A Comparison of Action Selection Methods for Implicit Policy Method Reinforcement Learning in Continuous Action-Space

Barry D. Nichols
School of Science and Technology
Middlesex University
London, England
Email: b.nichols@mdx.ac.uk

***Abstract*—In this paper I investigate methods of applying reinforcement learning to continuous state- and action-space problems without a policy function. I compare the performance of four methods, one of which is the discretisation of the action-space, and the other three are optimisation techniques applied to finding the greedy action without discretisation. The optimisation methods I apply are gradient descent, Nelder-Mead and Newton's Method. The action selection methods are applied in conjunction with the SARSA algorithm, with a multilayer perceptron utilized for the approximation of the value function. The approaches are applied to two simulated continuous state- and action-space control problems: Cart-Pole and double Cart-Pole. The results are compared both in terms of action selection time and the number of trials required to train on the benchmark problems.**

I. Introduction

Reinforcement learning (RL) [1], [2] has many successes in discrete and small action-space problems. However, many interesting problems have large and/or continuous action-space, for which it is not obvious which RL approach is the most appropriate.

The most common methods utilised in the application of RL to continuous action-space problems are either actor-critic methods (AC) [3] or direct policy search (DPS) [4]. AC requires the approximation of two functions: the value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, giving the expected long term reward from being in a given state $s \in \mathcal{S}$ and taking an action $a \in \mathcal{A}$, and the policy function $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which is a mapping from states to actions. It is also common, when using AC methods, to use the value function $V : \mathcal{S} \rightarrow \mathbb{R}$, which is the expected long-term, discounted reward when in state $s \in \mathcal{S}$ and following $\pi(s)$. DPS on the other hand seek to directly optimise the policy function $\pi(s)$ taking the value of the policy to be the sum of discounted rewards received when following that policy.

Another method, which is the method I investigate here, is the implicit policy method (IPM), which approximates the value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$; however, unlike AC methods, no policy function is stored. Instead the action to take from a given state is calculated as required from the Q function. Despite being the method of choice when the action-space is small or discrete [3], IPM is not frequently applied when the action-space is large or continuous. This is due to the fact that it becomes impossible to compare the values of every possible action, and it has been stated that applying optimisation to the action selection at every time-step would be prohibitively time consuming [3], [5]. However, there have been some examples of the successful use of IPM in continuous action-space [6]–[8].

Here I investigate IPM using several different action selection methods: discretisation, gradient descent, Newton's Method and Nelder-Mead. I compare these methods both in terms of performance and action selection time on two well-known continuous state- and action-space control benchmark problems from the literature: the Cart-Pole problem and related double Cart-Pole problem. In these experiments, apart from the action selection method used, all parameter values are kept constant. This allows a clear comparison of the performance of the action selection methods, both in terms of speed: action selection time and accuracy: reflected by the number of training trials required by each approach.

The rest of the paper is organised as follows. Firstly, in Section II, I give a brief summary of RL focussing on the approach used here. Then, in Section III, I present the details of the action selection methods which will be compared. In Section IV I specify the details of the neural network I use to approximate $Q(s, a)$, and the derivation of $\nabla_a Q(s, a)$ and $\nabla_a^2 Q(s, a)$, which are required to apply the derivative based action selection methods. Sections V and VI give the details of the experiments conducted on the Cart-Pole and double Cart-Pole domains respectively, including results obtained. Finally I present my conclusions and compare the performance of the approaches across the different problem domains in Section VII. In the conclusion I also give some indication of problems which may arise when applying these approaches with possible solutions and some directions for future work.

II. Background

An RL agent learns to perform a task by experimenting on that task and receiving feedback based on its current performance. At any given time-step, the agent is in state $s \in \mathcal{S}$ and must select an action $a \in \mathcal{A}$. After applying

the selected action a to the environment, the agent receives a numerical reward $r \in \mathbb{R}$ from the environment and transitions to the resulting state $s' \in \mathcal{S}$. The reward could be negative in order to represent a punishment for reaching undesirable states, or, in the case of delayed rewards, the reward may be zero until the terminal state. The delayed reward case is suitable for several tasks, e.g. a board game where the reward is based on the final outcome of the game, or a disaster avoidance task where we simply attempt to avoid failure. An example of disaster avoidance is the Cart-Pole problem used in this paper, described in more detail in Section V.

In order to improve its performance on the task, the RL agent seeks to maximise the rewards it receives, or to be precise the sum of long-term discounted rewards (1), where t is the time-step; r_{t+1} is the reward received in time-step $t+1$, after taking a_t from s_t, and $\gamma \in (0, 1]$ is the discount rate. Normally $\gamma < 1$ to ensure the sum of rewards remains finite and to assign more weight to rewards which will be received in the short-term.

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

The optimisation of the sum of discounted rewards is achieved by the construction of a value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, which is an approximation of the expected sum of long term rewards when taking the given action from the given state and then following the learned policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The learned policy is a mapping from states to the action to take when in that state. The Q function is defined as (2), and can be recursively defined as (3), which is the sum of the immediate reward and the expected sum of rewards which will be received resulting from taking the next action from the next state.

$$Q(s, a) = \mathbb{E}\left\{ \sum_{k=0}^{T-1} \gamma^k r(s_{k+1}, a_{k+1}) \Big| s_0 = s, a_0 = a \right\} \tag{2}$$

$$Q(s_t, a_t) = r + \gamma Q(s_{t+1}, a_{t+1}) \tag{3}$$

The learnt policy involves selecting the greedy action w.r.t. the value function. In small, discrete action-space the agent can select the greedy action by evaluating the value function at every possible action (4).

$$\arg\max_a Q(s, a), \quad \forall a \in \mathcal{A} \tag{4}$$

The agent must also perform exploratory actions in order to update the value function in unexplored areas of the state- and action-space, which may lead to an improved policy. There are several exploration methods commonly applied, in the discrete action-space this may involve selecting a random action with a small probability or selecting actions proportionately to their value. These methods are known as ϵ-greedy and Softmax exploration respectively [1].

The approximation of the expected sum of rewards $Q(s, a)$ can be updated on-line by temporal difference methods, in which the TD error δ (5) is minimised. This TD error is then used to update the values of $Q(s, a)$ (6), where α is a step-size parameter.

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{5}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \delta_t \tag{6}$$

A. Continuous State- and Action-Space

When the state-space is continuous it is not possible to apply a lookup table to store the Q values, as it is in the small, discrete state- and action-space setting. Therefore function approximation is often applied to approximate the Q function. However, when the action space is continuous it is also impossible to compare the values of each action from the current state; thus, other methods must be employed.

The approximation of the Q function could be achieved by an artificial neural network (ANN). Many types of ANN have been successfully applied to the approximation of the value function, e.g. CMAC, RBF, perceptron, MLP. Here, I focus my presentation on the multilayer perceptron (MLP) as this is the ANN I apply in this work. The MLP is a global function approximator capable of approximating any function to arbitrary accuracy, provided sufficient neurons are included in the hidden layer [9].

When an ANN is employed to approximate the value function, the update is then applied to the parameters of the ANN (the synaptic weights) rather than directly updating the value associated with the specific state and action. The update equation is then (7), where $\vec{\theta}$ is the parameter vector for the ANN.

$$\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta_t \nabla_{\vec{\theta}} Q(s_t, a_t) \tag{7}$$

In the continuous action-space setting exploration is often applied through Gaussian exploration [10]. In Gaussian exploration a Gaussian random variable is added to the greedy action before it is applied to the environment. This is more appropriate than ϵ-greedy and Softmax when the action-space is continuous.

When the action-space is continuous some researchers believe it is infeasible to directly solve the optimisation problem (4) at every time-step [3], [5]. Therefore, separate function approximators are often applied to approximating the policy function. This approach, which employs both the Q function and the π function, is the actor-critic method [5].

An alternative is to attempt to optimise the policy function without learning the value function. This is achieved by taking the value of the current policy to be the J function, which is defined as the sum of discounted rewards received in an episode when following the current policy (1). As this function is dependant only on the policy it is sometimes written as $J(\pi)$ or $J(\vec{\theta})$, where $\vec{\theta}$ are the parameters of the approximation of the policy function. These parameters are then updated in order to maximise the J function. This is direct policy search, which includes policy gradient [3] and evolutionary algorithm approaches [11], [12].

Another approach is to attempt to directly optimise the value function w.r.t. the action, in order to find the action

which is expected to lead to the highest sum of rewards. This optimisation must be performed at every time-step in order to select actions to take. Despite the fact that it has been suggested that directly solving this optimisation problem at every time-step is infeasible [3], [5], this is often the suggested approach if the action space is small and discrete [3]. Moreover, there are some examples of such approaches being successfully applied in the literature [7], [8], [13], where they are also compared to an AC method. I refer to this approach as implicit policy method (IPM), and it is this approach I focus on in this work. This approach does not require an explicit policy function as the action selection becomes a case of selecting the $a \in \mathcal{A}$ which maximises $Q(s, a)$. However, it is unclear which method should be applied to selecting this action. Some researchers have suggested the application of gradient descent [2], [8], whilst others discretise the action space in order to allow comparison of the values of all actions in this reduced action-space [8]. Alternative optimisation methods have been applied such as Newton's Method [7] and Nelder-Mead [13]. Each of these action selection methods are described in greater detail in Section III.

III. Action Selection Methods

In the following I describe the details of the methods used, which are discretisation, gradient descent, Nelder-Mead and Newton's Method. Discretisation of the action space has been applied in [8], gradient based optimisation has been suggested in [2], [8] and applied in [6], Newton's Method has been applied in [7], and Nelder-Mead has been utilised in [13]. These descriptions are in terms of scalar actions as the following experiments use scalar actions; however, all approaches described here could be adapted to higher dimensional action-spaces.

A. Discretisation

The simplest approach is to discretise the action-space and to select actions from this new, smaller action-space $\mathcal{A}_d = \{a_{min}, a_{min} + a_\Delta, \ldots, a_{max} - a_\Delta, a_{max}\}$, where a_{min} and a_{max} are the minimum and maximum action values respectively. This approach is described in [8].

This method allows the selection of actions from a reduced action-space which spans the continuous action-space with a granularity determined by the parameter a_Δ. The issues with this method are that it is difficult to know the granularity required for a given problem in advance, and the smaller a_Δ is the larger the cardinality of $\mathcal{A}_d$ will be. Therefore there is a trade off between coarseness of action-space and time taken to evaluate all actions in $\mathcal{A}_d$. On the other hand this algorithm benefits from its simplicity and that it has only one tunable parameter which must be set: a_Δ.

B. Gradient Descent

A more sophisticated approach is to apply the gradient descent algorithm [14], which utilises the first partial derivative of the value function w.r.t. the action to find a search direction. The only additional requirement when applying gradient descent is the first derivative of the value function, from which small updates are iteratively made to the action in the direction of the gradient (8), where β is a small positive step-size parameter.

$$a \leftarrow a - \beta \nabla_a Q(s, a) \tag{8}$$

Although gradient decent requires little computation, due to relying only on the first derivative of the value function for the search direction, it can be excruciatingly slow depending on the problem and is also highly susceptible to becoming trapped in local optima [14]. To mitigate the fact that gradient descent is susceptible to converging to local minima I restart this approach from several initial points. The search is run from each initial action in the set $\{a_{min}, a_{min} + a_\Delta, \ldots, a_{max} - a_\Delta, a_{max}\}$.

To allow the algorithm to terminate before the maximum number of iterations is reached, and therefore speed up the action selection process, an early convergence detection parameter ζ was used. The algorithm was terminated if the action at a given iteration is not sufficiently different from the action at the previous iteration, where ζ is the value used to determine if the change is sufficient using: $|a_k - a_{k-1}| < \zeta$.

This approach has considerably more parameters to tune than discretisation: maximum iterations η, step-size β, initial action step-size parameter a_Δ and an early convergence parameter ζ.

C. Nelder-Mead

An alternative method which does not rely on the derivative is Nelder-Mead [15]. This method utilises a simplex approach whereby when optimising points in an n-dimensional space, a simplex of $n+1$ points in n-dimensional space are maintained.

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. There are four parameters: ρ, χ, γ and σ, which affect reflection, expansion, contraction and shrinkage respectively. Here I apply the same values of these parameters as [16]: $\rho = 1, \chi = 2, \gamma = 0.5, \sigma = 0.5$. Other values were experimented with but none produced improved results. Therefore the only parameter to be tuned is the number of iterations of the Nelder-Mead algorithm to apply η.

As Nelder-Mead is a derivative-free method it does not require the calculation of the derivative of the value function, and thus can be applied in conjunction with any function approximation method. Also, the action selection can be very quick depending only on the number of iterations performed and the speed of calculating $Q(s, a)$, which is very fast with most function approximation techniques.

D. Newton's Method

Newton's Method [14] is a widely applied derivative-based approach, which utilises both the first and the second derivative of the objective function when calculating the search direction. By also utilising the second derivative Newton's Method can converge much faster than if only the gradient were used [14], [17].

Originally applied as an iterative approach to finding zeros of a function, Newton's Method is also widely utilised as an optimisation method by applying it to finding values where the first derivative is zero. In this case the first and second partial derivatives of $Q(s,a)$ w.r.t. a are used (9).

$$a \leftarrow a - \frac{\nabla_a Q(s,a)}{\nabla_a^2 Q(s,a)} \tag{9}$$

As this algorithm converges to points where the first derivative is zero, which may be local minima as well as local maxima, it is essential that this algorithm is repeated from several initial points in order to find the maximum.

As this method relies on the availability of the first and second partial derivatives of the value function it may not be applicable to all function approximation techniques. However when a MLP is used to approximate $Q(s,a)$ the partial derivatives can be calculated quickly based on the structure of the ANN.

There are additional considerations when Newton's Method is applied to optimizing an objective function with 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 [17].

In this approach, as with gradient descent, the algorithm was restarted from several different initial actions, thus an initial action step-size parameter a_Δ was applied to determine the initial actions. Also, as with gradient descent an early convergence parameter ζ and the number of iterations η have to be set. However, unlike gradient descent no step-size parameter is required. This results in the following three tunable parameters: maximum iterations η, initial action step-size a_Δ and early convergence parameter ζ.

IV. Artificial Neural Network

Here I provide details of the ANN architecture used to approximate $Q(s,a)$ in the following experiments. I will then present the derivation of the first and second partial derivatives of the value function w.r.t. a, which are required when applying gradient descent and Newton's Method to action selection.

A. Architecture

The ANN architecture used in these experiments was an MLP with one hidden layer. The activation function at the hidden nodes was hyperbolic tangent and linear at the output layer. The calculation of the output is shown in Equations (10), where $O(\vec{x})$ is the output of the ANN when presented with input vector $\vec{x}$ (comprising s and a); N and M are the number of nodes at the input and hidden layers respectively; $v_{i,j}$ is the synaptic weight between input i and hidden node j; w_j is the weight from hidden node j to the output node; h_out_j and h_in_j are the input to and the output from hidden node j; b_hid_j and b_out are the bias weights to the hidden and output nodes.

$$\begin{aligned} O(\vec{x}) &= b\_out + \sum_{j=1}^{M} w_j h\_out_j \\ h\_out_j &= \tanh\left(h\_in_j\right) \\ h\_in_j &= b\_hid_j + \sum_{i=1}^{N} v_{i,j} x_i \end{aligned} \tag{10}$$

B. Derivation of Partial Derivatives

The Gradient and Newton's Method action selection methods require $\nabla_a Q(s,a)$ and Newton's Method also requires $\nabla_a^2 Q(s,a)$. Both of which can be calculated analytically based on the equations of the MLPs output (10). The equations for $\nabla_a Q(s,a)$ and $\nabla_a^2 Q(s,a)$ are derived in Equation (11) and Equation (12) respectively.

$$\begin{aligned} \nabla_a Q(s,a) = \frac{\partial O(\vec{x})}{\partial x_N} &= \sum_{j=1}^{M} \frac{\partial O}{\partial h\_out_j} \frac{\partial h\_out_j}{\partial h\_in_j} \frac{\partial h\_in_j}{\partial x_N} \\ &= \sum_{j=1}^{M} w_j \tanh'\left(h\_in_j\right) v_{N,j} \end{aligned} \tag{11}$$

where $\tanh'(x) = 1 - \tanh^2(x)$.

$$\begin{aligned} \nabla_a^2 Q(s,a) &= \frac{\partial}{\partial x_N} \left[\sum_{j=1}^{M} w_j \tanh'\left(h\_in_j\right) v_{N,j} \right] \\ &= \sum_{j=1}^{M} w_j \frac{\partial \tanh'\left(h\_in_j\right)}{\partial h\_in_j} \frac{\partial h\_in_j}{\partial x_N} v_{N,j} \\ &= \sum_{j=1}^{M} w_j \tanh''\left(h\_in_j\right) v_{N,j}^2 \end{aligned} \tag{12}$$

where $\tanh''(x) = -2\tanh(x)\tanh'(x)$.

V. Cart-Pole

The Cart-Pole problem is the control of a cart on a limited track which has a pole attached to it by a free joint. The pole must be maintained in the balanced position by applying force to the cart, without the cart reaching the edge of the track (Fig.1).

The state vector comprises 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 is the force applied to the cart $a = F \in [-10, 10]\,\mathrm{N}$.

This problem is a very well known control benchmark problem often used by the RL community [1], [4], [10], [18], [19]. Commonly actions are limited to discrete values [19], [20], rather than the continuous range permitted here.

A. Description

The standard Cart-Pole problem, is a widely used control benchmark problem [10], [19]. In many applications the actions are limited to $\mathcal{A} = \{0, \pm 10\}\,\mathrm{N}$; here, however, continuous actions $\mathcal{A} = [-10, 10]\,\mathrm{N}$ are permitted in order to evaluate the performance of the algorithms in the continuous action-space. The parameters of the environment used in this

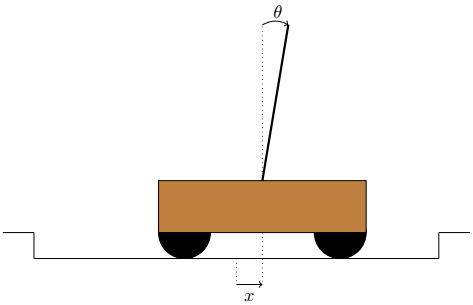

Fig. 1. Diagram of the Cart-Pole problem. The cart is on a limited track with a pole attached to it, but free to rotate. The angle of the pole from the balance position is θ and x is the position of the cart from the centre of the track.

TABLE I
CART-POLE PARAMETERS

Parameter	Value
Cart mass (m_c)	1 kg
Pole mass (m)	0.1 kg
Gravitational constant (g)	9.81 m/s^2
Half pole length (l)	0.5 m
Cart friction (μ_c)	5×10^{-4}
Pole friction (μ_p)	2×10^{-6}
Time increment (Δ_t)	0.02 s
Maximum force (F_{max})	10 N

experiment are listed in Table I, and the equations of motion used to update the environment are specified in (13), which are the same as those used in [19].

$$\begin{aligned} \phi &= -F - ml\dot{\theta}^2 \sin\theta + \mu_c \operatorname{sgn}(\dot{x}) \\ \ddot{\theta} &= \frac{g\sin\theta + \phi\cos\theta - \frac{\mu_p\dot{\theta}}{ml}}{l\left(\frac{4}{3} - \frac{m\cos^2\theta}{m_c+m}\right)} \\ \ddot{x} &= \frac{F + ml\left(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right) - \mu_c\operatorname{sgn}(\dot{x})}{m_c+m} \end{aligned} \tag{13}$$

For each episode the simulation was run for a maximum simulation time of 120 s 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 s 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. 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 a uniformly randomly generated offset in the range $[-0.05, 0.05]$. The selected action, including any exploration, was limited to the allowable range before being applied to the simulation.

TABLE II
CART-POLE REINFORCEMENT LEARNING PARAMETERS

Parameter	Value
RL discount rate (γ)	0.9
RL step-size (α)	0.2
ANN hidden node quantity	7
ANN initial weight range	[-0.2,0.2]
ANN learning rate	0.3
ANN momentum	0.75

B. Method

The RL algorithm applied to the updating of the value function was SARSA [1]. When running these experiments the RL parameters, including the parameters and architecture of the ANN used to approximate the value function, were kept constant for all four action selection methods. The values of these parameters are listed in Table II.

Gaussian exploration was applied by adding a random value, from the normal distribution, to the selected action. The exploration schedule was implemented by multiplying the random exploration value by an exploration parameter before adding it to the selected action. This exploration parameter was initially set to 1, but was reduced at each time-step by 0.001 until it reached 0. The exploration parameter was reset to 1 at the start of each trial.

When running the experiment with different action selection methods, only the action selection method was changed. Each of the action selection methods applied have some parameters to be tuned, the details of the parameter values used for each method are listed below. These values were found to produce the best results on this problem after some experimentation with different values.

The parameters which are specific to each of the action selection methods and their values for this experiment were as follows: Discretisation: $a_\Delta = 0.01$. Gradient descent: $a_\Delta = 0.05$, $\eta = 15$, $\zeta = 0.0001$ and $\beta = 0.05$. Nelder-Mead: $\eta = 5$. Newton's Method: $a_\Delta = 0.5$, $\eta = 15$ and $\zeta = 0.0001$.

C. Results

The results presented in Table III were produced by running the experiment 100 times with each of the four action selection methods. The table consists of the percentage of successful runs; the minimum, median and maximum number of trials taken before the RL agent was able to successfully balance the pole for the full simulation time of 120 s and the median time taken to select the action to be taken. A run was considered successful if the agent was able to balance the pole without reaching the edge of the track for the full simulation time within 1000 trials.

As can be observed from Table III, all methods were able to succeed in all runs; however, Newton's Method trained in fewer trials on average than all other methods, whilst also performing the action selection in considerably less time than gradient descent and discretisation. Nelder-Mead performed the action selection in the fastest time, but achieved a similar

TABLE III
CART-POLE RESULTS

Method	Successful runs	Trials to Train (min / median / max)	Action Selection Time
Discretisation	100%	53 / 109.5 / 236	39 µs
Gradient Descent	100%	49 / 103 / 258	221 µs
Nelder-Mead	100%	48 / 98.5 / 316	4 µs
Newton's Method	100%	37 / 75.5 / 372	9 µs

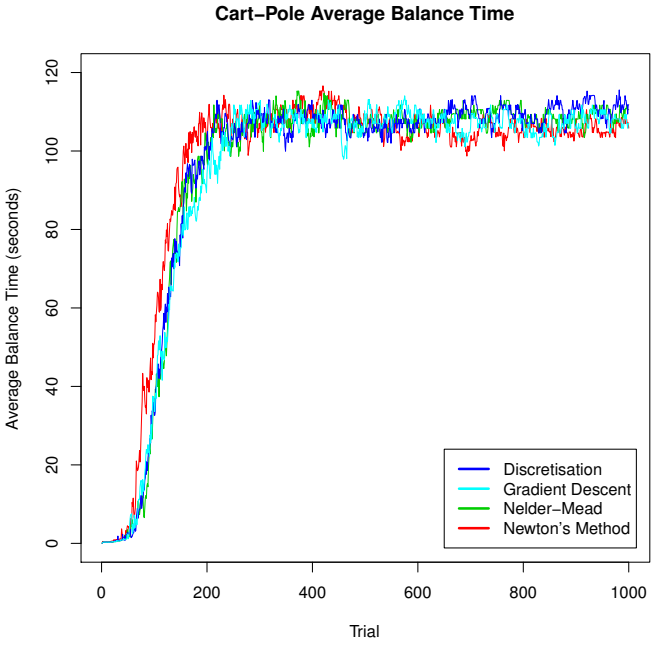

Fig. 2. Average balance time in each training trial over 100 runs.

average trials to train to that of discretisation and gradient descent. Discretisation and gradient descent require much more time for action selection as they evaluate $Q(s, a)$ many more times, due to the smaller a_Δ values. Although discretisation does not perform any iterative search, it still evaluates the value function from 201 different action values.

The average balance time in each trial for the four different action selection methods is shown in Fig. 2. This plot was produced by running the experiment 100 times for the maximum number of trials, i.e. not stopping at the first trial where balance was achieved. These balance times were averaged over all runs for each trial. Newton's Method achieves a longer balance time in the earlier trials than the other three methods which all perform similarly. However, in the trials after training, after approximately 200 trials depending on the method, all approaches achieve similar average balance times for all remaining trials.

VI. 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, both of which must be balanced [21]. The simulation in this experiment is the same as that of [5] except the maximum time of the simulation was 120 s rather than 20 s, making the task more challenging.

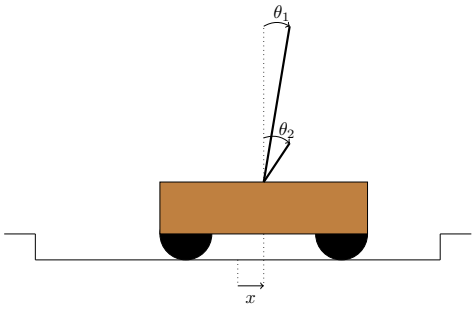

Fig. 3. Diagram of the double Cart-Pole problem. The cart is on a limited track with two poles, of differing lengths, attached to it. The poles are both free to rotate. The angles of the two poles from the balance position are θ_1 and θ_2. x is the position of the cart from the centre of the track.

The state vector comprises 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 is the force applied to the cart $a = F \in [-40, 40]$ N.

A. Existing Approaches

This double pole variant of the well known Cart-Pole benchmark has been used since [21] and has been applied as a benchmark for RL approaches [5], and for evolutionary algorithm based approaches [22]–[25]. In some cases these have also included a variant where the angular velocities are omitted from the state representation to make the task non-markovian; however, as I do not attempt to solve the problem of POMDPs here, I do not attempt this variant.

B. Description

The parameters used in this experiment are listed in Table IV, and the equations of motion used to update the environment are (14), as was used in [5].

$$\begin{aligned}
\phi_i &= 2m_i\theta_i^2 \sin\theta_i + \frac{3}{4}m_i \cos\theta_i \left(2\frac{\mu_i\dot{\theta}_i}{m_i l_i} + g\sin\theta_i\right) \\
\ddot{x} &= \frac{F - \mu_c \operatorname{sgn}(\dot{x}) + \sum_{i=1}^{2}\phi_i}{m_c + \sum_{i=1}^{2} m_i\left(1 - \frac{3}{4}cos^2\theta_i\right)} \\
\ddot{\theta}_i &= -\frac{3}{8l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_i\dot{\theta}_i}{m_i l_i}\right)
\end{aligned} \tag{14}$$

Each simulation was run for a maximum simulated time of 120 s 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 0.02 s 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 one of the poles fell or the cart reached the edge of the track and 0 otherwise.

TABLE IV
DOUBLE CART-POLE PARAMETERS

Parameter	Value
Cart mass (m_c)	1 kg
Pole one mass (m_1)	0.1 kg
Pole two mass (m_2)	0.01 kg
Gravitational constant (g)	9.81 m/s^2
Pole one length (l_1)	1 m
Pole two length (l_2)	0.1 m
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)	0.02 s
Maximum force (F_{max})	40 N

TABLE V
DOUBLE CART-POLE REINFORCEMENT LEARNING PARAMETERS

Parameter	Value
RL discount rate (γ)	0.9
RL step-size (α)	0.2
ANN hidden node quantity	12
ANN initial weight range	[-0.3,0.3]
ANN learning rate	0.2
ANN momentum	0.6

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 [5]. The selected action was limited to the allowable range before applying to the simulation.

This application of the double Cart-Pole problem is slightly different to that described in [5] in that the length of time the agent was required to balance the poles for was 120 s (rather than 20 s) which makes the task more difficult. Also the maximum range of exploratory actions in [5] was orders of magnitude larger than the allowable actions, thereby encouraging the agent to learn a bang-bang controller. Here, as I am specifically interested in continuous actions, I do not influence the learnt policy through use of such large exploratory actions.

C. Method

The same architecture and algorithms are applied as with the Cart-Pole problem (Section V), the parameters used in the double Cart-Pole problem are listed in Table V. Gaussian exploration was also applied as the Cart-Pole approach, but with the exploration parameter initially set to 1, and reduced at each time-step by 0.01 until it reached 0.

The parameters which are specific to each of the action selection methods and their values for this experiment were as follows: Discretisation: $a_\Delta = 0.01$. Gradient descent: $a_\Delta = 0.05$, $\eta = 10$, $\zeta = 0.0001$ and $\beta = 0.02$. Nelder-Mead: $\eta = 5$. Newton's Method: $a_\Delta = 0.5$, $\eta = 15$ and $\zeta = 0.0001$.

D. Results

The results were produced similarly with the single Cart-Pole results. Table VI presents the results of 100 runs of the experiment using each of the different action selection methods. The table shows the percentage of successful runs; the average number of trials taken to successfully balance the poles for the full simulation time of 120 s.

TABLE VI
DOUBLE CART-POLE RESULTS

Method	Successful runs	Trials to Train (min / median / max)	Action Selection Time
Discretisation	100%	75 / 283 / 671	148 µs
Gradient Descent	100%	61 / 280.5 / 723	509 µs
Nelder-Mead	100%	52 / 270 / 984	16 µs
Newton's Method	100%	33 / 191 / 705	38 µs

As can be seen from Table VI, all methods were able to achieve success in 100% of the runs. However, Newton's Method requires far fewer trials to train on average. Whilst the other three methods required a similar average number of trials to train.

Nelder-Mead achieved the fastest median action selection time (Table VI), taking half the time required by Newton's Method. However, Nelder-Mead and Newton's Method both took considerably less time to select actions than discretisation and gradient descent.

The average balance time per trial taken from 100 runs is presented in Fig. 4. This was produced by continuing training even after the agent is able to successfully balance the poles for the full simulation time (120 s). It can be seen that Newton's Method is able to balance for longer in the earlier trials, which suggests superior actions were selected. In later trials there is not such an obvious difference between the methods.

VII. CONCLUSION

Here I have applied four different continuous action selection methods to two continuous state- and action-space control benchmark problems from the literature. The SARSA RL algorithm was applied with an MLP approximating the state-action value function. In order to ensure the results of each of the action selection methods are directly comparable, all parameters for the RL approach and MLP were fixed, including the exploration schedule. For each approach only the parameters for that particular action selection method were tuned. The action selection methods were then compared both in terms of the time they took to select actions, and the quality of the overall RL approach when they were utilised.

From the experiments conducted here it can be seen that when Newton's Method is employed for action selection training can be achieved in fewer trials than with the other methods considered here. Newton's Method was also the second fastest in action selection time. Nelder-Mead was the fastest to select actions, taking approximately half the time required by Newton's Method. However, both of these methods took considerably less time than discretisation and gradient descent to select actions.

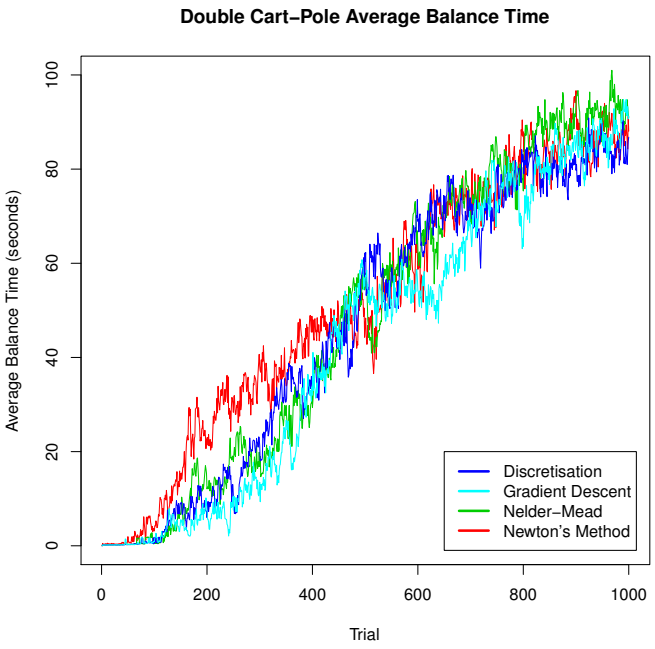

Fig. 4. Average balance time in each training trial of 100 runs.

As the multiple runs of Newton's Method from different initial actions do not depend on previous runs, the time taken could be reduced by applying parallel computation. This could also be applied to both gradient descent and discretisation, however, they were not able to match Newton's Method in terms of trials to train, and also took far longer to select actions. Nelder-Mead would not be subject to such speed improvements as each iteration requires the previous values.

The main problem with the use of Newton's Method is the fact that with a higher dimensional action-space extra computation would be required to repeatedly compute the inverse of the Hessian. It would be interesting to investigate the possibility of applying Newton's Method to problems with higher dimensional continuous action-space, and the impact this has on the action selection time. It may be possible to overcome this extra computation through the application of quasi-Newton methods [17].

Future work should investigate the application of a wider range of action selection methods and also compare the performance of the different approaches on a larger set of benchmarks. It would be particularly interesting to investigate how the action selection methods scale to problems with higher dimensional action-spaces and the applicability of these methods on physical control problems, rather than simulations, where the action selection time is more crucial.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press, 1998.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[3] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 6, pp. 1291–1307, 2012.

[4] M. Riedmiller, J. Peters, and S. 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*, April 2007, pp. 254–261.

[5] H. van Hasselt, "Reinforcement learning in continuous state and action spaces," in *Reinforcement Learning*, ser. Adaptation, Learning, and Optimization, M. Wiering and M. Otterlo, Eds. Springer Berlin Heidelberg, 2012, vol. 12, pp. 207–251.

[6] M. Lee and C. W. Anderson, "Convergent reinforcement learning control with neural networks and continuous action search," in *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014 IEEE Symposium on*, Dec 2014, pp. 1–8.

[7] B. D. Nichols and D. C. Dracopoulos, "Application of Newton's Method to action selection in continuous state-and action-space reinforcement learning," in *European Symposium on Artificial Neural Networks*, 2014, pp. 141–146.

[8] J. C. Santamarí, R. S. Sutton, and A. Ram, "Experiments with reinforcement learning in problems with continuous state and action spaces," *Adaptive behavior*, vol. 6, no. 2, pp. 163–217, 1997.

[9] S. Haykin, *Neural Networks and Learning Machines*, 3rd ed. Prentice Hall, November 2009.

[10] H. van Hasselt and M. A. Wiering, "Reinforcement learning in continuous action spaces," in *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, April 2007, pp. 272–279.

[11] D. C. Dracopoulos and B. D. Nichols, "Swing up and balance control of the acrobot solved by genetic programming," in *Research and Development in Intelligent Systems XXIX*, M. Bramer and M. Petridis, Eds. Springer London, 2012, pp. 229–242.

[12] S. Duong, H. Kinjo, E. Uezato, and T. 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*, December 2008, pp. 1540–1544.

[13] B. D. Nichols, "Continuous action-space reinforcement learning methods applied to the minimum-time swing-up of the acrobot," in *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, October 2015, pp. 2084–2089.

[14] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, August 2000.

[15] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[16] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder-mead simplex method in low dimensions," *SIAM Journal of Optimization*, vol. 9, pp. 112–147, 1998.

[17] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York: Cambridge University Press, 2004.

[18] L. C. Baird and H. Klopf, "Reinforcement learning with high-dimensional, continuous actions," Wright Laboratory, Tech. Rep., 1993.

[19] J. Si and Y.-T. Wang, "Online learning control by association and reinforcement," *Neural Networks, IEEE Transactions on*, vol. 12, no. 2, pp. 264–276, March 2001.

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

[21] A. P. Wieland, "Evolving neural network controllers for unstable systems," in *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. 2. IEEE, 1991, pp. 667–673.

[22] F. Gruau, D. Whitley, and L. Pyeatt, "A comparison between cellular encoding and direct encoding for genetic neural networks," in *Proceedings of the First Annual Conference on Genetic Programming*. MIT Press, 1996, pp. 81–89.

[23] F. J. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuroevolution," in *IJCAI*, vol. 99, 1999, pp. 1356–1361.

[24] V. Heidrich-Meisner and C. Igel, "Evolution strategies for direct policy search," in *Parallel Problem Solving from Nature–PPSN X*. Springer, 2008, pp. 428–437.

[25] ——, "Neuroevolution strategies for episodic reinforcement learning," *Journal of Algorithms*, vol. 64, no. 4, pp. 152–168, 2009.