a_{min}}^{a_{max}}$
                if $f(x_c) \leq f(x_r)$ then
                    $x_1 \leftarrow x_c$        ▷ accept $x_c$
                    continue        ▷ next iteration
                end if
            end if
        end if
        $x_1 \leftarrow [x_0 + \sigma(x_1 - x_0)]_{a_{min}}^{a_{max}}$        ▷ perform shrink step
    end for
    sort($\boldsymbol{x}$,$f(\cdot)$)
    return $x_0$
end procedure
```

comparison between the approaches.

The optimization methods have different parameters, and when using Nelder Mead we did not need to keep track of the previous action, as the simplex retains the best action from the previous iteration. The details can be seen in Algorithm 6.5, where $[x]_a^b$ limits x to be within a and b and sort($\boldsymbol{x}$, $f(\cdot)$) sorts the vector $\boldsymbol{x}$ in order of the output of $f(\cdot)$, i.e. the sorted $\boldsymbol{x}$ will satisfy:

$$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_n). \tag{6.7}$$

Unlike Newton's Method which started from the same initial actions each time the method was called, we achieved better results when initialising the actions to 0 and a uniformly randomly selected value in the range $[a_{min}, a_{max}]$. This introduction of this stochastic element to the algorithm also creates a difference with the Newton's Method

Table 6.1: Optimization Methods Applied to Cart-Pole Problem

Optimization Method	Average Action Selection Time (sec)	Successful runs (%)	Successful Testing Runs (%)
Gradient	0.00052s	100%	94%
Newton's Method	0.00003s	100%	98%
Nelder Mead	0.00003s	100%	100%
Genetic Algorithm	0.0033s	0%	0%

approach, which is completely deterministic. The number of iterations performed was controlled by a parameter, which was set to 10 in producing the results in Table 6.1. Although this could also be tuned for the application, to take full advantage of the maximum allowable action selection time.

The problems we tested these algorithms on in Chapter 7 all have a scalar action; however, this method is clearly well suited to continuous, vector-valued actions, and it would be interesting to see how this approach performs in higher dimensional action-spaces.

6.4 Preliminary Results

Each of the aforementioned methods were applied to the action selection on the Cart-Pole problem (described in more detail in Chapter 7) the results of which are presented in Table 6.1, giving the median action selection time as well as the percentage of runs which produced controllers able to successfully balance the pole in one of the training episodes and the percentage which could also balance the pole in all ten testing trials for the given run, in which exploration and training was turned off. The details of this experiment are the same as described in Section 7.2.

GA was infeasibly slow, despite the fact that the quantity of generations and the population size used were insufficient to succeed in the benchmark task, and therefore, is not considered an acceptable action selection method, despite the longest amount of time being spent on parameter tuning of the GA approach. The gradient method also produced poor performance in terms of both action selection time: on average taking more than fifteen times longer to select the greedy action than Newton's Method and Nelder Mead, and also producing a learnt policy with inferior performance on the testing trials. Therefore, we focus on the two methods which performed well in a reasonable length of time: Newton's Method and Nelder Mead.

6.5 Chapter Summary

The optimization methods applied here allow fast, accurate greedy action selection without the requirement of a separate policy function which adds more complexity to the training process. We are not aware of other such applications in the literature, only of suggestions of the possible application of the gradient [30, 47], which performed worse than the methods used here (Table 6.1 on the preceding page); or discretization methods which do not allow the use of a true continuous action space and thus force the trade-off between speed and precision. Such a compromise may result in slow and/or poor action selection.

The application of GA to the action section was also considered, as GA is a global derivative-free optimization technique, however, after some experimentation it became apparent that GA, even with an insufficient number of generations and population size to achieve acceptable performance, requires too many evaluations of the objective function and therefore takes a prohibitively long time to be utilised in action selection. This problem is particularly important as this optimization must be performed even once the agent is fully trained due to the fact that the policy is not stored separately from the value function, which is why we selected the Nelder Mead method and Newton's Method which both performed very well and selected actions in a very short time.

Although both Nelder Mead and Newton's Method can be applied to optimising functions taking a vector as input rather than a scalar, the computation involved in applying Newton's Method may be excessive as the inverse of the Hessian must be computed; furthermore, as Newton's Method requires that the Hessian is positive definite, thus modified versions of the algorithm may be required which maintain the positive definiteness of the Hessian, such as quasi-Newton methods [9]. However, the Nelder-Mead approach would require minimal extra computation time when extended to higher dimensional action-spaces, therefore, it would be interesting to observe the performance on such problems as future work.

Chapter 7

Experiments

In this chapter the novel approaches proposed in Chapter 6: NM-SARSA and NelderMead-SASRSA are applied to three continuous state- and action-space RL benchmark control problems from the literature: Acrobot (Section 7.1), Cart-Pole (Section 7.2) and double Cart-Pole (Section 7.3). These benchmark problems were applied as they are difficult continuous state- and action-space control problems used in the literature and therefore allow the comparison of the proposed methods to existing methods.

The performance of these novel algorithms is compared to that of existing state-of-the-art algorithms from the literature, including CACLA, adaptive-critic and GP, for which we present novel results on the Acrobot problem. We also compare the results achieved by all of these methods to those from the literature.

Each of the benchmark control problems are presented separately, including details of the task; details of the application of each approach; and results including a comparison of the performance of the different approaches. At the end of the chapter there is also a summary including a comparison of the approaches across the different tasks.

All results presented in this section were obtained through software implementations of the control problems in C++11, which were run on a laptop with an i5 processor. The GP approach utilises multi-threaded programming in order to evaluate the fitness of the two children resulting from crossover in parallel.

7.1 Acrobot

The acrobot is a two link robot, based on a human acrobat. The links are connected by an actuated joint and one end is connected to a bar by an unactuated joint [50], see Fig. 7.1 on page 72.

The state $s = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^\top$ comprises the first and second angles θ_1 and θ_2 and the respective angular velocities $\dot{\theta}_1$ and $\dot{\theta}_2$. The action a is the torque τ to be applied to the actuated joint. The acrobot is a difficult control task as it is a four dimensional, highly non-linear, under-actuated control problem [50, 51].

7.1.1 Existing Approaches

There have been many approaches to the acrobot problem, but often they solve only the simpler half-swing-up problem [53, 62], and are also often restricted to discrete control $\tau \in \{\pm\tau_{max}, 0\}$. Also, many of the full swing-up approaches either allow large torque values [15, 36, 50, 59, 61, 63] or limit the control to discrete actions [8, 29, 57].

The balance task is currently best solved by the linear quadratic regulator (LQR) traditional control method [51], which is often applied even when computational intelligence approaches are utilised in solving the swing-up task [19, 29, 59]. However, the LQR approach, unlike computational intelligence methods, requires knowledge of the equations of motion in order to calculate the coefficients of the linear controller. Moreover, this method has a very small basin of attraction from which it is able to balance the acrobot [51].

Here we focus on the swing-up control problem of the acrobot, as it has successfully been achieved by several methods and therefore allows the methods presented here to be, relatively, easily compared to many approaches to control problems. Whereas the LQR controller applied to almost all approaches of the balance control [29, 50, 51, 61, 63].

7.1.2 Details

The task of the acrobot is to swing up, in the minimum time, from the initial state $s_0 = [0, 0, 0, 0]^\top$ to the inverted, unstable, target state $s_t = [\pi, 0, 0, 0]^\top$ and then to remain balanced in that position. This is often split into two tasks: swing-up and balance, with two distinct controllers which are switched at a given point. The controller switching point can be either found through trial and error [50] or by applying other techniques to find the most appropriate point to switch [63].

The results presented here are based on computer simulations of the acrobot using the equations of motion (7.1) as described by [53]. The parameter values used are shown in Table 7.1 on the following page. The angular velocities were limited to $-4\pi \leq \dot{\theta}_1 \leq 4\pi$ and $-9\pi \leq \dot{\theta}_2 \leq 9\pi$. Continuous values were allowed for $a = \tau \in [-2, 2]$. The acrobot state s was updated every 0.05 simulated seconds, using the Euler method of numerical integration, and the controller selected the τ to apply to the actuated joint every 0.2 seconds.

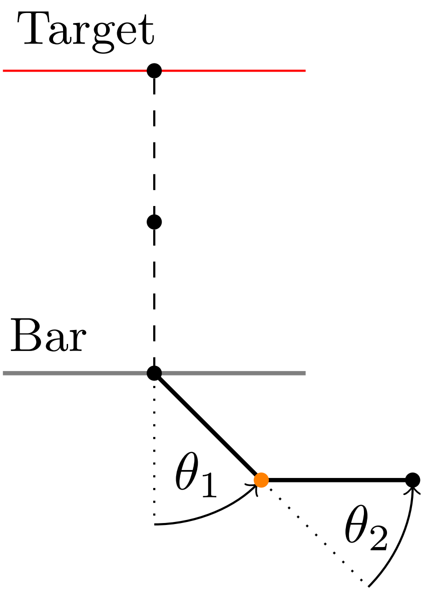


Figure 7.1: The full swing-up of the acrobot with target position. The dashed line represents target position and the orange joint is the actuated joint.

Table 7.1: Acrobot parameter values

Parameter	Symbol	Value
Mass of link 1	m_1	1
Mass of link 2	m_2	1
Length of link 1	l_1	1
Length of link 2	l_2	1
Length to centre of mass of link 1	l_{c1}	0.5
Length to centre of mass of link 2	l_{c2}	0.5
Inertia of link 1	I_1	1
Inertia of link 2	I_2	1
Gravitational constant	g	9.8

$$\begin{aligned}
\ddot{\theta}_1 &= -d_1^{-1}\left(d_2\ddot{\theta}_2 + \phi_1\right)\\
\ddot{\theta}_2 &= \left(\tau + \frac{d_2}{d_1}\phi_1 - m_2 l_1 l_{c2}\dot{\theta}_1^2 \sin\left(\theta_2\right) - \phi_2\right) \div \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)\\
d_1 &= m_1 l_{c1}^2 + m_2\left(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2}\cos\left(\theta_2\right)\right) + I_1 + I_2\\
d_2 &= m_2\left(l_{c2}^2 + l_1 l_{c2} cos(\theta_2)\right) + I_2\\
\phi_1 &= -m_2 l_1 l_{c2}\dot{\theta}_2^2 \sin\left(\theta_2\right) - 2 m_2 l_1 l_{c2}\dot{\theta}_2\dot{\theta}_1 \sin\left(\theta_2\right)\\
&\quad + \left(m_1 l_{c1} + m_2 l_1\right) g \cos\left(\theta_1 - \frac{\pi}{2}\right) + \phi_2\\
\phi_2 &= m_2 l_{c2} g \cos\left(\theta_1 + \theta_2 - \frac{\pi}{2}\right)
\end{aligned} \tag{7.1}$$

7.1.3 Method

The parameters used for each approach are those which were found to perform best on this task. When Gaussian exploration was applied $\sigma^2 = 1$ was used in generating the random exploration value.

Genetic Programming Approach

GP was applied to the acrobot swing-up task [14] by including the angles and angular velocities, which collectively form the state s, in the set of terminal nodes, along with a randomly generated constant $R \in [-10, 10]$:

$$\mathcal{TS} = \{\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2, R\}$$

the limit applied to R was selected through experimentation and this range was found to produce the best results in these experiments.

The fitness of each individual in the population was calculated by running a simulation of the acrobot where the GP program was presented with the current state s and returned the action a, which in this case was the torque τ to apply to the actuated angle. The action was limited to the allowable range using:

$$\tau = \begin{cases} \tau_{MIN}, & \text{if } \tau < \tau_{MIN} \\ \tau_{MAX}, & \text{if } \tau > \tau_{MAX} \\ \tau, & \text{otherwise} \end{cases}$$

The function set used for this task was:

$$\mathcal{FS} = \{+, -, \times, \div, \tanh, \text{abs}, \text{iflt}\}$$

where $\div$ was the protected divide function, and iflt is the if-less-than function, as described by [32]. The details of all functions used are given in Table 7.2 on the next page. The functions included in $\mathcal{FS}$ were determined through experimentation.

Each run began with the initialization of the population, the details of which is given in Table 7.3 on the following page, the fitness of each individual was calculated and then 100 generations of GP were executed, there is no target fitness to use for early stopping before all generations are complete, as it is not clear how long the optimal policy would take to swing-up the acrobot [7]. Experimentation was carried out with more generations but this did not lead to improved performance.

The fitness was evaluated using the adjusted fitness to exaggerate the difference

Table 7.2: Functions used for the function sets

Symbol	Function	Formula
$+$	add	$arg_1 + arg_2$
$-$	subtract	$arg_1 - arg_2$
$\times$	multiply	$arg_1 \times arg_2$
$\div$	protected divide	$\begin{cases} arg_1/arg_2, & \text{if } arg_2 \neq 0 \\ 0, & \text{otherwise} \end{cases}$
tanh	hyperbolic tangent	$\tanh(arg_1)$
abs	absolute value	$\|arg_1\|$
iflt	if less than	$\begin{cases} arg_3, & \text{if } arg_1 < arg_2 \\ arg_4, & \text{otherwise} \end{cases}$

Table 7.3: Genetic programming parameters

Parameter	Value
Probability of crossover (P_c)	90%
Probability of reproduction (P_r)	10%
Probability of mutation (P_m)	0%
Population size (M)	2000
Number of generations (G)	100
Probability of choosing internal points for crossover (P_{ip})	90%
Maximum size of initial GP trees (D_i)	6
Maximum size of GP trees created during run (D_c)	17
Initialisation method	grow
Selection method	fitness proportionate over selection
Over-selection cumulative percentage (c)	16%

between the best individuals [32]. The raw fitness was calculated as:

$$f(i) = t_{up}(i) \tag{7.2}$$

where t_{up} is the time to reach the inverted position, defined as when the following evaluated to true:

$$\cos(\theta_1) \leq -0.95 \wedge \cos(\theta_2) \geq 0.95 \wedge |\dot{\theta}_1| \leq 0.5 \wedge |\dot{\theta}_2| \leq 0.5$$

As fitness proportionate over-selection was used, the normalized fitness was calculated from from the raw fitness (7.2) as described in Section 4.2.3.

Adaptive Critic Approach

The AC method as used in [48] which was described in Section 5.1 was applied to the full swing-up task of the acrobot. A slight modification had to be made in order to apply the method to the swing-up problem rather than the cart-pole balancing problem mentioned in that work. The approach in [48] recorded a history of the number of time-steps taken before the pole fell down in the last 6 trials, and if 5 of them was within 5 time-steps of the previous trial the training was considered to have become 'stuck' and the weights of the neural networks were re-initialised. This was not possible on the acrobot swing-up task due to the fact that when the trial fails the time will always be the maximum time; thus, the maximum reward received in each of the previous 6 trials was recorded and if they were within 1 of the previous trial for 5 trials the neural networks weights were re-initialised. The parameter values used are presented in Table 7.4 on the next page along with those of the other methods. Learning rate was reduced by 0.05 until it reached a minimum value of 0.005, and the number of updates performed on the neural networks and error thresholds were: $N_c = 10$; $N_a = 200$; $T_c = 0.01$; $T_a = 0.001$.

CACLA

CACLA as described in Section 5.1.2 was applied to the Acrobot problem with MLPs for both the actor and critic networks, the general RL parameters are listed in Table 7.4 on the following page with the CACLA specific variance parameter $\beta = 0.001$.

Newton's Method SARSA

The general RL parameters used for the NM-SARSA approach can be seen in Table 7.4, the parameters specific to the Newton's Method optimization algorithm were: maximum iterations of Newton Method was 20, initial action step-size was 1 and early convergence $\zeta = 0.001$. The learning rate was reduced by multiplying it by 0.85 at each iteration, and after each trial the exploration coefficient was reduced by 0.0001 until it reached 0.1 and where it remained for the rest of the trials.

Nelder Mead SARSA

The general RL parameter settings used for the NelderMead-SARSA approach can be seen in Table 7.4 alongside those of the NM-SARSA approach, the same parameters were used except for those specific to the optimization method in order to retain the comparability between the two approaches. In the NelderMead approach, the number of iterations of the Nelder Mead optimization algorithm for action selection was

Table 7.4: Acrobot RL Agent Parameters

Parameter	NM-SARSA	NelderMead	CACLA	Adaptive Critic
Hidden Nodes	12	12	7	13
RL step-size (α)	0.2	0.2	0.2	N/A
RL discount rate (γ)	0.9	0.9	0.9	$\alpha = 0.95$
Learn rate	0.3	0.3	0.2	0.5
Momentum	0.75	0.75	0	0
Exploration	Gaussian	Gaussian	Gaussian	Gaussian

ten. The other parameters for Nelder Mead were as specified in Section 6.1.2 and not changed, and the initial simplex was 0 and a random value.

The exploration coefficient was reduced by 0.0001 after each trial until it reached a value of 0.1, where it remained for any further trials. The ANN learning rate was reduced at each time-step by multiplying it by 0.85.

7.1.4 Results

The performance of the various approaches to the acrobot swing-up task are listed in table 7.5 on page 79. As can be seen, all methods were able to match the performance of alternative approaches from the literature, in some cases outperforming existing approaches.

As can be seen in table 7.5 on page 79, the GP approach produced the controller with the fastest swing-up time. The fitness improvement across the generations of the best run can be seen in Fig. 7.2 on the next page, and the swing-up trajectory can be seen in Fig 7.3 on page 78.

However, the GP method was only a fraction of a second faster than the Nelder-Mead SARSA method; moreover, as can also be seen in table 7.5 on page 79 the expected number of simulations used to evolve the controller using GP is far more than is used by all other methods presented here and therefore took far longer to run, days rather than hours. However, this was still far fewer than that of another TD method from the literature [11]. The two AC methods: Adaptive-Critic and CACLA both produced similar results, which were also similar to the results of another RL method from the literature [11]. The GP method and the two implicit policy methods: NM-SARSA and Nelder-Mead SARSA all outperformed existing methods which use the same torque limits; furthermore, the swing-up speed achieved using GP and implicit policy methods was comparable to the results achieved by other methods in the literature which permit far greater torque values [36, 50].

[1] Expected number of fitness evaluations based on generations; population size; and P_c.

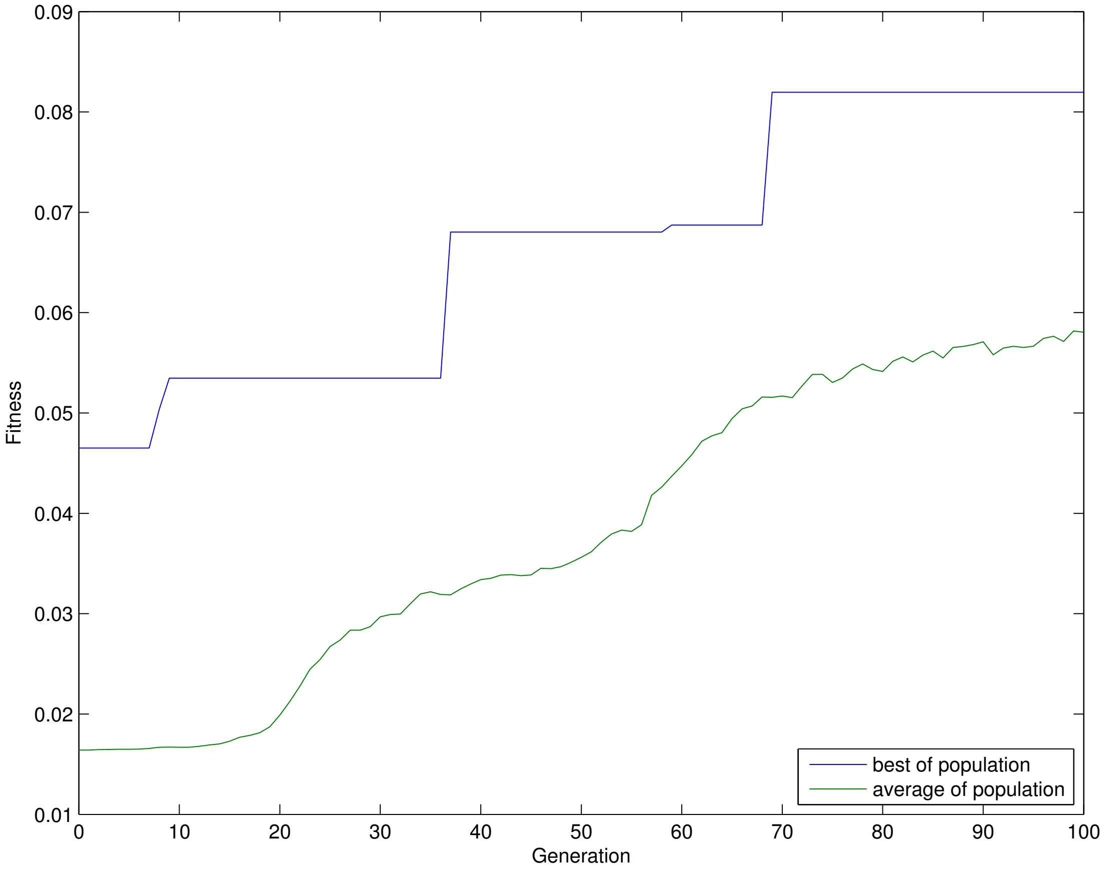


Figure 7.2: Fitness of the best individual and average of population taken from an average run of the swing-up task.

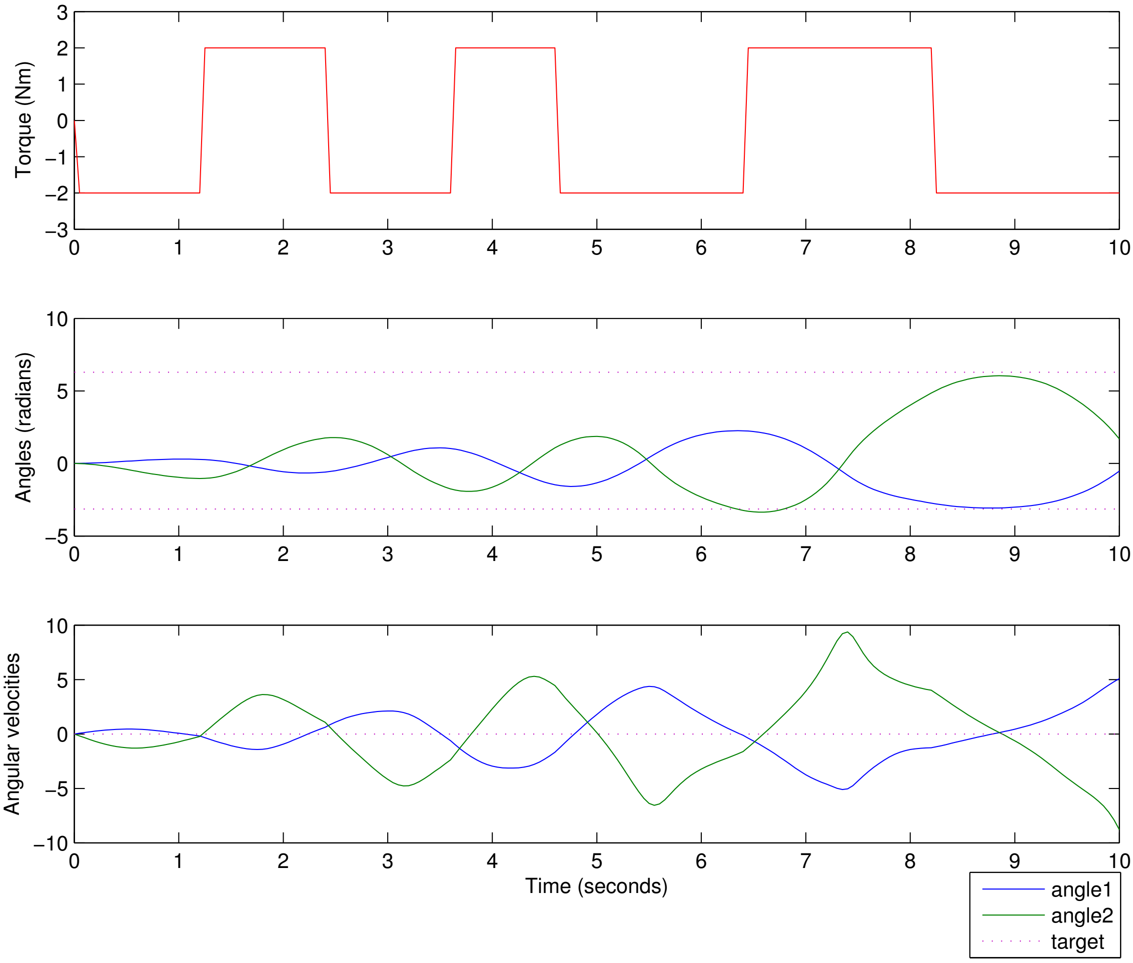


Figure 7.3: The best genetic programming acrobot Swing-up controller. At 8.8 seconds the angles are close to $-\pi$ and 2π respectively (equivalent to the balance position), at the same time as the angular velocities are both very close to zero.

Table 7.5: Acrobot Swing-up Time of Different Techniques

Method	Torque Range (Nm)	Swing-up Time (s)	Success Rate (% Runs)	Number of Training Trials (Average)
Genetic programming [14]	**[-2,2]**	**8.8**	**100%**	**182,000**[1]
Adaptive Critic	**[-2,2]**	**10.95**	**100%**	**152.65**
CACLA	**[-2,2]**	**10.8**	**100%**	**100.56**
NM-SARSA	**[-2,2]**	**9.3**	**82%**	**523.95**
Nelder-Mead SARSA	**[-2,2]**	**9.05**	**86%**	**969.39**
Continuous TD(λ) [11]	[-2,2]	~11	Not specified	1,000,000
Control theory [36]	[-15,25]	~9	N/A	N/A
Partial feedback linearization [50]	$(-\infty, \infty)$	~10	N/A	N/A

Whilst the best results were achieved using GP in this experiment, the approach is excruciatingly time consuming: each experiment took days to run, even when exploiting the availability of multiple processor cores to evaluate both children from crossover in parallel; furthermore, a very large number of these runs is required in order to find the combination of functions for the function set and to tune the other parameters: number of generations; number of runs; initialization; maximum depth; elitism; etc. NelderMead-SARSA, on the other hand, achieved comparable results with very little parameter tuning at all, also the results of applying NM-SARSA also were only slightly worse. If these approaches were given the amount of parameter tuning time applied to GP it is possible that they would exceed the performance of GP.

However, whilst NM-SARSA and NelderMead-SARSA both performed very well in terms of minimum swing-up time, they both required several times more training episodes than the AC methods, on average, before they achieved the first successful swing-up trial. They were also the only methods which suffered with some unsuccessful runs. It is possible that more parameter tuning may yield agents which do not suffer from these problems, another approach to addressing these problems may be to incorporate a test that learning is progressing with a weight re-initialisation, similar to that used by the Adaptive-Critic.

[1] Expected number of fitness evaluations based on generations; population size; and P_c.

7.2 Cart-Pole

The Cart-Pole problem is a cart on a limited track with a pole attached to the middle by a free joint. The pole must remain balanced by applying force to the cart, without the cart reaching the edge of the track.

The state vector comprised the pole angle; pole angular velocity; cart distance from centre of track; and cart velocity $s = [\theta, \dot{\theta}, x, \dot{x}]^\top$. The action was the force applied to the cart $a = F \in [-10, 10]$.

7.2.1 Existing Approaches

The Cart-Pole problem is a very well known control benchmark problem used by both the EA [22, 32] and RL [2, 46, 48, 53, 56] communities.

Often the actions are limited to discrete values [32, 48], but here we attempt the continuous action cart-pole problem. We select CACLA as a state-of-the-art method to compare the results of the proposed methods of NM-SARSA and NelderMead-SARSA to, as it has previously outperformed several continuous state- and action-space RL algorithms on this task [56].

7.2.2 Details

The standard Cart-Pole problem, is a widely used control benchmark problem [48, 56]. In many applications the actions are limited to $\mathcal{A} = \{0, \pm 10\}$N; here, however, continuous actions $\mathcal{A} = [-10, 10]$N are permitted in order to evaluate the performance of the algorithms in the continuous action-space. The parameters of the environment used in this experiment are listed in Table 7.6 on the following page, and the equations of motion used to update the environment are specified in (7.3), which is the same as [48].

$$\begin{aligned}
\ddot{\theta} &= \frac{g \sin\theta + \cos\theta \left(-F - ml\dot{\theta}^2 \sin\theta + \mu_c \operatorname{sgn}(\dot{x})\right) - \frac{\mu_p \dot{\theta}}{ml}}{l\left(\frac{4}{3} - \frac{m\cos^2\theta}{m_c + m}\right)} \\
\ddot{x} &= \frac{F + ml\left(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right) - \mu_c \operatorname{sgn}(\dot{x})}{m_c + m}
\end{aligned} \tag{7.3}$$

For each episode the simulation was run for a maximum of 120 simulated seconds and was terminated immediately if either the pole fell or the cart reached the edge of the track. The pole was considered to have fallen if $|\theta| > \pi/15$, and the cart was considered to have reached the edge of the track if $|x| > 2.4$. Every 0.02 simulated seconds the RL agent selected an action; the environment was updated using the Runge Kutta fourth order numerical integration method and then the reward was calculated.

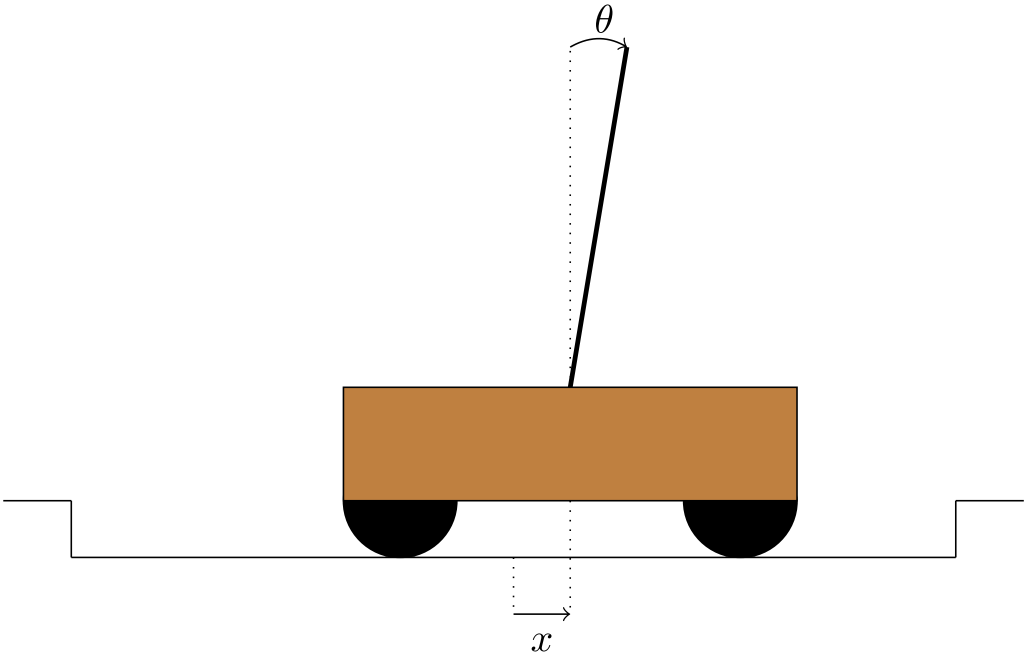


Figure 7.4: Diagram of the Cart-Pole problem.

Table 7.6: Cart-Pole Parameters

Parameter	Value
cart mass (m_c)	1
pole mass (m)	0.1
gravitational constant (g)	9.81
half pole length (l)	0.5
cart friction (μ_c)	5×10^{-4}
pole friction (μ_p)	2×10^{-6}
time increment (Δ_t)	0.02
maximum force (F_{max})	10N

The reward was zero at all time-steps until failure, i.e. the pole fell or the cart reached the edge of the track, at which time a reward of -1 was received.

The initial state at the beginning of each episode was $[0 + o, 0, 0, 0]^\top$, where o was uniformly randomly generated offset in the range $[-0.05, 0.05]$. The selected action was limited to the allowable range before being applied to the simulation.

7.2.3 Method

In all approaches Gaussian exploration was applied by adding a Gaussian random value with $\sigma^2 = 1$ to the action, this value was multiplied by an exploration coefficient which was reduced in order to reduce the exploration as training progressed.

The experiment was run three times: without noise; with uniform noise in the range

$[-F_{max}/2, F_{max}/2]$; and finally with Gaussian noise $\sim \mathcal{N}(0,1)$ added to selected action, this was done after the action had been limited to the allowable range therefore the actual applied action may not be within the action limits. The agent was not made aware of any noise applied.

All parameter values listed are those which were found to perform best in the noise free setting, parameters were not re-tuned when the experiments were re-run with noise added to the selected action.

CACLA Approach

CACLA was applied to this task for comparison against a state-of-the-art method as it previously outperformed a number of other continuous action-space RL methods on this task [56]. The approach was basically as [56] using a 12 hidden node MLP, with tanh activation at the hidden layer and linear output, for both the actor and critic. The parameters used are given in Table 7.7 on the next page, and the CACLA variance parameter $\beta = 0.001$. Momentum was not applied in updating of the MLPs in CACLA, as after some experimentation it was found not to produce improved results. Exploration was kept at 1 in training episodes as CACLA relies on exploration to update the policy function.

NM-SARSA Approach

The RL parameters used for NM-SARSA are listed in Table 7.7 on the following page with those of the other approaches. The NM-SARSA specific parameter values were: maximum iterations = 5; early convergence parameter $\zeta = 0.001$; initial action step-size $a_{\Delta} = 0.5$. The exploration coefficient was set to one at the start of each trial and reduced by 0.001 at each time-step until it reached zero.

NelderMead-SARSA Approach

The RL parameter values applied to the NelderMead-SARSA approach are presented Table 7.7 on the next page. Ten iterations of the Nelder Mead optimisation algorithm were performed for action selection. The exploration coefficient was set to 1 at the beginning of each trial and reduced by 0.001 at each time-step until it reached 0.

7.2.4 Results

Results were produced from 50 different runs of NM-SARSA, NelderMead-SARSA and CACLA. As soon as the agent succeeded in one of the episodes no further training took place; however, ten testing episodes were carried out with no training or exploration in order to test the performance and generalisation capability of the controller. The

Table 7.7: Cart-Pole RL Agent Parameters

Parameter	NM-SARSA	NelderMead-SARSA	CACLA
Hidden Nodes	12	12	12
RL step-size (α)	0.2	0.2	0.1
CACLA variance update (β)	N/A	N/A	0.001
RL discount rate (γ)	0.9	0.9	0.8
Learn rate	0.3	0.3	0.1
Momentum	0.7	0.75	0
Exploration	Gaussian	Gaussian	Gaussian

Table 7.8: Cart-Pole Results

Method	Noise	Success Rate	Episodes to Train			Testing
			Median	Min	Max	Success Rate
CACLA	None	100%	129	68	267	98%
	Gaussian	100%	136	76	314	92%
	Uniform	100%	149	84	269	90%
NM-SARSA	None	100%	98	52	215	98%
	Gaussian	100%	79	39	127	94%
	Uniform	100%	81	48	155	90%
NelderMead SARSA	None	100%	112	48	188	100%
	Gaussian	100%	124	52	218	96%
	Uniform	100%	134	43	293	84%

testing success rate is the percentage of runs in which the agent succeeded for all ten testing episodes.

Table 7.8 shows the success rate: percentage of runs in which the agent was able to successfully balance the pole in a training episode; median, minimum and maximum number of episodes taken to train the agent to balance the Cart-Pole for 120s; and also the testing success rate. The best results are highlighted in green and the worst in red. NM-SARSA and NelderMead-SARSA achieve comparable performance to CACLA, in terms of success rate, but NM-SARSA does so in consistently fewer training episodes, NelderMead-SARSA also trains in fewer episodes than CACLA although the difference is less extreme.

Fig. 7.5 on the next page was produced on the noise-free simulation, where 1000 episodes were run, regardless of whether the agent could balance the pole, in order to calculate the average balance time per episode averaged over the 50 runs. The average balance time of NM-SARSA is always slightly longer than that of CACLA. This also

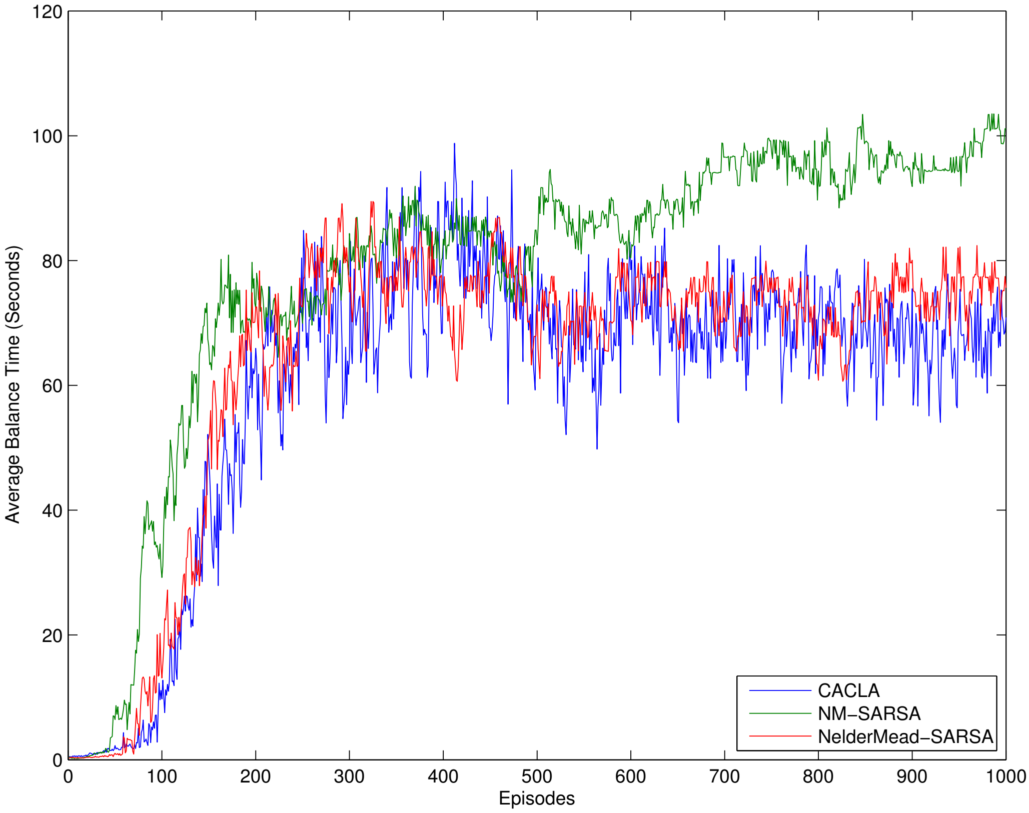


Figure 7.5: Cart-Pole training progress, balance time for each episode averaged over 50 runs.

shows the similarity in the training time of NelderMead-SARSA and CACLA. Although both CACLA and NelderMead-SARSA still experienced some runs with short balancing time in later episodes which produced the levelling off effect when the averages were calculated for the plot, NM-SARSA seemed to suffer less and less with short runs in later episodes: leading to the curve still increasing towards the later episodes. This may be due to the fact that Newton's Method when started from the same set of initial points will always return the same action, provided the value function isn't changed, therefore once exploration is reduced there is far less chance of taking a poor action. However, with NelderMead, the initial simplex is randomly initialised, thus returned actions may be different. This will result in non-deterministic action selection which may have resulted in some poorer actions being selected, even after training had reached a good level where most runs balanced for the full time by these episodes.

7.3 Double Cart-Pole

The double Cart-Pole problem is an extension of the standard Cart-Pole problem, whereby the cart has two poles, of different lengths, attached which both must be balanced [58]. The simulation in this experiment is the same as that of [55] except the maximum time of the simulation was 120s instead of 20s, making the task more challenging.

The state vector comprised the angle and angular velocity of each pole; cart distance from centre of track; and cart velocity $s = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2, x, \dot{x}]^\top$, and the action was the force applied to the cart $a = F \in [-40, 40]$.

7.3.1 Existing Approaches

The double pole variant of the well known Cart-pole benchmark has been used since [58] and has been used in several papers as a benchmark for EA based approaches [20, 22, 26, 27], which often also include a variant where the angular velocities are omitted from the state representation to make the task non-markovian; however, as we do not attempt to solve the problem of POMDPs here, we do not attempt this variant.

More recently, CACLA has been applied to this problem [55] achieving improved results compared to the approach in [26], in which the approach was shown to outperform NAC. Thus, we use CACLA as the state-of-the-art approach to compare our results to.

7.3.2 Details

The parameters used in this experiment are listed in Table 7.9 on page 87, and the equations of motion used to update the environment are (7.4), as was used in [55].

$$\begin{aligned} \ddot{x} &= \frac{F - \mu_c \operatorname{sgn}(\dot{x}) + \sum_{i=1}^{2} 2m_i\theta_i^2 \sin\theta_i + \frac{3}{4}m_i \cos\theta_i \left(2\frac{\mu_i\dot{\theta}_i}{m_i l_i} + g\sin\theta_i\right)}{m_c + \sum_{i=1}^{2} m_i \left(1 - \frac{3}{4}cos^2\theta_i\right)} \\ \ddot{\theta}_i &= -\frac{3}{8l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_i\dot{\theta}_i}{m_i l_i}\right) \end{aligned} \tag{7.4}$$

Each simulation was run for a maximum of 120 simulated seconds and was terminated immediately if either of the poles fell or the cart reached the edge of the track. A pole was considered to have fallen if $|\theta_i| > \pi/15$, and the cart was considered to have reached the edge if $|x| > 2.4$. Every 0.02s an action was selected by the RL agent, the environment was updated using the Runge Kutta fourth order method and the reward was calculated. The reward was -1 if a pole fell or the cart reached the edge of the

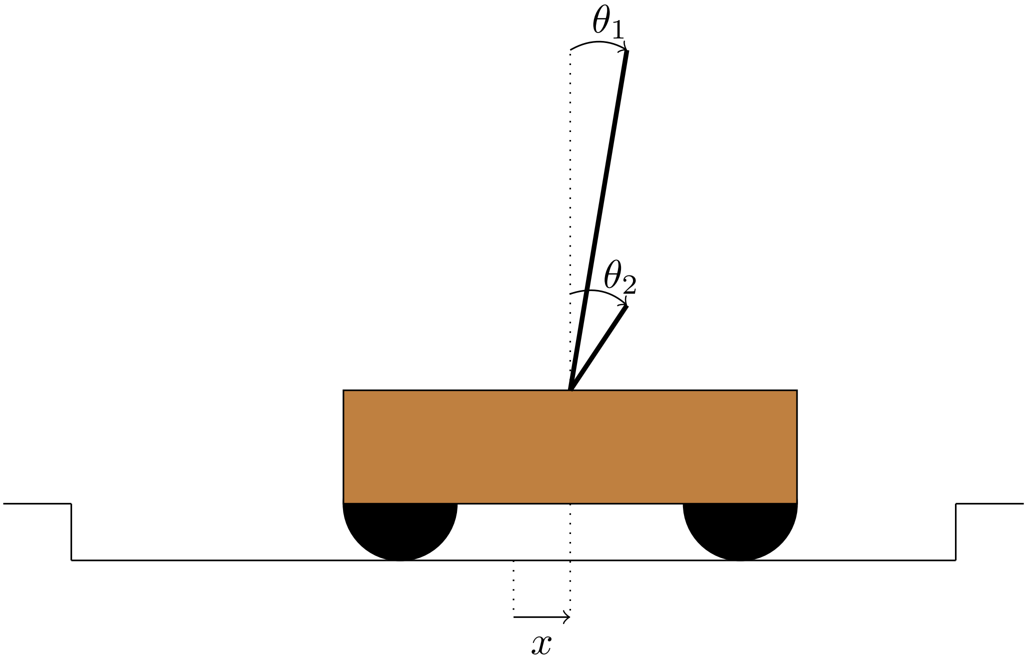


Figure 7.6: Diagram of the double Cart-Pole problem.

track and 0 otherwise.

The initial state at the beginning of each episode was $[\frac{\pi}{180}, 0, 0, 0, 0, 0]^\top$, this is the same as the initial state used in [55]. The selected action was limited to the allowable range before applying to the simulation.

7.3.3 Method

This application of the double Cart-Pole problem is slightly different to that described in [55] in that the length of time the agent was required to balance the poles for was 120s (rather than 20s) which makes the task more difficult. Also the magnitude of the exploratory actions in [55] were orders of magnitude larger than the allowable actions, thereby forcing the agent to learn a bang-bang controller. Here, as we are specifically interested in continuous actions, we do not influence the learnt policy through use of such large exploratory actions.

The experiment was run three times: without noise; with uniform noise in the range $[-F_{max}/2, F_{max}/2]$; and finally with Gaussian noise $\sim \mathcal{N}(0,1)$ added to selected action, this was done after the action had been limited to the allowable range therefore the actual applied action may not be within the action limits. The agent was not made aware of any noise applied.

All of the parameters used were found to perform the best of those tested on the noise-free setting, but were not re-tuned when noise was added.

Table 7.9: Double Cart-Pole Parameters

Parameter	Value
cart mass (m_c)	1kg
pole one mass (m_1)	0.1kg
pole two mass (m_2)	0.01kg
gravitational constant (g)	9.81
pole one length (l_1)	1m
pole two length (l_2)	0.1m
cart friction (μ_c)	5×10^{-4}
pole one friction (μ_1)	2×10^{-6}
pole two friction (μ_2)	2×10^{-6}
time increment (Δ_t)	0.02s
maximum force (F_{max})	40N

CACLA Approach

The CACLA algorithm was applied to the double Cart-Pole task for comparison purposes, as it was shown to outperform the results achieved using CMA-ES and NAC approaches [55] when compared to the results in [26]. In [55] the CACLA policy function approximator was limited to a linear architecture in order to make it directly comparable to linear methods; however, we apply non-linear MLP to the policy function in order to allow the approximation of more general functions.

The parameters used for the CACLA approach are given in Table 7.10 on the next page and the CACLA specific variance parameter used was $\beta = 0.001$. An exploration reduction schedule was not applied to CACLA, instead every tenth episode was run without exploration to determine whether learning had been achieved. This was found to perform far better than any exploration schedule tested with CACLA on this task.

NM-SARSA Approach

The parameters used for the NM-SARSA approach are listed in Table 7.10 on the following page, maximum iterations was 10; $a_\Delta = 0.5$; early convergence parameter $\zeta = 0.001$. Exploration was not used in this approach.

NelderMead-SARSA Approach

The RL parameters applied to this approach are presented alongside the other approaches in Table 7.10 on the next page. The number of NelderMead iterations used was 5. No exploration was used in this approach.

Table 7.10: Double Cart-Pole RL Agent Parameters

Parameter	NM-SARSA	NelderMead-SARSA	CACLA
Hidden nodes	12	12	12
Learn rate	0.2	0.2	0.1
RL step-size (α)	0.2	0.2	0.2
RL discount rate (γ)	0.9	0.9	0.9
Momentum	0.75	0.75	0
Exploration	N/A	N/A	Gaussian

Table 7.11: Double Cart-Pole Results

Method	Noise	Success Rate	Episodes to Train			Testing
			Median	Min	Max	Success Rate
CACLA	None	96%	430	20	970	N/A
	Gaussian	94%	430	180	970	90%
	Uniform	40%	720	290	980	32%
NM-SARSA	None	100%	89	2	464	N/A
	Gaussian	100%	106	50	639	72%
	Uniform	98%	131	49	602	72%
NelderMead SARSA	None	96%	141	28	482	80%
	Gaussian	98%	170	46	651	80%
	Uniform	94%	294	49	953	72%

7.3.4 Results

Each approach was allowed a maximum of 1000 episodes to train, but exploration was stopped after the agent successfully balances the poles and a further 10 testing episodes were performed without exploration or training, testing the generalisation capability of the learnt controller. Except for the noise-free setting where, as the initial state was the same for each episode, there would be no difference in the trajectory for any number of runs once training and exploration was stopped. This was run 50 times and the results are shown in table 7.11.

The performance achieved by NM-SARSA and NelderMead-SARSA was considerably better than that of CACLA in terms of success rate and training was achieved in considerably fewer episodes. CACLA achieved slightly better testing success rate on the Gaussian noise setting but considerably worse on the uniform noise setting.

The difference in performance of NM-SARSA and NelderMead-SARSA was not great on this task, however, NM-SARSA did produce consistently slightly better results

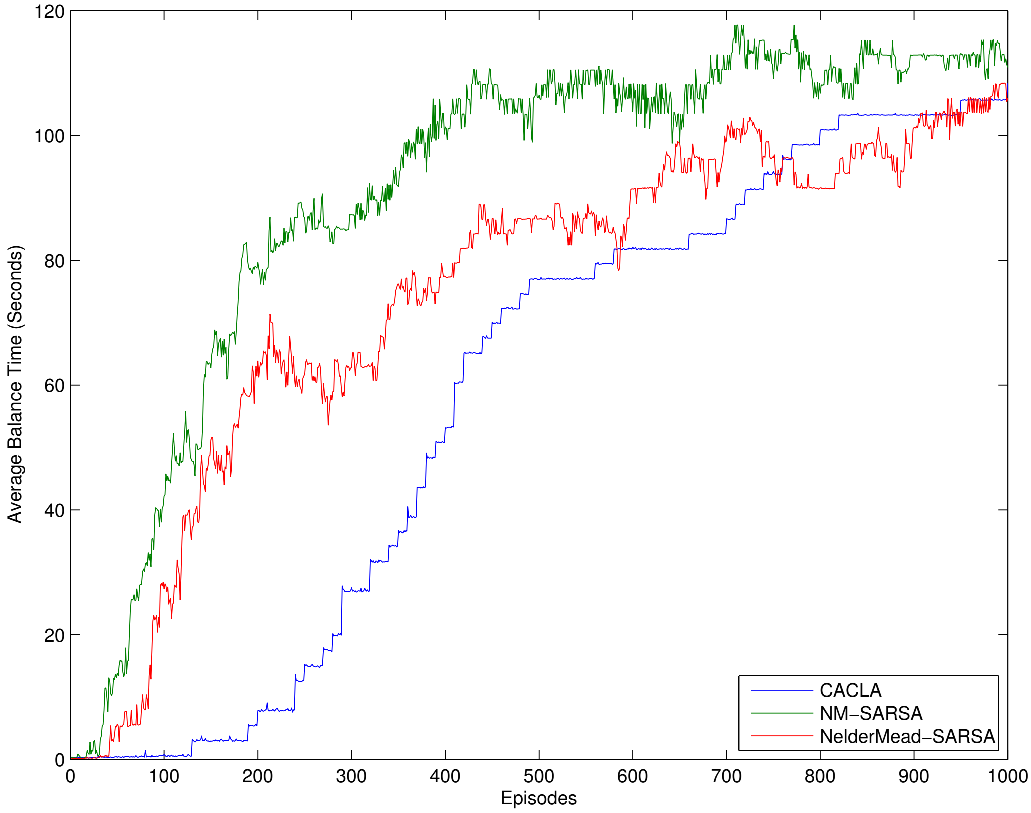


Figure 7.7: Double Cart-Pole training progress, balance time for each episode averaged over 50 runs.

in terms of average training episodes.

Fig. 7.7 was produced on the noise-free simulation, where 1000 episodes were run, regardless of whether the agent could balance the poles, in order to calculate the average balance time per episode averaged over the 50 runs. As there was no exploration reduction schedule used for CACLA, once CACLA succeeded in one episode the exploration coefficient was set to zero for all future episodes in that run to avoid the average time being close to zero due to exploratory actions. As can be seen in Fig. 7.7, NM-SARSA clearly trains in far fewer episodes than CACLA on this task, and, as with Table 7.11 on the previous page, shows that NelderMead-SARSA performed similarly with NM-SARSA but requiring a few more episodes to train.

7.4 Chapter Summary

In this chapter we have presented the results of applying the novel approaches to action selection in continuous action-space described in chapter 6 as well as existing approaches to continuous state- and action-space reinforcement learning problems. In all cases the novel approaches at least matched the performance of existing approaches, exceeding it in the Cart-Pole and double Cart-Pole problems.

Although the adaptive-critic, CACLA and GP approaches applied here are not novel algorithms and they have been applied to numerous RL control problems, these are the first applications of these approaches to the problem of the full swing-up of the two section acrobot that we are aware of. The results are also compared to results from the literature and the results presented here compare well. Also, the GP approach, along with NelderMead-SARSA, achieved the fastest swing-up time that we are aware of using these parameter settings for the standard two section acrobot; however, the GP approach is far more computationally intensive. This could be offset by calculating the fitness of all of the new generation in parallel rather than just the two children as was done here, but that would require far more processors as the population contained 2000 individuals. The implicit policy methods on the other hand could run faster on a standard computer without requiring any parallelization.

There are obviously trade-offs with the approaches presented: the action selection speed is slower when implicit policy methods are used, therefore, if very fast action selection is required the number of iterations used for the optimization algorithm may have to be reduced, potentially leading to worse results. But if DPS methods are used training may take much longer, including many more trials, which is very time consuming. Also, they cannot be updated online as implicit policy methods can be. AC methods may provide an acceptable balance of the two if online and faster training is required and the final policy must also be very fast to evaluate.

However, the time taken on action selection by the methods applied here was not long, both NM-SARSA and NelderMead-SARSA took, on average, 0.00003 seconds on the Cart-Pole problem; therefore, this approach should not be dismissed for all problems.

Chapter 8

Conclusion

8.1 Thesis Summary

This thesis investigated the problems associated with the application of reinforcement learning in the continuous state- and action-space investigated the use of several numerical optimization methods to directly solve the optimization problem:

$$\arg\max_{a} Q(s,a), \quad \forall a \in \mathcal{A}(s)$$

which is required for continuous action selection without learning an explicit policy function. These included gradient descent, which is mentioned as a possible approach in the literature [47]; Newton's Method; Nelder Mead and Genetic Algorithms. The best performing approaches: Newton's Method and Nelder Mead were then compared to a state-of-the-art continuous state- and action-space RL approach from the literature: CACLA on three continuous state- and action-space control benchmark problems from the literature: Acrobot swing-up, Cart-Pole and double Cart-Pole, on which the proposed methods showed improved performance. CACLA was selected as the state-of-the-art approach to verify the performance of these novel approaches against as it has been shown to outperform many alternative approaches on the Cart-Pole and double Cart-Pole problems in the literature, including wire-fitting; natural actor-critic and an evolutionary approach [55, 56]. We also produced new results of applying genetic programming to the full, two-section Acrobot swing-up task for the first time.

The results of the genetic programming approach to the minimum time full swing-up of the Acrobot was published in [12, 14]; and the results of applying Newton's Method to the action selection problem when applying implicit policy methods to the Cart-Pole and double Cart-Pole problems compared to the CACLA method were published in [40].

We began by introducing the essential background material in reinforcement learning (Chapter 2); artificial neural networks in Chapter 3; and evolutionary algorithms in Chapter 4. Then we expanded on the problems which arise from applying RL to problems with continuous state- and action-spaces in Chapter 5, and investigated the application of numerical optimization methods to the continuous action selection problem in Chapter 6. Finally, in Chapter 7 we empirically tested these approaches and compared them to state-of-the-art methods on the Acrobot; Cart-Pole; and double Cart-Pole problems, on all of which the proposed methods outperformed the state-of-the-art methods they were compared to.

Aside from presenting these approaches to action selection in the continuous action-space and showing empirically their performance compares favourably to existing methods, in applying existing methods to the benchmarks, we present novel results for some existing approaches on the Acrobot problem. The most interesting of which is the GP approach which we show to outperform existing approaches, even though the GP results were almost matched by the implicit policy methods we propose in far less time and with far less parameter tuning.

8.2 Discussion

The major contributions to knowledge of this thesis are the two algorithms resulting from the application of numerical optimization methods: NM-SARSA and NelderMead-SARSA which perform the continuous action selection far quicker than gradient descent, and without requiring the discretization of the action space. Also novel results are included for the existing methods of GP, CACLA and Adaptive-Critic on the Acrobot problem in order to compare the novel approaches to existing methods. Results were also produced from the application of the CACLA method to the Cart-Pole and a slightly more difficult version of the double Cart-Pole, where the required balance time was significantly longer than that of the approach from the literature [55], in order to directly compare it to the methods proposed.

The optimization methods applied in these two approaches were selected after some preliminary experimentation on the Cart-Pole problem, where they considerably outperformed the alternatives considered: GA and gradient ascent both in terms of testing performance and action selection time.

The results from the Acrobot experiments showed that the proposed methods were able to swing-up the acrobot in a faster time than those from the literature [11, 50], and produced comparable performance with a method from the literature which allowed much larger torque values [36]. Whilst the two actor-critic methods applied produced similar swing-up times to existing methods from the literature, but the genetic programming approach produced slightly faster swing-up time than the proposed methods.

The GP approach however took by far the longest to run and required the most parameter tuning of all methods applied. However, the two IPM approaches were the only to have any unsuccessful runs on the Acrobot problem, but it is believed that this could be corrected with either more parameter tuning or by including a test that the neural networks weights have not become stuck, as is applied with the Adaptive-Critic approach.

On both the Cart-Pole and double Cart-Pole the NM-SARSA approach performed best, on average, in terms of success rate and also required far fewer episodes to train than the CACLA approach, with the NelderMead-SARSA approach producing slightly worse results than NM-SARSA, but still outperforming CACLA, and even in some cases outperforming NM-SARSA.

The proposed approaches enable the application of RL to continuous state- and action-space problems without a separate policy function, as is used in AC methods, and without discretizing the action-space which is applied in some other approaches. Both methods achieved good performance on the benchmark problems they were applied to with very little parameter tuning. The action selection times were much faster than by applying gradient ascent, which is sometimes offered as a possible approach in the literature [47]. However, the action selection time is still slower than that of a dedicated policy function due to the time spent solving the optimization problem, and therefore, if the action must be selected faster than is possible with these methods it may still be necessary to apply actor-critic methods.

8.3 Future Work

Although the methods achieved good results on the problems they were applied to here, it is important that they are applied to other benchmarks as well. It would be particularly interesting to observe their performance on problems with higher dimensional action-spaces, for which it may be necessary to adjust the NM-SARSA approach to use a quasi-Newton method in order to speed up the process and maintain the positive definiteness of the Hessian. However, the NelderMead-SARSA method should be immediately applicable to higher dimensional action-spaces without any adjustments.

It would also be interesting to compare alternative optimization algorithms, such as approaches which can take advantage of the benefits of both derivative-free and derivative-based optimization, as well as extensions to the optimization methods utilised here, which may further speed up action selection and increase the applicability of these approaches to problems where action selection must be very fast.

Also, as the algorithms presented here are the basic versions without eligibility traces or extensions and variations to the function approximation or optimization methods used, such as variable learning rates [18]; adaptive activation functions [18]; alternative structures: additional hidden layers, bridged MLP, fully connected cascade [60], etc.; adjusting the line-search step length [41]. Future work should investigate what performance improvements can be made by applying such enhancements.

References

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

[2] Leemon C. Baird and Harry Klopf. Reinforcement learning with high-dimensional, continuous actions. Technical report, Wright Laboratory, 1993.

[3] Andrew G. Barto, Richard S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, 13(5):834–846, September 1983.

[4] Jonathan Baxter and Peter L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, National University, 1999.

[5] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 6 edition, 1972.

[6] Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actori–critic algorithms. *Automatica*, 45(11):2471–2482, 2009.

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

[8] Gary Boone. Minimum-time control of the acrobot. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 4, pages 3281–3287, April 1997.

[9] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, 2004.

[10] Lucian Buşoniu, Robert Babuška, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Boca Raton, Florida, 2010.

[11] Rémi Coulom. High-accuracy value-function approximation with neural networks. In *European Symposium on Artificial Neural Networks*, 2004.

[12] Dimitris Dracopoulos, Dimitrios Effraimidis, and Barry D. Nichols. Genetic programming as a solver to challenging reinforcement learning problems. volume 8 of *Horizons in Computer Science Research*, pages 145–174. Nova Publications, Hauppauge, NY, USA, 2013.

[13] Dimitris C. Dracopoulos and Dimitrios Effraimidis. Genetic programming for generalised helicopter hovering control. In Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta, editors, *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 25–36, Malaga, Spain, April 2012. Springer Verlag.

[14] Dimitris C. Dracopoulos and Barry D. Nichols. Swing up and balance control of the acrobot solved by genetic programming. In Max Bramer and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXIX*, pages 229–242. Springer London, 2012.

[15] Sam Duong, Hiroshi Kinjo, Eiho Uezato, and Tetsuhiko Yamamoto. On the continuous control of the acrobot via computational intelligence. In Been-Chian Chien, Tzung-Pei Hong, Shyi-Ming Chen, and Moonis Ali, editors, *Next-Generation Applied Intelligence*, volume 5579 of *Lecture Notes in Computer Science*, pages 231–241. Springer Berlin / Heidelberg, 2009.

[16] Sam Chau Duong, Hiroshi Kinjo, Eiho Uezato, and Tetsuhiko Yamamoto. A switch controller design for the acrobot using neural network and genetic algorithm. In *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pages 1540–1544, December 2008.

[17] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[18] Laurene Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms and Applications.* Prentice Hall, New Jersey, 1994.

[19] Ryo Fukushima and Eiho Uezato. Swing-up control of a 3-dof acrobot using an evolutionary approach. *Artificial Life and Robotics*, 14:160–163, 2009.

[20] Faustino J. Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361, 1999.

[21] Ivo Grondman, Lucian Buşoniu, Gabriel A. D. Lopes, and Robert Babuška. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1291–1307, 2012.

[22] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 81–89. MIT Press, 1996.

[23] José Antonio Martín H. and Javier de Lope. Ex<a>: An effective algorithm for continuous actions reinforcement learning problems. In *Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE*, pages 2063–2068, November 2009.

[24] José Antonio Martín H., Javier de Lope, and Darío Maravall. The k-nn-td reinforcement learning algorithm. In José Mira, José FerrÃąndez, José Álvarez, Félix de la Paz, and F. Toledo, editors, *Methods and Models in Artificial and Natural Computation. A Homage to Professor MiraâĂŹs Scientific Legacy*, volume 5601 of *Lecture Notes in Computer Science*, pages 305–314. Springer Berlin / Heidelberg, 2009.

[25] Simon Haykin. *Neural Networks and Learning Machines (3rd Edition)*. Prentice Hall, 3 edition, November 2009.

[26] Verena Heidrich-Meisner and Christian Igel. Evolution strategies for direct policy search. In *Parallel Problem Solving from Nature–PPSN X*, pages 428–437. Springer, 2008.

[27] Verena Heidrich-Meisner and Christian Igel. Neuroevolution strategies for episodic reinforcement learning. *Journal of Algorithms*, 64(4):152–168, 2009.

[28] Verena Heidrich-Meisner, Martin Lauer, Christian Igel, and Martin A Riedmiller. Reinforcement learning in a nutshell. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, pages 277–288, 2007.

[29] Tobias Jung, Daniel Polani, and Peter Stone. Empowerment for continuous agent-environment systems. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 19:16–39, February 2011.

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

[31] Sham Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems 14*. MIT Press, 2002.

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

[33] Matthew R. Kretchmar and Charles W. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Neural Networks, 1997., International Conference on*, volume 2, pages 834–837. IEEE, 1997.

[34] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the nelder-mead simplex method in low dimensions. *SIAM Journal of Optimization*, 9:112–147, 1998.

[35] Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

[36] Xu-Zhi Lai, Jin-Hua She, Simon X. Yang, and Min Wu. Comprehensive unified control strategy for underactuated two-link manipulators. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(2):389–398, April 2009.

[37] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[38] John A. Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[39] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer, 2006.

[40] Barry D. Nichols and Dimitris C. Dracopoulos. Application of Newton's method to action selection in continuous state- and action-space reinforcement learning. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, pages 141–146, April 2014.

[41] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, August 2000.

[42] Jason Pazis and Michail G. Lagoudakis. Learning continuous-action control policies. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL '09. IEEE Symposium on*, pages 169–176, April 2009.

[43] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008.

[44] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A field guide to genetic programming.* Lulu. com, 2008.

[45] Warren B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality.* John Wiley & Sons, Hoboken, New Jersey, 2 edition, 2011.

[46] Martin Riedmiller, Jan Peters, and Stefan Schaal. Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 254–261, April 2007.

[47] Juan C. Santamarí, Richard S. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive behavior*, 6(2):163–217, 1997.

[48] Jennie Si and Yu-Tsung Wang. Online learning control by association and reinforcement. *Neural Networks, IEEE Transactions on*, 12(2):264–276, March 2001.

[49] Matthijs T.J. Spaan. Partially observable markov decision processes. In Marco Wiering and Martijn Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 387–414. Springer Berlin Heidelberg, 2012.

[50] Mark W. Spong. Swing up control of the acrobot. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, volume 3, pages 2356–2361, May 1994.

[51] Mark W. Spong. The swing up control problem for the acrobot. *Control Systems, IEEE*, 15(1):49–55, February 1995.

[52] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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

[54] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38:58–68, March 1995.

[55] Hado van Hasselt. Reinforcement learning in continuous state and action spaces. In Marco Wiering and Martijn Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 207–251. Springer Berlin Heidelberg, 2012.

[56] Hado van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 272–279, April 2007.

[57] Lex Weaver and Jonathan Baxter. Reinforcement learning from state and temporal differences. Technical report, Department of Computer Science, Australian National University, 1999.

[58] Alexis P. Wieland. Evolving neural network controllers for unstable systems. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 667–673. IEEE, 1991.

[59] Lukasz Wiklendt, Stephan Chalup, and Rick Middleton. A small spiking neural network with lqr control applied to the acrobot. *Neural Computing & Applications*, 18:369–375, 2008.

[60] Bogdan M. Wilamowski. Neural network architectures and learning algorithms. *Industrial Electronics Magazine, IEEE*, 3(4):56–63, December 2009.

[61] S. S. Willson, Philippe Mullhaupt, and Dominique Bonvin. Quotient method for controlling the acrobot. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pages 1770–1775, December 2009.

[62] Xin Xu, Dewen Hu, and Xicheng Lu. Kernel-based least squares policy iteration for reinforcement learning. *Neural Networks, IEEE Transactions on*, 18(4):973–992, July 2007.

[63] Junichiro Yoshimoto, Masaya Nishimura, Yoichi Tokita, and Shin Ishii. Acrobot control by learning the switching of multiple controllers. *Artificial Life and Robotics*, 9:67–71, 2005.

List of Publications

[BDN1] Barry D. Nichols and Dimitris C. Dracopoulos. Application of Newton's method to action selection in continuous state- and action-space reinforcement learning. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, pages 141–146, April 2014.

[BDN2] Dimitris C. Dracopoulos and Barry D. Nichols. Swing up and balance control of the acrobot solved by genetic programming. In Max Bramer and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXIX*, pages 229–242. Springer London 2012.

[BDN3] Dimitris C. Dracopoulos, Dimitrios Effraimidis, and Barry D. Nichols. Genetic programming as a solver to challenging reinforcement learning problems. volume 8 of *Horizons in Computer Science Research*, pages 145–174. Nova Publications, Hauppauge, NY, USA, 2013.