FACULTY OF ENGINEERING
BEng in Engineering Design
Year 4 Individual Research Project

Autonomous Learning for Adaptive Agents

Autonomous Systems for Independent Living

Final Report - Report

Jerome Wynne

Project thesis submitted in support of the degree of Bachelor of Engineering

Project Advisor: Thea Morgan, Queens School of Engineering

June 5, 2018

Acknowledgements

I acknowledge and thank my project advisor, Dr Thea Morgan, for her patience, guidance, and helpful feedback. Likewise, I am grateful to Dr Ben Mitchinson of this project's sponsor company, Consequential Robotics, for the time and advice he invested in the project team. I would also like to thank Dr Paul Harper and the other Engineering Design Programme Directors for the roles they have played in shaping my development across the last four years.

The simulation environment used in this report was adapted from the `gym` Python library developed by OpenAi, a not-for-profit AI research company based in San Francisco, California. I thank them and their sponsors. In a similar spirit, I am grateful to the Python Foundation, the LaTeX Project, and all of the developers that have donated their time and wisdom maintaining and improving the Python packages `matplotlib`, `numpy`, `pandas`, and `pytorch`. These pieces of software enabled many of this report's simulation results and figures.

The last set of people I must acknowledge are those that made this past year enjoyable and stimulating: the Wu-Tang Clan, my brother, and my friends in Engineering Design. Their hard work and commitment to keeping it real continues to inspire me.

Declaration

The accompanying research project report entitled 'Autonomous Learning for Adaptive Agents' *is submitted in the fourth year of study towards an application for the degree of Bachelor of Engineering in Engineering Design at the University of Bristol. The report is based upon independent work by the candidate. All contributions from others have been acknowledged above. The views expressed within the report are those of the author and not of the University of Bristol.*

I hereby declare that the above statements are true.

SIGNED .....

FULL NAME .....

DATE .....

Executive Summary

Approximately 4 million people in England cite themselves as being limited 'a lot' in their daily activities by physical health problems. All of these people rely on the support of informal carers - spouses, children, friends, and neighbors - to go about their daily lives. As a result, England is estimated to be home to 5.4 million informal carers, making non-professional care the country's dominant form of social care. This group research project - *Autonomous Systems for Independent Living* - aims to develop a product that assists with the activities of day-to-day living, allowing some of the people receiving care to recover their independence.

The Project Brief specified that the Year 4 Individual Research Projects should investigate how technologies from robotics and autonomous systems (RAS) could be applied to create a commercially viable care system in Year 5. An initial phase of exploratory research at the beginning of Year 4 indicated that the Year 5 product could be most useful if it mitigated the effects of physical health problems in the context of a care recipient's home. The group's Individual Research Project themes are therefore focused on distinct areas of RAS, but are framed such that their outcomes can be combined in Year 5 to create a complete home-care system.

This Individual Research Project, *Autonomous Learning for Adaptive Agents*, was motivated by the fact that a person receiving care typically needs help with more than one of their daily activities. It developed and critiqued an intelligent controller that can learn to solve generic continuous control tasks. The proposed design was inspired by previous research in a field of artificial intelligence known as 'reinforcement learning', which offered a flexible framework for designing a general-purpose learning system. The controller enables the Year 5 design team to consider the possibilities offered by a home care system that can acquire new functionalities autonomously. The project's main contribution is a control system that is simple, fast, and capable of learning to solve control tasks without specialist intervention. The analysis that enabled this controller's development also explains why a popular approach to designing RL controllers needlessly constrains their learning rate.

The project consisted of three phases. In the first phase, the requirements of the Year 5 system were analyzed and distilled into a corresponding set of performance metrics, which were chosen to represent the technical and practical obstacles currently affecting RL's commercial viability. These metrics were then used to rank RL controllers from the literature. The second phase of work evaluated the performance of the top-ranking controllers on a testbed of continuous control tasks. Two of the controllers relied on nonlinear programming procedures, so an appropriate optimization algorithm was selected based on the results of a preliminary benchmark experiment. This experiment revealed why many existing RL controllers suffer from poor wall-clock learning times. The third and final phase of work involved synthesizing the results of the testbed study with recent literature findings to create an intelligent controller that meets the needs of the Year 5 system. The proposed design relies on simple random search over the parameters of a weighted sum of radial basis functions. The controller's hyperparameters[1] were optimized by way of a sensitivity study on a single testbed problem, then validated on the testbed's two remaining problems. The project's outcomes and research findings were appraised with respect to the individual and group project objectives to produce an assessment of how it might provide a route to autonomous control in Year 5.

The experimental results indicated that the intelligent controller developed here can be trained to a similar level of performance as existing RL control methods in $1/100^{\text{th}}$ the learning time. The controller can also solve nonlinear control problems without relying on onerous learning procedures. Both of these outcomes are substantial steps towards a practically viable embedded learning system for the Year 5 stage of work.

[1] A hyperparameter is a parameter of a machine learning model that is not optimized by the learning algorithm used to fit the model (i.e. it is specified beforehand by an engineer).

Contents

List of Figures

List of Tables

1 Introduction

1.1 Group Project Aims

In England, 4 million people cite themselves as being substantially limited in their daily activities by disability, illness, or age [1]. 5.4 million friends, neighbours, and family members support these people by acting as non-professional carers [2]. As a result, there are almost four times as many informal carers as there are professionals employed by the NHS[2] and private care providers [2]. Among those receiving care, an estimated 84% suffer from a physical limitation [3], a proportion reflected by the fact that carers are most likely to provide 'practical help' such as preparing meals, doing the laundry, and going to the shops [4].

Autonomous technologies may represent a solution to some of the UK's social care problems. Physical care tasks often have a clearly defined objective: to help a person out of bed, convey an item across a room or around a house, or to help a person to stand up. Clearly defined objectives are more easily represented in intelligent systems than abstract ones such as companionship or emotional support. Physical tasks are solved by torques and forces, quantities that existing robotics technology has proven adept at providing. Care robotics is currently a nascent sector, but there is reason to think that in the near future it will be more valuable. Problems with the performance of control and perception systems for robotics are gradually yielding to faster computers, more reliable sensors, and better-performing machine learning models.

This group explored how robotics and autonomous systems (RAS) could be used to help a person to prolong or recover their independence with respect to the Instrumental Activities of Daily Living (IADLs). The IADLs are tasks that health professionals use to assess a person's degree of physical and cognitive independence[3]. We chose to make the Activities a focal point of our work because they represented a pertinent and widely-recognized measure of independence, a measure biased towards physical tasks. We further scoped the brief to exclusively home-based care, the dominant form of care in England and a setting where a person could plausibly recover a high degree of autonomy.

The key stakeholder in the project was the care recipient. Secondary stakeholders included the care recipient's family, their carer, and the community they lived in. Following discussions with the Project sponsor, Consequential Robotics, and preliminary research into RAS, the group agreed to collectively cover all of the RAS research themes outlined in a whitepaper delivered by the UK-RAS Network in 2017 [5]. The group also adopted the representative IADL of 'getting out of bed' for the Year 4 phase of work. This clarified how the Individual Project outcomes would interact in Year 5[4] and gave context to the group's development process, depicted in Figure 1.

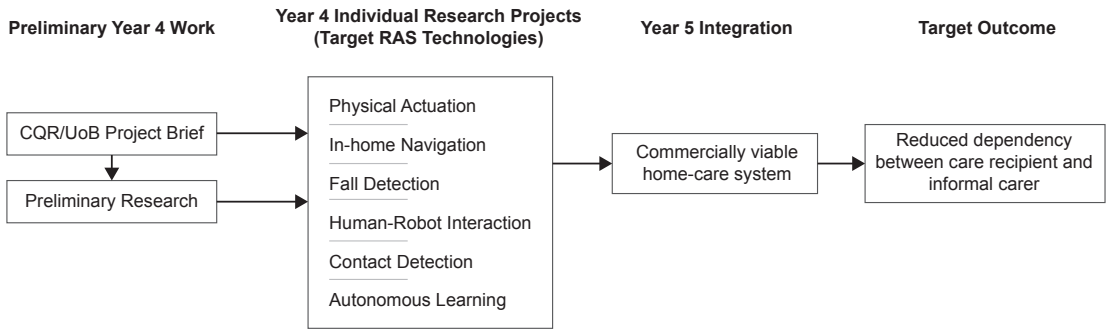


Figure 1: The group project design process that this work forms a part of.

[2]Including all doctors, nurses, and support staff.
[3]For example, two of the Activities are 'cleaning the house' and 'preparing meals'.
[4]The envisioned relationship between the separate Year 4 subsystems is depicted in Appendix Figure 1.

1.2 Individual Project Aims

Robots have been used in the manufacturing industry for almost half a century [6]. They offer uninterrupted operation, unparalleled reliability, and the capacity to perform tasks that are either unsafe or cannot be completed by human beings. A manufacturing procedure is amenable to automation because it can be broken down into a sequence of steps. Each system in an assembly line is designed to complete a single clearly delimited task: the abstract reasoning necessary to perceive the problem as a whole is left to an engineer. While each task is physically completed at assembly time by the robot, the overall problem is solved well in advance by specialist expertise.

Domestic automation has succeeded for problems of a type similar to that encountered on a factory floor. The washing machine, dish-washer, and thermostat all have tightly defined purposes. In the context of social care however, there is reason to think that a narrow-purpose product is unlikely to meaningfully increase a person's independence. Someone that needs help with one of the Activities of Daily Living is likely to need help with another [2]. Their needs are also likely to change over time, due to either health reasons or lifestyle choices. Within a single task, what is suitable for one person and their home may be insufficient elsewhere. A controlled environment similar to the drum of a washing machine is unfeasible, but so too is the clutter and expense associated with multiple infirmity- or activity-specific products.

This Individual Research Project develops an intelligent controller that can learn to solve unfamiliar control tasks autonomously. It was originally motivated by the prospect of realizing a product in Year Five that is capable of providing assistance across several of the Activities of Daily Living. The overall aim of the project was to deliver a controller that an automated system could credibly use to learn novel control tasks without specialist intervention. This aim was factored into a series of objectives, each of which were associated with several stages of development work.

1.2.1 Individual Project Objectives

Stage 1 - Literature Review & Requirements Analysis

(a) Survey alternative approaches to intelligent control, summarize their associated benefits and weaknesses, and determine suitable evaluation tasks for the controllers tested in Stages 2 and 3.

(b) Determine what properties an intelligent controller requires if it is to be used in the Year 5 system.

(c) Generate a set of metrics that quantify how well existing intelligent controllers satisfy these requirements.

(d) Use data from the literature to rank a set of candidate controllers identified in the Literature Review.

Stage 2 - Optimizer Selection & Testbed Evaluation of Existing Controllers

(a) Describe the controllers selected for evaluation on the testbed tasks.

(b) Benchmark a set of nonlinear optimization algorithms to determine which is most appropriate for the controllers that require them.

(c) Generate performance data for the top-ranked controllers using the control task testbed.

(d) Analyze the collected data and critique the controllers w.r.t. the performance metrics chosen in Stage 1 (c).

Stage 3 - Solution Generation & Testbed Validation

(a) Resolve the shortcomings of the controllers tested in Stage 2 (d) by proposing a novel controller design.

(b) Optimize this controller via a hyperparameter sensitivity study on a single control problem.

(c) Validate the optimized configuration on a set of further control tasks, drawing comparisons with the outcomes of Stage 2 (c) where necessary.

Stage 4 - Discussion & Appraisal

(a) Critically appraise the controller against the requirements and performance measures from Stages 1 (b) and (c).

(b) Discuss the limitations of the proposed control approach and suggest how these might be lifted in future work.

(c) Specify what applications the developed controller could viably be used for in the Year 5 project.

1.3 Report Overview

The report is structured around the project objectives listed above. The literature review justifies and explains the chosen control approach, then explores the ideas and issues that shape how its controllers are designed. The bulk of the report is dedicated to describing the process and outcomes of each project stage. These can be summarized as:

1. Pairing requirements of the Year 5 system with relevant controller performance metrics, then using these metrics to rank a selection of controllers from the literature (Stage 1).
2. Explaining how these controllers function and exploring how one of their components, their optimizer, should be chosen (Stage 2 (a), (b)). Both of these activities feed into an analysis of the process that limits the commercial viability of many current RL control systems.
3. Evaluating the top-ranking controllers on a testbed of continuous control tasks, then synthesising a novel controller in response to the shortfalls of the tested offerings (Stage 2 (c), (d), Stage 3 (a)).
4. Optimizing the novel controller via a hyperparameter sensitivity study, validating its final design on the control testbed, and comparing its performance with the controllers taken from the literature (Stage 3 (b), (c)).

The main body of the report closes with an appraisal of the novel controller's design (Stage 4). Its benefits and shortcomings are discussed, along with how it could be applied in Year 5. Directions for future work are also suggested, many of which are oriented towards resolving how the controller should be integrated into a commercial product. The report's conclusion summarizes the project's progress relative to its initial motivation, reviews how its objectives were met, and highlights how its contributions will influence the group's work in the Year 5 Design Project.

2 Background & Literature Review

This section provides the reader with the context necessary to follow the rest of the report. It justifies the type of intelligent controller that was selected for evaluation and describes the learning problem within its framework. The subsequent literature review shows how low-level technical problems could proliferate through to high-level feasibility issues: these problems informed the performance metrics and candidate controllers that were used in the ranking and experimental stages of work.

2.1 Approaches to Intelligent Control

Control theory and machine learning are large and diverse research fields. Approximately 150,000 core research[5] papers were published across the two disciplines in 2017 [7] [8]. This research project focused on intelligent control techniques developed within an area of artificial intelligence called 'reinforcement learning', a topic that sits at the intersection between machine learning and control. Reinforcement learning (RL) is essentially learning by trial-and-error: the field's roots are in psychology and an area of mathematics known as dynamic programming. A dynamic program is a recipe the can be used to optimize a sequential decision-making process, such as a series of investments or the route taken by a delivery van. Dynamic programming emerged in the 1950s in the context of optimal control [9], but it was not until the early 1980s that it was applied in a learning system to produce what would be recognized today as a reinforcement learning controller [10].

The decision to investigate RL techniques was prompted by several observations. First, other branches of control theory often assume that a model of a problem's dynamics is available. This is true of optimal control [9], where two popular approaches - model-predictive control and the linear quadratic regulator - are only possible if a model of the environment's dynamics is accessible [11]. In some cases, it is also assumed that these dynamics can be approximated using a system of time-invariant linear differential equations, as is common in classical control theory [12]. Reinforcement learning, on the other hand, allows a controller to be designed even if the target environment is unknown or is loosely defined [10]. This property was attractive because the architecture of the Year 5 system was not known and the assistance it would provide would necessarily vary from person-to-person. Whether the Year 5 product provided support across several ADLs or only one, its controller would need to adapt its response based on the needs and characteristics of its user. On account of its flexibility and a broader need to shrink the space of research evaluated, RL was chosen to be the central focus of the literature review and the project as a whole.

2.2 Reinforcement Learning Background

A controller that is capable of learning to solve various tasks needs a standard way of expressing those tasks. In RL, the controller learns how to map states to actions that maximize cumulative future reward (a.k.a. 'returns'). This problem can be formalized using the following concepts.

- State information - which might consist of joint angle measurements, rangefinder data, and video feeds - is encoded in the elements of a vector s. The set of all possible states corresponds to the space $\mathcal{S}$.
- The actions available to the controller - such as motor torques or end-effector coordinates - correspond to elements of a vector a. The set of all available actions forms the action space $\mathcal{A}$.
- The action that the controller chooses in each state is described using a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$. This function is known as the controller's 'policy'.
- The controller's environment is treated as a function that maps the current state and action to the next state. In

[5] As opposed to research that applied techniques from the disciplines to other fields.

general this transition may also be a function of variables that are hidden to the controller[6] (i.e. that are not part of the state vector s). The controller experiences the state trajectory in discrete time, allowing the state vector to be indexed according to time e.g. s_t is the state at time t.

- The objective of a task is represented using a scalar function $r(s,a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. This is known as a task's 'reward function'. The controller receives a reward input each time it undergoes a state transition.

The controller's objective is to maximize the amount of reward it generates across all future time-steps. It does this by changing its policy based on feedback from the environment. By convention [10] [9], this problem is usually stated as:

$$\text{Given environment } f \text{ with reward function } r \text{ and current state } s_t = s,$$

$$\text{maximize} \quad V^{\pi}(s) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r\big(s = s_{t+k},\ a = \pi(s_{t+k})\big)\Big] \quad : \ s_t = s,\ s_{t+k+1} = f(s_{t+k}, a_{t+k}) \tag{1}$$

$$\text{across all states } s \in \mathcal{S} \text{ with respect to the policy } \pi.$$

$V^{\pi}(s)$ is called the state-value function, or returns, of policy π - the average sum of rewards the controller receives across all time-steps when starting in state s then following policy π. V^{π} measures both how good a policy is and how valuable states are under that policy. The average, $\mathbb{E}$, is necessary in case the environment, reward function, or policy are stochastic[7]. The scalar $0 < \gamma < 1$ discounts future rewards so that $\lim_{t \rightarrow \infty} \gamma^t = 0$ and V^{π} is finite. If the environment has a termination condition, such as a time limit, then the problem task is called 'episodic' and the ∞ in the summation should replaced with a termination time-step T. The discount factor should also be omitted.

Most modern reinforcement learning techniques rely on an extension of the state-value function known as the action-value function [13] [14]. It quantifies how valuable an action is in a given state: how much reward the controller would receive if it began in state s, took action a, and followed policy π thereafter. Q^{π} is the action-value function for policy π, where

$$Q^{\pi}(s,a) = \mathbb{E}\Big[r(s_t = s, a_t = a) + \gamma V^{\pi}(s_{t+1})\Big] \quad \text{for all } s \text{ in } \mathcal{S},\ a \text{ in } \mathcal{A} \tag{2}$$

The action-value function allows an controller to compare how favourable the actions available to it are. By updating its policy to select actions with a higher action-value, the controller is guaranteed to improve its policy[8].

The controller searches for a policy that maximizes $V^{\pi}(s)$ using a learning algorithm. This component, alongside the choice of reward function and environment, determines how quickly the controller learns and how successful its final policy will be. All RL learning algorithms can be described as alternately

(1) estimating $V^{\pi}(s)$ (or $Q^{\pi}(s,a)$) for the current policy across all states and actions,

(2) updating the policy, $\pi(s) = a$, so that it selects higher-value actions.

Repeating (1) $\rightleftharpoons$ (2) until the policy ceases to improve is known as generalized policy iteration (g.p.i.) [10] . The consensus in the literature is that generalized policy iteration is the most efficient means of finding an optimal policy (i.e. solving Equation 1), although different learning algorithms implement the process in different ways. An important

[6] The reward and environment transition could also be functions of variables that the controller does not observe (i.e. that do not form part of the state vector).

[7] A stochastic policy selects its actions probabilistically - policy improvement consists of altering the probability of each action so that better actions are more likely to be selected. Likewise, in a stochastic environment the state transition is probabilistic - the same action will not always yield the same state transition.

[8] The models discussed later will only appear to contain Q^{π}, but in fact also contain an estimate of V^{π}, since $V^{\pi}(s) = Q^{\pi}(s, \pi(s))$. This means that they conform to the problem described by Equation 1.

distinction to make is between the true action-values and state-values for a given policy π, written $Q^\pi(s,a)$ and $V^\pi(s)$, and the controller's estimates of these values, $\hat{Q}^\pi(s,a)$ and $\hat{V}^\pi(s)$.

The components of the RL control problem discussed so far are shown interacting in Figure 2. Although an RL controller can learn autonomously, the engineer must specify its environment, reward function, and the actions and observations it has access to. Moreover, the RL framework assumes that

- the objective of a task can be expressed or approximated via a reward function,
- an environment's transition and reward dynamics are fixed with respect to time,
- the controller has access to state data that is relevant to the reward it receives,
- and the controller is permitted to explore and fail during learning.

The strength of these assumptions is context-dependent. For simple tasks it may be clear what reward function is appropriate. For more complex behaviours this is often not the case: current RL research efforts are directed towards strategies for reward function design, including generating reward functions from demonstrations [15] [16] and using policies that can be applied to new reward functions without needing re-training [17]. The second and third assumptions depend on how the RL problem is framed by an engineer. Providing useful state data can be expensive. Some forms of data may also require extensive pre-processing before they can be used by a controller (audio and video feeds, for example). Problems relating to whether a controller is allowed to fail during learning are clearly important to how an intelligent controller can be used in Year 5, and will be discussed in detail in the report section corresponding to Stage 4. For the time being, assume that an appropriate reward can be specified, the environment is stationary and provides useful state information, abundant interaction with the environment is permitted, and that failure is an acceptable part of learning.

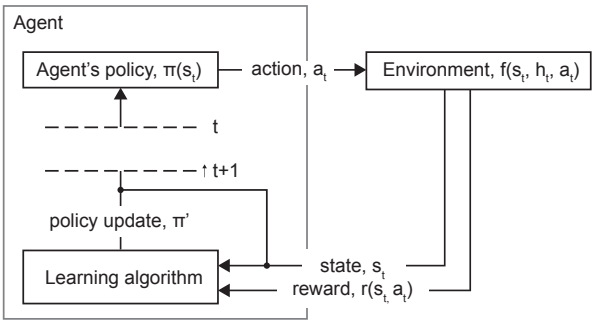


Figure 2: The reinforcement learning problem. A controller (agent) must maximize the reward it receives from an environment by modifying how it maps from states to actions. This mapping can be interpreted as a decision process. Boxes depict problem components. Arrows depict flow of information.

2.3 Problems in Contemporary Reinforcement Learning

RL controllers can be understood in terms of the problems they are designed to solve. Whereas the previous section outlined the RL problem structure, this section underlines the practical difficulties that complicate its solution. These difficulties form the basis for a critique of modern RL control and research practice, which further informs the experiments and performance metrics used later in this report.

2.3.1 Continuous Action- and State-Spaces

In small discrete state- and action-spaces it is possible to use tables to store a policy and to maintain estimates of action-values. In operation, the controller chooses an action by looking up the current state in a policy table. To update the action-value estimate for state-action pair (s, a), the controller amends the corresponding cell in a table of $\hat{Q}^\pi(s, a)$ values [10]. In continuous state- and action-spaces, the tabular approach is not possible. The spaces could be discretized, but recent work instead favours using functions to approximate Q^π and to represent π [13] [18]. $\hat{Q}^\pi(s, a; \theta^Q)$ and $\pi(s; \theta^\pi)$ are then functions parametrised by the sets θ^Q and θ^π respectively. The controller must adjust the members of $\theta^{\hat{Q}}$ so that $\hat{Q}^\pi$ approximates Q^π (step 1 of g.p.i.), and update the elements of θ^π so that the policy earns more reward (step 2 of g.p.i.).

2.3.2 The Curse of Dimensionality

Robots typically operate in continuous state and action spaces that can contain dozens or hundreds of dimensions [14]. The volume of the domains of $V^\pi(s)$ and $Q^\pi(s, a)$ increases exponentially with each additional dimension in $\mathcal{S}$ or $\mathcal{A}$, meaning that the region immediately adjacent to a datapoint occupies very little of the space as a whole, a problem illustrated in Figure 3a. A few datapoints in a big space can be fit by lots of different plausible functions, many of which will not accurately describe the true function generating the data. This is an issue seen across machine learning known as *the curse of dimensionality* [19]. In RL, the curse can make it difficult to accurately estimate the functions V^π and Q^π, which hampers the controller's rate and quality of learning [9].

2.3.3 Estimating the Value Function

Two methods for estimating Q^π/V^π have been proposed in the literature. *Monte-Carlo* methods record the rewards and states the controller observes when executing policy π, then compute the returns following each state according to Equation 1. These point-wise estimates of $V^\pi(s)$ are used to fit the approximation function $\hat{V}^\pi$ [14]. *Temporal-difference* (TD) methods, on the other hand, update the parameters of $\hat{V}^\pi$ or $\hat{Q}^\pi$ at every time-step by computing the difference

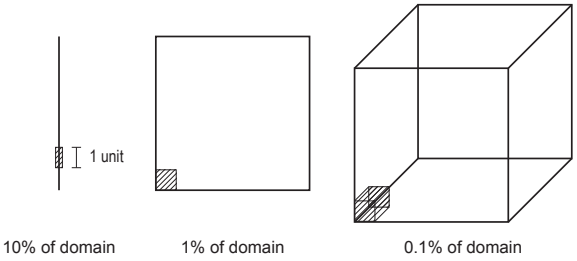
$$\underbrace{\delta}_{\text{TD-error}} = \underbrace{\hat{Q}^\pi(s_{t-1}, a_{t-1})}_{\text{Estimated value of previous state-action}} - \underbrace{\left(r_{t-1} + \gamma V^\pi(s_t)\right)}_{\text{Reward received + discounted estimate of current state's value}} \tag{3}$$

A valid action-value function Q^π must satisfy Equation 2, in which case the average TD-error across all state transitions is zero. Temporal-difference updates were inspired by the mechanism that allows synaptic modification (i.e. learning) in the frontal cortex. An animal's reward prediction errors are encoded in a dopamine release that modulates synaptic strength [20]. Recent RL controllers usually use TD-learning because it is thought to be more sample-efficient[9] than Monte-Carlo estimation [10], however this conclusion is based exclusively on empirical evidence.

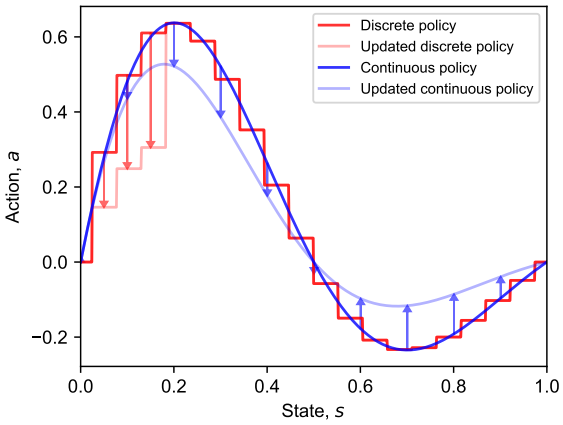
2.3.4 Biased Data

Using a function to approximate a policy's action-values introduces statistical problems that can prevent the controller from learning successfully. If $\hat{Q}^\pi(s, a)$ is tabulated, then the action-values can be updated individually. If $\hat{Q}^\pi$ is a function, then the action-value estimates are coupled under that function's parameters and cannot be adjusted in isolation, as is depicted in Figure 3b. Accurately estimating $\hat{Q}^\pi$'s error therefore requires state-action pairs to be drawn evenly from across the state and action spaces [13]. Figure 4a shows how fitting a function using data from a small part of the domain can produce an inaccurate fit. This problem is especially pertinent in RL because an controller's most recent experiences are correlated and occupy may occupy only a small region of the action-value function's domain, the joint state-action space $\mathcal{S} \times \mathcal{A}$.

[9] Sample complexity is the average number of state transitions an RL controller must observe before it can learn a usable policy - in other words, how much experience the controller requires. It is estimated empirically for a given environment. A controller that is said to be 'sample-efficient' has a low sample complexity relative to other methods.



(a) The curse of dimensionality - it can be difficult to fit a function to datapoints occupying a high-dimensional space. The data's sparsity permits many plausible functions to be fit with low error. In robotics, the domains of $Q^\pi(s,a)$ and $V^\pi(s)$ can contain many dimensions.



(b) Discrete policies can have their actions altered separately; shifting the parameter of a continuous policy can change the actions it maps to in all states.

Figure 3: Obstacles to learning (1)

2.3.5 Learning Divergence

A policy update can dramatically affect which states a controller visits and what returns that controller receives. In the context of g.p.i., a large[10] policy update can make any existing estimates of Q^π/V^π grossly inaccurate. If a policy update is applied before $\hat{Q}^\pi/\hat{V}^\pi$ are made more accurate, then the policy may degrade rather than improve. If this phenomena recurs, then the controller will fail to learn successfully [14] [13].

2.3.6 Choice of Policy and Value Function Parameterisation

The parameterisations chosen for $\hat{Q}^\pi(s,a;\theta^Q)$ and $\pi(s;\theta^\pi)$ influence how efficiently a controller can learn, the accuracy of its action-value approximation, and the maximal performance of the policies it converges upon. Flexible functions with many parameters such as those shown in Figure 4b can take on a wider variety of shapes[11] than simple ones with few parameters, but require more data and compute time to be fit. Nonlinear[12] functions are particularly susceptible to divergence, but saw a resurgence in interest following a paper published in 2015 showing that neural networks, a type of nonlinear function, could represent policies that surpassed human-level performance across a set of video-games [13]. This achievement was impressive since the state vector contained only the intensity values of the game's on-screen pixels, but came at the cost of a large increase in the controller's complexity, particularly with regards to preventing divergence. Since then, neural networks have been the default choice of policy function in RL research papers, however there is evidence that simpler learning algorithms using linear policy functions can achieve a similar level of performance on locomotion problems [21]. It seems likely that the litany of state-of-the-art successes of neural networks in other areas of machine learning, such as machine vision [22] and natural language processing [23], are part of the reason that researchers currently favour them for RL.

2.3.7 Non-Determinism & Controlled Randomness

In 2017, Henderson et al. [24] [25] exposed experimental flaws in several seminal RL papers [26] [27] [18] that represented a wider 'reproducibility crisis' within RL research [28]. This crisis was the result of RL controllers' sensitivity to internal and environmental stochasticity. Randomness suffuses the learning process: the parameters of

[10] 'large' in the sense that it substantially changes the controller's returns.

[11] This is only a heuristic - for example, the function $g(x) = \text{sign}\left(\sum_{i=0}^{100} \theta_i x\right)$ has one-hundred parameters but can take on only two values.

[12] A nonlinear function is a function for which the output is not directly proportional to each of its inputs: for example, the 9th-order polynomial and neural network in Figure 4b are nonlinear functions.

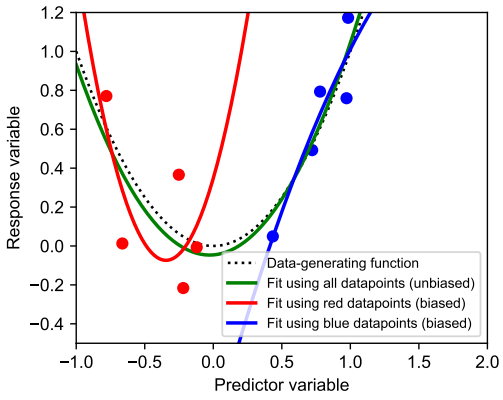
the action-value function and policy are often randomly initialized, quantities such as V^π and Q^π are estimated using a random sample of state transitions, policy search relies on random perturbations in either action- or parameter-space, the state of the environment at the beginning of an episode can be randomly generated, and the task environment itself may contain noise-generating[13] processes. These factors mean that a controller's performance is non-deterministic and must be evaluated across many different initialisations of its random number generator (separate initialisations are known as 'seeds').

2.3.8 Choice of Evaluation Environments

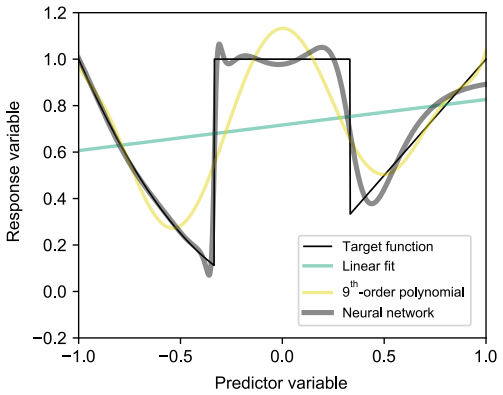
The apparent performance of a controller depends on the selection of environments used to evaluate it. There is evidence that no one controller performs best across all types of task [24], however only a limited set of benchmarks are widely used within the RL research community [25]. To the best of this author's knowledge, there do not yet exist any techniques for predicting how a controller will perform on new environments given its performance on previous ones. The aforementioned benchmarks used by researchers were created on the basis of history and practical utility, as opposed to a consideration of how their dynamics differ. This means that it is difficult to profile how a controller's performance will vary as a function of environment dynamics. Moreover, the outward complexity of a task is not necessarily an accurate measure of how difficult it is to solve: a simple 2D pendulum swing-up environment studied in this project is, by one fundamental measure of difficulty, harder to solve than a 3D 27-degrees-of-freedom humanoid walking task[14] [21].

2.3.9 Summary

This section highlighted the problems that a successful RL controller must contend with. An effective controller will have an appropriately flexible policy representation, be able to accurately and quickly estimate Q^π or V^π using a minimum of prior experience, avoid having its learning process diverge, and will converge as quickly as possible on an optimal policy. The controller should also be robust to random parameter initializations, the settings of its hyperparameters, and the scale of the reward, action, and state domains it receives[15].



(a) Functions fit on a restricted part of the target function's domain may be inaccurate.



(b) Choice of parameterisation affects the quality of fit achieved by an approximator.

Figure 4: Obstacles to learning (2)

[13] In this context, 'noise' refers to a change caused by a variable that is not observed (i.e. that does not form part of the state vector).
[14] The walking task can be solved by a linear policy, whereas the pendulum problem cannot be.
[15] This was not discussed explicitly under any particular sub-heading, but represents an important way of standardizing how a controller 'perceives' an environment.

3 Research Methodology & Experimental Results

3.1 Overview

This section details how the project's objectives were met. It begins by defining the requirements that need to be satisfied if an intelligent controller is to add value to the Year 5 product. These requirements, alongside the technical issues discussed in the Literature Review, determined the performance metrics that were used to rank RL controllers from the literature. The highest-ranked controllers were selected for evaluation in Stage 2, where they were tested against a suite of continuous control tasks. Before then, however, a benchmark study of several nonlinear optimization algorithms explored why two of the chosen controllers suffer from slow wall-clock learning times. This allowed an appropriate optimizer to be specified for the controller implementations evaluated in Stage 2, and drew attention to a practical limitation of many recent RL controllers that is unmentioned in the literature. The data gathered in Stage 2 revealed several shortcomings in existing RL controllers: Stage 3 involved designing a controller that mitigated these weaknesses. This design was then subjected to a hyperparameter sensitivity analysis and a validation study, both of which used the testbed. The controller was then appraised with respect to the other controllers and the Year 5 requirements. Figure 5 outlines this research process.

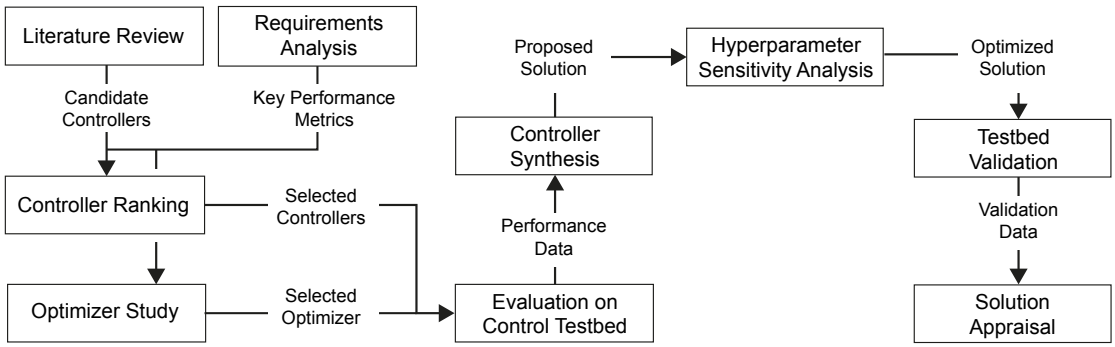


Figure 5: Functional diagram of the research process that was undertaken. Boxes correspond to work stages, arrows to stage inputs/outputs.

3.2 Stage 1 (b-c) - Project Requirements & Key Performance Metrics

3.2.1 Stage 1 (b) - Project Requirements

This section specifies the requirements and performance metrics that were used to evaluate the RL controllers considered in later stages of work. While no fixed design had been or is yet determined for the Year 5 robot, it was reasonable to assume that an intelligent controller would need to be able to

- learn a policy that could accomplish the task at hand (i.e. is 'usable'),
- learn consistently,
- learn quickly,
- learn efficiently (as measured by sample complexity),
- and learn across a range of environments.

In addition to these quantifiable requirements, the controller used in Year 5 needed to be as simple as possible. Two ways of measuring this were identified: the number of controller hyperparameters that an engineer needed to specify and, heuristically, how readily a controller could be explained to other people.

3.2.2 Stage 1 (c) - Key Performance Metrics

Policy quality: A policy's quality can be measured in terms of its returns and its qualitative (i.e. visually assessed) behaviour. Policies that generate lots of reward and appear to behave near-optimally are classified as being successful. A learnt policy's returns can also be compared with those received by a dummy policy, which samples actions randomly from the action space, or the task's optimal policy (if it is known). Qualitative analysis of an controller's behaviours is a natural way of seeing whether a policy is usable. For a detailed insight into a controller's overall performance, however, it is necessary to quantitatively aggregate returns data across several environments, over many separate episodes, and over multiple different seeds.

Learning consistency: Reliability is a requirement for most engineering components. For an intelligent controller to be a successful part of the Year 5 product, it will need to provide tight confidence bounds on the time it takes to learn a usable policy. Two metrics suitable for measuring reliability are the variance of the time taken to converge on a usable policy (learning reliability), and the variance in policy returns at convergence[16] (performance reliability a.k.a. policy robustness).

Learning rate/Learning efficiency: Fast learning is desirable because it lowers the financial cost of renting or purchasing computer hardware and improves the user experience. A robot that learns quickly is impressive. Rate of learning is a function of the controller's learning algorithm, implementation software, and execution hardware[17]. The effects of hardware and software on run-time are rarely mentioned in published research, whereas the relationship between a controller's learning algorithm and its sample complexity often is [29] [18] [28]. Both wall-clock learning time and sample complexity are relevant measures of controller performance in this work.

Task generality: The capacity to solve a range of control tasks would make an RL controller useful across several of the subsystems that may form a part of the Year 5 product. The ability of a controller to learn across a range of tasks is usually proven by presenting its learning curves[18] for a set of several different tasks. A benefit of this approach is that it demonstrates that the learning process depends on the environment as well as the controller. A shortfall of it is that differences between environments have not yet been quantified in a way that allows engineers to predict when one RL controller will out-perform another, or will learn successfully at all. This makes the evaluation process uncomfortably speculative and is a problem that needs to be explored further, both in Year 5 and within the wider RL research community. Time constraints meant that the standard approach of evaluating the controllers across a series of benchmark tasks was adopted for this project.

Table 1 summarizes the requirements of the Year 5 project. Each requirement is associated with at least one of the performance metrics developed above. These metrics are used in the next section to rank learning controllers from the literature and are used later when evaluating the performance of the highest-ranked controllers across a set of continuous control tasks.

3.3 Stage 1 (d) - Ranking of Reinforcement Learning Controllers

In this subsection a set of reinforcement learning controllers for continuous control are ranked using the chosen performance metrics. The data that informed the ranking is described, abbreviated explanations of the controllers considered are given, and the ranking process undertaken is justified. The set of candidate controllers consists of two widely used

[16] This variance is computed for individual controllers (i.e. seeds) across several dozen episodes of an environment.

[17] Learning rate also depends on environment variables such as the environment dynamics, reward function, state representation, and available actions. These aren't mentioned since they are held fixed for the purposes of evaluating the various controllers.

[18] A learning curve is a plot of a controller's returns against either the number of state transitions or the number of episodes it has experienced since being initialized.

[19] The mean of the returns is taken w.r.t. time, whereas the variance is w.r.t. random number generator seed.

research baselines and two recent research contributions. All four controllers date from within the last three years and can be said to accurately represent the current state-of-the-art in reinforcement learning. The ranking reveals that Augmented Random Search (ARS) [28] surpasses the other controllers on all attributes. Normalized Advantage Functions (NAF) [27] and Deep Deterministic Policy Gradients (DDPG) [18] follow, with Proximal Policy Optimization (PPO) [30] scoring lowest.

This stage of work indicated that there exist few resources detailing the performance of RL controllers numerically (i.e., in a table). Table 2 contains the data that was used to rank the candidate controllers. It was assembled using the results of Henderson [24], Mania [28], and Gu [27], then corroborated using benchmark data from Duan [25]. In the task environments these authors studied, a controller learned to walk using a novel actuator morphology. The reward was a function of the agent's horizontal velocity. Only the relative performance of the controllers within single tasks was of interest in this project: the magnitude of the returns on different tasks is a product of the choice of reward function and should be ignored[20], hence the results in Table 2 are normalized by the row maximums. The controller's returns at convergence were averaged[21] across the final few episodes of training to produce its 'average returns'. The mean number of episodes required to reach a reward threshold was taken as a measure of sample complexity. The maximum, median, and mean average reward received after a given number of training steps were used as measures of the policy quality the controller achieved. These quantities were computed across several separate seeds: fewer seeds indicates that the associated estimates had a greater uncertainty attached.

ARS is a simple controller that performed well across all environments. It uses a linear policy[22] rather than a neural network and estimates V^π by evaluating the policy π on a single episode (i.e. it relies on Monte Carlo estimates). It is comparatively easy to implement, the behaviour of its policy cam be readily interpreted[23], and the data gathered indicated that it had superior sample efficiency (contradicting conventional wisdom regarding Monte Carlo methods). In addition to this, ARS had previously been shown to learn locomotion policies orders of magnitude more quickly than the other controllers that were considered [28].

DDPG, NAF, and PPO are substantially more complex than ARS. They use nonlinear functions to explicitly estimate

[20] An open problem in RL is how best to compare controllers. Average returns received at convergence are a widely-used but flawed measure, because the qualitative performance of an agent is unlikely to vary linearly with its returns.

[21] Averaged in the sense of an arithmetic mean.

[22] A linear policy maps states to actions using a linear function, i.e. $a = \theta^T s = \pi(s)$, where θ is a vector of learnt parameters. The states are weighted then summed to produce an action, making it simple to see which states influence the action selected.

[23] This is because each of its parameters corresponded to the rate of change of an action with respect to a state.

Year 5 Controller Requirement	Performance Metric
Learns a usable policy on a single task	Mean returns
	Variance of mean returns[19]
Learns a usable policy quickly	Mean sample complexity
	Mean wall-clock learning time
Learns a usable policy consistently	Variance of sample complexity
	Variance of wall-clock learning time
Learns across different types of task	Learning curves across several environments
Conceptual/implementation simplicity	Controller complexity

Table 1: Performance metrics mapped to the Year 5 requirements for a learning system.

Q^{π} and to represent the policy π. They also rely on many algorithmic accessories that complicate their explanation, as will be seen in the next section. This is particularly true for PPO, which introduces additional hyperparameters over DDPG and NAF and uses a convoluted loss function for step 1 of g.p.i.. Unlike ARS, these controllers are susceptible to divergence: they estimate $Q^{\pi'}$ by altering an existing estimate of Q^{π}, where π' is an updated version of the policy π. They are difficult to use in practice because they take a long time to train (even on expensive computing hardware) and are comparatively onerous to implement. This makes it hard to debug them, optimize their hyperparameters, and replicate their behaviours with respect to published research findings (particularly if a complete specification of their many hyperparameters is omitted for publication).

Given their limitations, it may be asked what benefits DDPG, NAF, and PPO offer. The answer is that linear policies cannot solve certain types of problems, whereas sufficiently large neural networks have been proven to have the capacity to approximate any continuous function [31]. ARS, on the other hand, is limited in the class of problems it can solve because it uses a linear policy.

The ranking process consisted of several steps. The controllers were ranked on the attributes outlined in the previous section, then the rank indices were weighted and summed to produce an overall score for each controller. The weight of each attribute was selected by considering the relationship between the Year 5 requirements. A convoluted controller is difficult to implement, test, and analyze, factors that make development more expensive and risky. Simplicity is essential to the practical success of an intelligent controller in Year 5. By contrast, sample complexity, wall-clock run-time and policy quality are theory-oriented measures of controller performance. To determine which controller was most successful in balancing development and performance requirements, simplicity was assigned the same weight as the theory measures combined.

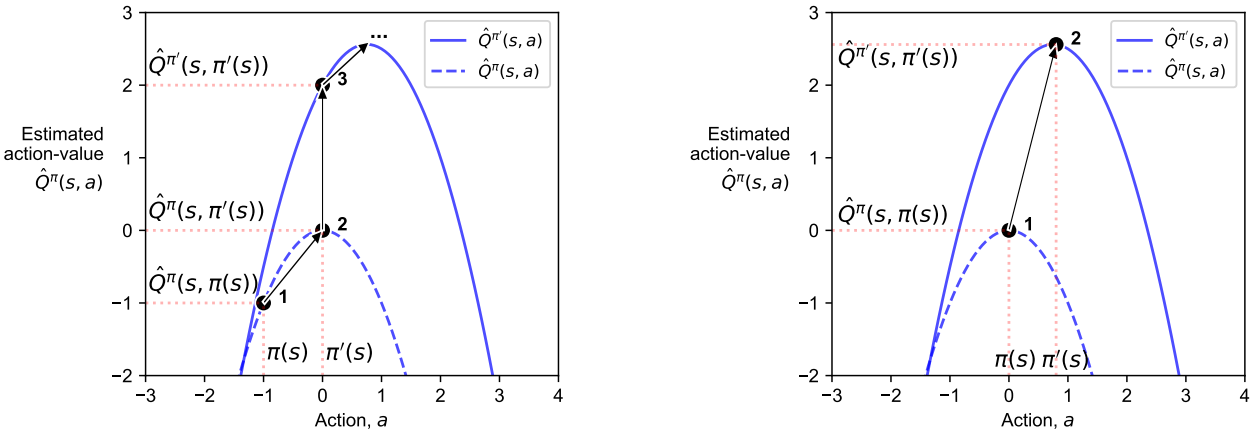
The overall score for each controller is shown in the bottom subtable of Table 2. Based on the rankings and foregoing discussion, ARS, DDPG, and NAF were selected for evaluation on the control testbed. Initially only two controllers were to be tested (ARS and DDPG), however sufficient project time and budget were available for data to be gathered on a third (NAF). This data was a valuable resource: it provided additional context for evaluating DDPG/ARS, and underlined the central criticism of DDPG and NAF-type learning algorithms that emerged from the optimizer study in Stage 2 (b).

Table 2: (Top) Data gathered from the literature; (Centre) Controller attribute ranking; (Bottom) Rank histogram and score calculation.

Metric	Environment	Controller				# of seeds	Source
		DDPG	PPO (*TRPO)	ARS	NAF		
Average # of episodes to reach a reward threshold	Hopper	> PPO	1*	**0.2**	≈DDPG	3	Mania et al.
	HalfCheetah		1*	**0.4**		3	
	Walker2d		**0.59***	1		3	
	Ant		1*	**0.28**		3	
Maximum average reward after ≈10^6 updates	Hopper	0.33	0.38	**1.00**	-	3	
	HalfCheetah	**1.00**	0.28	0.77	-	3	
	Walker2d	0.38	0.19	**1.00**	-	3	
	Ant	0.10	0.00	**1.00**	-	3	
Median average reward after ≈10^4 updates	Hopper	-	-	**1.00**	-	100	
	HalfCheetah	-	-	**1.00**	-	100	
	Walker2d	-	-	**1.00**	-	100	
	Ant	-	-	**1.00**	-	100	
Mean average reward after ≈10^6 updates	Hopper	0.59	**1.00**	-	-	5	Henderson et al.
	HalfCheetah	**1.00**	0.60	-	-	5	
	Walker2d	0.53	**1.00**	-	-	5	

Attribute	Attribute weight	DDPG	PPO	ARS	NAF
Simplicity rank	3	3	4	1	2
Sample complexity rank	1	= 3	2	1	= 3
Wall-clock runtime rank	1	= 2	3	1	= 2
Policy quality rank	1	= 2	= 2	1	= 2
Policy variance rank	-	-	-	-	-

Controller	Rank frequency				Weighted scores
	Rank 1	Rank 2	Rank 3	Rank 4	
ARS	4	0	0	0	1*(3 + 1 + 1 + 1) **= 6.00**
NAF	0	3	1	0	0.75*(3 + 1 + 1) + 0.5*(1) **= 4.25**
DDPG	0	2	2	0	0.75*(1 + 1) + 0.5*(3 + 1) **= 3.50**
PPO	0	2	1	1	0.75*(1 + 1) + 0.5*(1) + 0.25*(3) **= 2.75**
Rank score	1	0.75	0.5	0.25	



(a) DDPG executes each step of g.p.i. independently.

(b) NAF performs both steps of g.p.i. at once.

Figure 6: Operating principles of two of the controllers evaluated on the control testbed, DDPG and NAF.

3.4 Stage 2 (a) - Selected RL Controller Architectures

This subsection describes the controllers that were selected for evaluation on the control task testbed. The discussion attempts convey the features of the controllers that were most relevant to their relative viability for Year 5: by necessity, some details are omitted. The most important outcome of this section is an overview of how DDPG and ARS differ, since these differences were crucial to analyzing the experimental testbed data in Stage 2 (c) and making informed design decisions in Stage 3, where the novel controller was created.

3.4.1 Augmented Random Search (ARS)

ARS [28] finds an optimal policy by performing a random search over the policy's parameter space. As has been mentioned, its policy is linear: each action it generates a is determined according to $a = \pi(s; \theta^\pi = \{w\}) = w^T s$, where w and s are vectors of real values[24]. ARS - like all RL controllers - is trained by exposing it to a series of episodes of its environment. At the beginning of each episode the ARS controller generates N random parameter perturbations $\lambda_i \sim \mathcal{N}(0, I),\ i = 1, ..., N$, where λ_i is a vector of the same size as w that is sampled from a multidimensional normal distribution $\mathcal{N}$. $2N$ policies are then executed against the environment, two for each perturbation direction: $\pi(s; w + \nu\lambda_i)$ and $\pi(s; w - \nu\lambda_i)$, where ν is a hyperparameter controlling the mean perturbation magnitude. After the $2N$ policies have been applied to an episode each, the controller updates the maintained policy parameter vector w in the top b directions with largest return R. The policy update rule therefore has the form

$$w := w + \frac{\alpha}{b\sigma_r} \sum_{i=1}^{b} (R_{i,+} - R_{i,-})\delta_i \tag{4}$$

where σ_r is the standard deviation of the returns across the current batch of episodes[25], α is a hyperparameter controlling the update magnitude and $R_{i,+}$ is the returns of the policy $\pi(s; w + \nu\lambda_i)$. This update rule uses the gradient of the returns with respect to the policy parameters gradually move towards a better policy. In practice, $N = 1$ and $b = 1$ are sufficient to obtain fast convergence.

3.4.2 Deep Deterministic Policy Gradients (DDPG)

DDPG [18] is also a gradient-based learner, but is more elaborate than ARS. It stores all of the transitions it experiences, (s_j, a_j, r_j, s_{j+1}), in an 'experience replay buffer'. After each state transition, it updates the parameters of two functions, $\hat{Q}^\pi(s, a; \theta^Q)$ and $\pi(s; \theta^\pi)$, both of which are neural networks. $\hat{Q}^\pi$'s parameters are updated in a direction that reduces the mean sum-of-squares TD-error, the latter of which is estimated using a random sample of n transitions from the experience replay buffer. This update has the form[26]

$$\theta^Q := \theta^Q - \alpha \nabla_{\theta^Q} \frac{1}{n} \sum_{i=1}^{n} \delta_i (\hat{Q}^\pi)^2 \tag{5}$$

where α is a hyperparameter controlling update magnitude and the TD-error of the i^{th} sampled transition, δ_i, is a function of the current action-value approximator, $\hat{Q}^\pi$. The purpose of this update is to make $\hat{Q}^\pi \to Q^\pi$. The update described by Equation 5 corresponds to steps 2-3 in Figure 6a. Immediately after this update is applied, but before the next state transition, the controller updates the policy's parameters. The update rule is in a similar spirit to the one above: the policy's parameters are shifted in a direction that increases $\hat{Q}^\pi$ and will, hopefully, increase Q^π, provided

[24] Assume that only one action is generated here, but be aware that all of the controllers in this section can be extended to choose multiple actions per state transition.

[25] Normalizing by the s.d. of the returns is necessary to prevent the parameter update magnitudes from becoming larger as the controller's returns increase.

[26] In this equation and the one below it θ^Q refers to the parameters of the neural network, not to the set containing those parameters.

that $\hat{Q}^\pi \approx Q^\pi$. The policy update corresponds to steps 1-2 in Figure 6a and serves the purpose of making the agent's policy better. DDPG updates $\hat{Q}^\pi$ and π after each transition, possibly several times, until the policy's returns plateau.

To encourage DDPG to explore actions that it wouldn't choose under its current policy, noise is added to the actions it selects during training[27]. The process generating this noise must be chosen and tuned by the engineer. A further consideration is that DDPG computes the right-hand term of the TD-error of Equation 3 using stabilized versions of Q^π and π that are known as 'target networks'. These networks are updated using much smaller α values and essentially exist to prevent the learning process from diverging. Target networks are a prime example of an algorithmic tweak that attempts to prevent divergence but increases the time taken for a successful policy to be learnt.

3.4.3 Normalized Advantage Functions (NAF)

NAF [27] is similar to DDPG in several respects. The key difference is that NAF defines $\hat{Q}^\pi(s, a)$ as a neural network with its maximum value in each state at the action selected by the current policy, $\pi(s)$. This implies that $\hat{Q}^\pi$ will only produce zero TD-error if it is the action-value function of the optimal policy. NAF can therefore learn an optimal policy by updating the parameters of $\hat{Q}^\pi$ and π in the direction which locally reduces the mean sum-of-squares (s.s.) TD-error the most. The mean s.s. TD-error is estimated using a sample of past transitions from an experience replay buffer. Figure 6b visualizes how NAF's updates affect the controller's policy and action-values in a single state. Both the policy and the action-value function are altered in a single step. Apart from the way it collapses its policy update and action-value improvement into a unified update, NAF is similar to DDPG: it updates after every state transition, it relies on an experience replay buffer, exploration noise, and a target network, and randomly initializes its parameter values at the start of training.

3.5 Comparison

DDPG and ARS have several key differences that will be referred to frequently across the remaining sections of this report. Since DDPG and NAF share many features, the following comparison could equally well be applied to NAF and ARS. The differences of special relevance are that

- ARS updates its policy every $2N$ episodes, whereas DDPG applies an update after each state transition. If an episode contains T transitions, then DDPG applies $2NT$ updates per 1 ARS update. This has substantial consequences for the controllers' relative wall-clock runtimes.
- ARS is simpler than DDPG. It does not use target networks, exploration noise processes, replay buffers, or performance accelerators associated with neural networks[28]. This makes ARS easier to implement, test, analyze, and share with other engineers.
- DDPG uses TD-errors to modify an existing estimate of Q^π between policy updates. ARS estimates a policy's returns from scratch for each update using a Monte Carlo method. As the ensuing optimizer study shows, this is the main cause of DDPG's divergence problems.

Further differences that were minor considerations when considering the outcomes of the testbed study were that

- DDPG uses two neural networks, both of which may contain hundreds or thousands of parameters. ARS uses only a linear function, which has as many parameters as there are elements of the state vector. ARS's policy can be described verbally, whereas DDPG's cannot.
- The parameters of DDPG's neural network are usually randomly initialized. ARS specifies that w should be initialized as a vector of zeros, $\mathbf{0}$. This affects how consistently either controller converges.

[27] This isn't strictly necessary but is recommended by the original paper's authors.

[28] Performance-enhancing modifications to neural networks were not discussed in this section but are used in many RL papers. These techniques confer better performance at the expense of obfuscating the learning process.

3.6 Stage 2 (b) - Optimizer Selection/Divergence Study

This stage of work explored the optimization processes that limit the convergence speeds of DDPG and NAF. Its outcomes were twofold. First, it explained why DDPG's approach to executing step (1) of generalized policy iteration is unlikely to be practical in Year 5. By extension, the ensuing analysis precluded many other RL controllers built around a similar line of thinking, including NAF. The timing data gathered in Stage 2 (c) supports the explanations given. Second, this study validated the choice of optimizer used by DDPG and NAF in the experiments of Stage 2 (c) . This was accomplished by evaluating the results of an experiment benchmarking several optimization algorithms.

3.6.1 Background

DDPG and NAF have in common two optimization processes, both of which have been presented before in the context of generalized policy iteration. These are:

Process 1 Estimating Q^π for the current policy by minimizing the average sum-of-squares TD-error, $\sum_{i=1}^{n} \delta_i^2$, where δ_i is the TD-error defined in Equation 3 and n is the size of a sample of past state transitions.

Process 2 Maximizing $Q^\pi(s, a)$ indirectly by updating the policy π in a way that increases the value of $\hat{Q}^\pi$ across all states.

Process 1 allows the controllers to accurately estimate the returns on their current policy, whereas Process 2 updates that policy so that the controller will receive larger returns in the future. By alternating between these two steps, a successful policy can be learnt.

In DDPG, these processes correspond to unconstrained nonlinear programming problems: optimization problems characterized by objective functions that are nonlinear and possibly non-convex[29] in their parameters. The objective function of Process 1 is $\sum_{i=1}^{n} \delta_i^2$, a quantity that is reduced by altering the parameters θ^Q of the function $\hat{Q}^\pi$. The objective function of Process 2 is $\hat{Q}^\pi$, which can be increased by changing the parameters θ^π of the policy π. Since $\hat{Q}^\pi$ and π are parametrised as neural networks (nonlinear functions) in DDPG and NAF, Processes 1 and 2 are nonlinear programming problems. They must therefore be solved using an appropriate nonlinear optimization algorithm.

While these processes are formulated as unconstrained problems in DDPG, they are in fact subject to a hidden constraint: that the policy update must not cause learning to diverge. If the policy update is too big, then $\hat{Q}^\pi$ becomes out-dated and induces awry policy updates in the future. Target networks were originally introduced to avoid this problem [13], but are synonymous with slow, non-robust learning procedures. The experiment in this section emphasizes why the divergence constraint causes DDPG, NAF, and a host of other RL controllers [26] [26] [32] to learn slowly. In the process it highlights why a practically viable RL controller is unlikely to rely on a persistent estimate of the action-value function, Q^π.

3.6.2 Method

The rate at which DDPG and NAF can find a usable policy depends on the optimizer they use to improve their policy and action-value function. In this study, a selection of optimizers were screened for use in the testbed experiments. Three optimizers were ranked according to their ability to minimize the output of a function with respect to its input, as measured by wall-clock convergence time and the quality of the solution obtained after a fixed number of iterations. The objective function was non-convex and continuous, properties that made it similar to the objective functions

[29] A non-convex function has multiple peaks and valleys, features that can mislead an optimizer into believing that it has reached the global minimum of a function when instead it is stuck at a local minimum. Non-convexity is part of the reason why the convergence rate and final policy quality of a neural network can be very sensitive to how its parameters are itialised.

optimized within DDPG and NAF. The objective function was taken from the optimization literature[30] then engineered to fit the needs of the experiment. The resulting objective was

$$f(x_1, x_2) = \underbrace{(x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2}_{\text{Original objective}} + \underbrace{(x_1 - x_1^*)^2 + (x_2 - x_2^*)^2}_{\text{Paraboloid}} \quad : x_1, x_2 \in [-6, 6] \tag{6}$$

where x_1^* and x_2^* are scalar constants, x_1 and x_2 are scalar variables, and f is the objective function. The superimposed paraboloid causes f to have one global minimum at (x_1^*, x_2^*) and three local minima elsewhere. A contour plot of this function is shown in Figure 7a.

The optimizers selected for comparison were gradient descent [34], the low-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (LBFGS) [35], and the Nelder-Mead Simplex method (NMS) [36]. These were chosen on the basis that their update magnitudes differ and rely on different amounts of information about the function being optimized: NMS only needs to be able to evaluate the objective function, gradient descent requires the objective's local gradient with respect to its inputs, and LBFGS estimates the function's local curvature.

Each optimizer was tested by initializing it at the boundary of the function's domain then allowing it to carry out 20 iterations. The function's value was recorded before each update, as was the time taken to perform all 20 updates. 400 uniformly spaced initialization locations were used[31] to produce a data-set of 400 trajectories, each 20 iterations long, for each optimizer.

3.6.3 Results & Discussion

The results of this experiment are shown overleaf. Figure 8a contains histograms for the time taken to perform 20 iterations (i.e. complete a single trajectory). Mean trajectory times are shown in Table 7b and are plotted as dashed lines on the histograms. There was reason to think that the times measured on the test function might not accurately represent their relative performance when applied to larger functions such as deep neural networks. LBFGS and NMS evaluate the objective function multiple times per iteration, whereas gradient descent effectively only requires two evaluations[32]. The experiment was therefore repeated using a large function[33] instead of f.

Figure 8b contains a plot of the trajectories from all 400 runs of each algorithm against f. LBFGS converged on a local minimum within a single step and was stable there for the remaining nineteen iterations. This happened because the first step of LBFGS is equivalent to gradient descent with line search[34]. NMS, on the other hand, converged erratically, as can be seen by the tangle of curves in the red facet of Figure 8b. Gradient descent converged gradually. Each successive update it applied was smaller than the last, since the objective function's gradient approached zero moving towards its minimum.

On f, the mean runtimes for gradient descent and NMS were almost identical, whereas LBFGS's was three times larger. The timing results for the large function produced distributions similar to those of Figure 8a, except LBFGS's distribution was shifted such that its mean time increased three-fold, as the data in the second column of Table 7b confirms. In terms of overall wall-clock runtime to convergence then (i.e. weighting iterations by time taken), LBFGS converged over twice as quickly as gradient descent.

[30] The original objective is known as Himmelblau's function, named after the author that proposed it in his book on applied nonlinear programming in 1972 [33].

[31] NMS stores three points between updates. It was initialized by randomly generating two points at the function's boundary, then taking the third point as one of the 400 pre-specified locations.

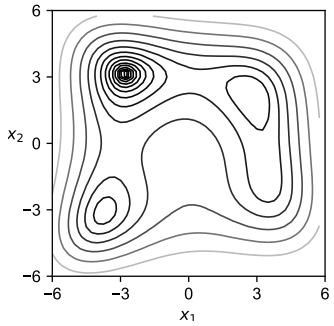
[32] Effectively two iterations. One forward evaluation, then one 'backward' evaluation to the compute the gradient of the output with respect to the inputs.

[33] An arbitrary fully-connected neural network with two inputs, two hidden layers containing 64 nodes each, and one output (≈ 5000 parameters).

[34] This means that f is minimized along the line of inputs that passes through the initialization location in the direction of the objective's gradient.

LBFGS may appear to be the most desirable choice of algorithm to apply in DDPG - it converged quickly to a local minimum in terms of both wall-clock time and number of iterations required. However, it was not suitable for use in DDPG because line search can produce updates that substantially alter the controller's policy. This risks causing divergence. The update size for NMS can be tuned, but its inconsistent update magnitudes are undesirable. Gradient descent produces comparatively small updates, each of which are applied quickly.

These results point to a fundamental constraint on the speed at which DDPG, NAF, and the other controllers previously cited[35], can learn. Large parameter update steps are necessary for fast optimization processes, since only a few of them need be applied, but can cause divergence when used in the context of generalized policy iteration, at least as it is realized by these controllers. Attempting to modify an estimate of the action-value function for one policy, $\hat{Q}^{\pi}$, such that it becomes a good estimate for another policy π', $\hat{Q}^{\pi} \rightarrow \hat{Q}^{\pi'}(s)$, will either cause learning to be slow, since policy updates must be small, or to fail entirely, since large policy updates initiate divergence.

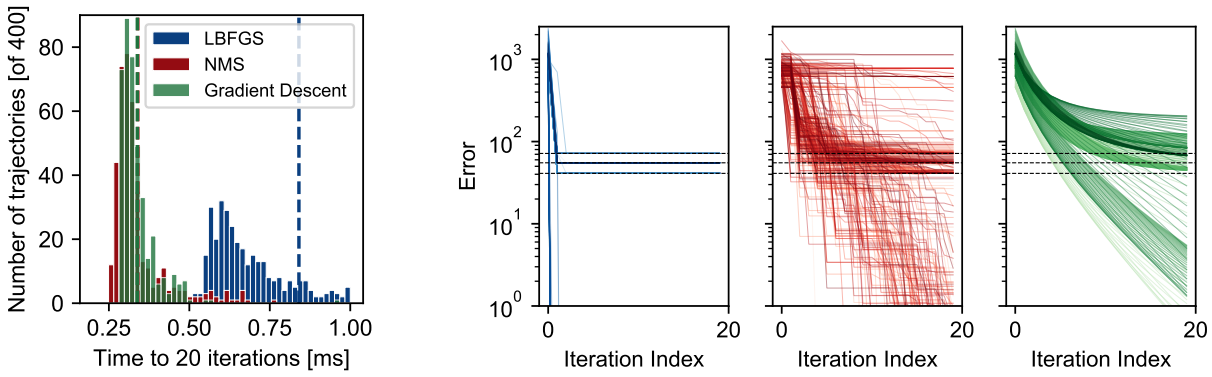


(a) Contour plot of f, the benchmark function. Darker lines correspond to a lower error. The function's global minimum is in the top-left hand corner[36].

	Average trajectory time [ms]	
Algorithm	Himmelblau	Neural network
LBFGS	0.84	2.77
Nelder-Mead Simplex	0.34	0.46
Gradient Descent	0.34	0.28

(b) Mean trajectory times for each optimizer by function optimized. The neural network takes longer to evaluate[37] than f because it requires more basic arithmetic operations to be performed.

Figure 7



(a) Histogram of the optimization algorithms' trajectory times.

(b) Trajectory curves for LBFGS, NMS, and gradient descent. The values of the local minima are displayed as dashed black lines. The y-axis is logarithmic.

Figure 8: Time histogram and convergence curves for the optimization study.

[35] excluding ARS.

[37] Gradient descent performed faster on the larger of the two functions. This was thought to be because the scientific computing Python package used, pytorch [37], optimizes how gradient descent is applied the package's neural network objects.

3.7 Stage 2 (c-d) - Testbed Evaluation of Intelligent Controllers

The work in this section evaluated the controllers selected in Stage 1 (d) on a testbed of continuous control tasks. The controllers' shortcomings were identified by applying the metrics specified in Stage 1 (c) to measurements taken across three different problem environments. This analysis informed the controller synthesis stage of work, where a viable intelligent controller was designed for the Year 5 system.

3.7.1 Method

The testbed consisted of three control environments taken from the reinforcement learning literature [38]. These environments are described in Table 3 and are shown in Figure 9. All of them are episodic with time horizons of $T = 400$. An episode of Pendulum terminated when $t = T$, whereas CartPole and MountainCar had conditions that allowed an episode to terminate prematurely. In CartPole's case, it was if the pole fell ($\psi > 0.26$); for MountainCar, if the car passed a flag at $x = 0.95$. The default reward functions for the problems were simplified to create a binary success/failure signal, r in $\{-1, 1\}$, a decision that will be explained in the Appraisal. The initial states of the pendulum and cartpole were randomly generated, whereas the car was always initialized with zero velocity at the bottom of the valley.

Table 3: Specification of testbed environments. ϕ is the pendulum's angle relative to the vertical. x is the car's horizontal location. y is the cart's horizontal location. ψ is the pole's angle to the vertical.

Environment	Description	Action	State	Reward
Pendulum	Swing-up & balance an under-actuated[38] pendulum.	Torque, $a = \tau$ τ in $[-1, 1]$	$s = (\cos\phi,\ \sin\phi,\ \dot{\phi})$ ϕ in $[-\pi, \pi)$	$r = \begin{cases} 1 & \text{if } \phi \text{ in } [-\frac{\pi}{4}, \frac{\pi}{4}] \\ -1 & \text{otherwise} \end{cases}$
MountainCar	Accelerate an underpowered car out of a valley.	Thrust, $a = \mathcal{T}$ $\mathcal{T}$ in $[-1, 1]$	$s = (x,\ \dot{x})$ x in $[0, 1]$	$r = \begin{cases} 1 & \text{if } x \geq 0.7 \\ -1 & \text{otherwise} \end{cases}$
CartPole	Balance a hinged pole on a laterally movable cart.	Lateral force, $a = F$ F in $[-10, 10]$	$s = (y,\ \dot{y},\ \psi,\ \dot{\psi})$ ψ in $[-\pi, 1]$	$r = \begin{cases} 1 & \text{if } \lvert\psi\rvert < 0.26,\ \lvert y\rvert < \frac{1}{4} \\ -1 & \text{if } \lvert\psi\rvert \geq 0.26 \end{cases}$

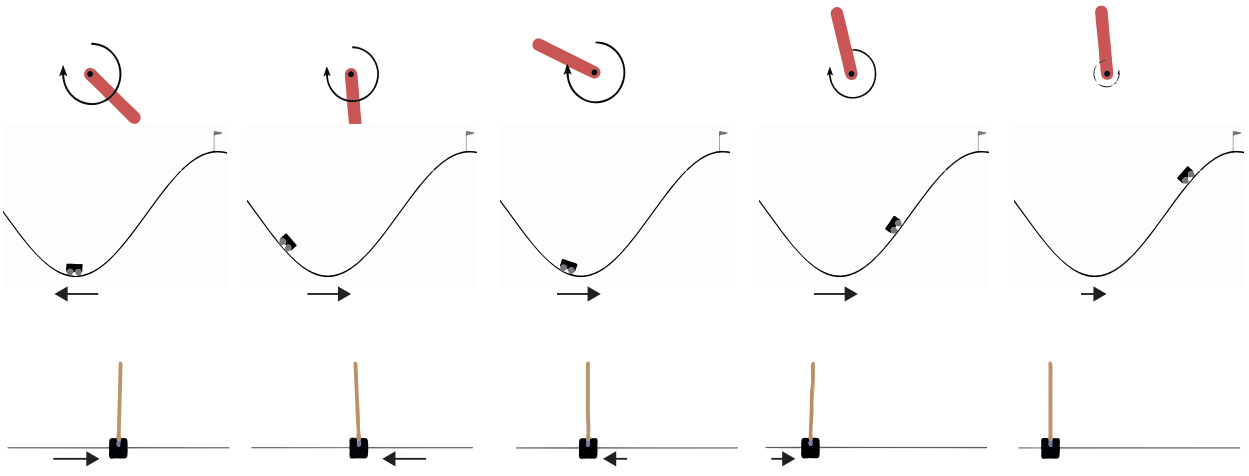


Figure 9: The control environments that constituted the testbed. Time develops reading from left-to-right.

[38] 'Under-actuated' means that the controller needed to learn to swing the pendulum back and forth, since it could not deliver a torque large enough to swing it up directly.

The experiment consisted of running the three controllers - DDPG, NAF, and ARS - against each of the environments for 550 episodes across 30 seeds. An episode contained 399 state transitions unless an early termination condition was met. DDPG and NAF were configured to perform 5 updates per state transition, whereas ARS applied an update per episode. The returns received in each episode were recorded, as was the time taken for the controller to perform each of its updates. NAF was terminated after only 10 seeds on the MountainCar environment since its slow runtime put the project's resource budget at risk.

The controllers were implemented in Python using scientific computing libraries. They were programmed according to descriptions from their original papers: DDPG, from a 2015 paper by Lillicrap et al. [18]; NAF, from a 2016 paper by Gu et al. [27]; and ARS, from a 2018 paper by Mania et al. [28]. The implementations of DDPG and NAF were checked by comparing their performance against open-source implementations of the controllers that were available on the web [39] [38] [40]. By inspecting the datatype and value of each line of code in the codebase produced, it was possible to gain further confidence that the controllers had been programmed correctly.

Each controller's hyperparameters were set to the values specified in the papers cited above. The hardware used to run the experiments was a Microsoft Azure Standard D4 v3 (4 vcpus, 16GB memory) instance, which is qualitatively similar in performance to a mid-range laptop in 2018.

3.7.2 Results & Discussion

The learning curves generated for each controller-environment-seed combination are shown in Figure 10. Figure 11a plots the cumulative fraction of controllers that reached a returns threshold[39] by a given episode, and Table 11b contains summary statistics for the quality and robustness of the learnt policies.

It was necessary to take a 20-point moving average of the returns to produce the displayed learning curves. This was because DDPG and NAF exhibited policy robustness issues after approaching maximum returns - on one episode they might achieve near-optimal returns, whereas on the next their performance would collapse and receive near-minimal returns. Even under aggressive averaging, the performance of DDPG and NAF can be seen to fluctuate spectacularly: this is especially true on DDPG×MountainCar and NAF×CartPole. ARS learnt robustly on both MountainCar and CartPole, for which its learning curves remained stable once they reached a near-optimal policy. ARS failed to learn to Pendulum however, a matter that will be discussed in the next section.

ARS learnt approximately one-hundred times faster than NAF and DDPG, as is shown by the text on the CartPole plots in Figure 10. Similar times were recorded when the controllers were applied to the Pendulum and MountainCar environments. This observation corroborates findings by Mania [28] in which ARS outpaces another RL controller, Trust Region Policy Optimization, that is functionally similar to DDPG and NAF.

DDPG and NAF did not converge consistently on MountainCar, a finding that is revealing. The initial state of this environment in each episode is fixed. Variance in the controllers' convergence can therefore be attributed to either or both of the stochastic processes that take place inside the controllers: sampling from the experience replay buffer, and initializing the parameters of the neural networks for $\hat{Q}^{\pi}$ and π. By contrast, ARS's performance on MountainCar is effectively uniform across all 30 of its trials, as shown by its overlapping learning curves. This indicates that although ARS generates its policy update directions randomly, the overall time to convergence is reasonably consistent. This same line of reasoning does not apply to the Cartpole environment however, where the controller's convergence is clearly much more sensitive to the perturbation directions that are generated.

[39] Approximately 95% of maximum returns.

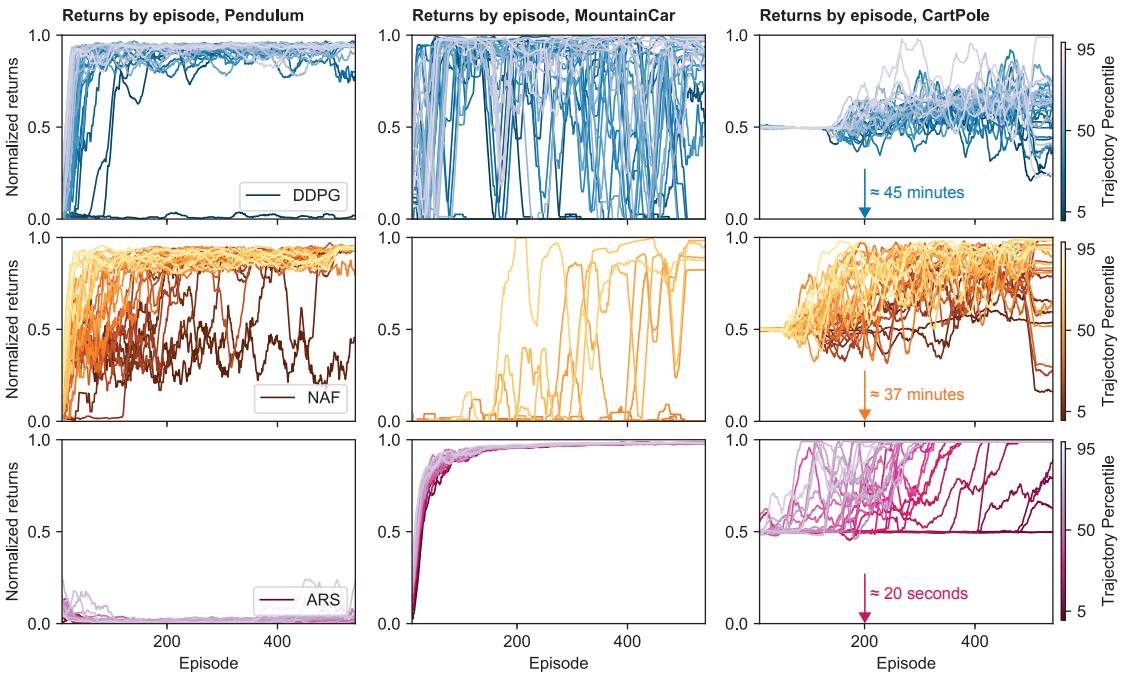
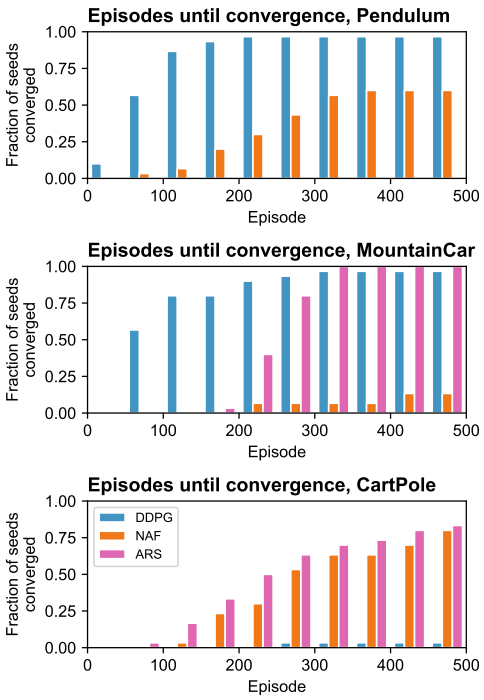


Figure 10: Testbed learning curves. Each curve corresponds to a single seed of controller. A 20-point moving average is applied to make the trends visible.



(a) Proportions of testbed seeds reaching a returns threshold[40].

Environment	Controller	Median avg. returns	Median s.d. of returns
Pendulum	DDPG	0.93	0.048
	NAF	0.90	0.064
	ARS	0.02	0.019
	ARS-rbf	0.85, 0.88	0.14, 0.10
MountainCar	DDPG	0.69	0.377
	NAF	0.00	0.036
	ARS	0.98	0.006
	ARS-rbf	1.04, 1.04	0.012, 0.013
CartPole	DDPG	0.62	0.143
	NAF	0.85	0.353
	ARS	1.00	0.001
	ARS-rbf	0.56, 0.91	0.051, 0.059

(b) Testbed data. The average (mean) and standard deviation of returns is computed across returns for episodes 300-550. The second set of results for ARS-rbf are computed for episodes 800-1000.

Figure 11

(3.7.2 continued) Figure 9 shows that DDPG had superior sample complexity relative to NAF and ARS on Pendulum and MountainCar, but scarcely learnt a usable policy on CartPole. While a controller should ideally be able to learn a useful policy using as little experience as possible, sample complexity is most meaningful when considered in the context of wall-clock learning time. While at least 50% of DDPG's seeds had reached the returns threshold within 100 episodes on both Pendulum and MountainCar, those 100 episodes required about 23 minutes per controller. ARS, on the other hand, required ≈ 300 episodes of experience to meet the same 50% mark, but did so in about 30 seconds per controller (i.e. forty-six times faster).

The arguments presented in this section are borne out by the numerical data in Table 11b: ARS learns near-optimal policies that perform consistently (i.e. have returns with a low s.d.), whereas DDPG and NAF are capable of learning on Pendulum but typically have less reliable returns (have a low policy robustness).

[35]These plots only measure how many iterations it took for a policy that met the threshold to be learnt: they do not provide any insight into the stability of the learnt policies.

3.8 Stage 3 (a-c) - Solution Generation & Validation

The results of Stage 2 indicated that DDPG and NAF trained slowly, learnt inconsistently, and did not learn policies that were robust. ARS, meanwhile, was unable to solve the Pendulum environment. The purpose of Stage 3 was to develop a controller that could solve environments similar to Pendulum without requiring lengthy training times or suffering from robustness issues. This section explains why ARS was unable to solve the pendulum problem and introduces an alternative policy representation that allowed it to do so. The proposed design enables the ARS controller to solve a larger class of environments while retaining its fast training time.

3.8.1 Stage 3 (a) - Controller Synthesis

ARS uses a linear policy representation: given a state s_t, the controller generates an action a_t according to[41]

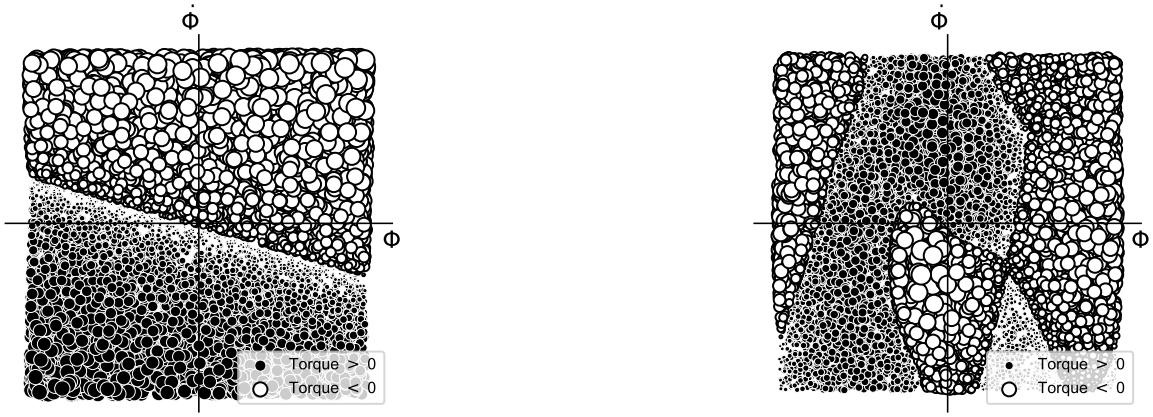
$$a_t = \pi(s_t; \theta^\pi = \{w_1, ..., w_n\}) = \sum_{i=1}^{n} w_i s_{t,i} = w^T s_t \tag{7}$$

where w_i are scalar parameters of the policy π and $s_{t,i}$ is the i^{th} component of the n-dimensional state at time-step t. w_i is therefore the rate of change of the controller's action with respect to the i^{th} state component.

ARS's learning algorithm must find the values of w_i that maximize returns. Plotting the action against the state components, Equation 7 demarcs a plane that passes through the origin and has a surface normal in the direction w. By using a linear policy, an engineer assumes that the environment's optimal policy can be well-approximated by this plane. If the environment's optimal policy is nonlinear, then the policy surface will be bowl-shaped, jagged, or possibly wiggly, making it difficult to find values of w_i that produce large returns.

Figure 12a is a plot of one of the policies learnt by ARS on Pendulum: compare it to Figure 12b, which is a plot of an approximately optimal policy for Pendulum. For ARS to perform, it would have needed to orient the plane described by its policy such that it approximated the bumpy surface shown on the right. This is clearly a difficult thing for it to do: no 'good' orientation of the plane exists. In the Year 5 system the intelligent controller may be required to solve problems that have nonlinear optimal policies similar to Pendulum's. The controller described below preserves the simplicity and speed of ARS, but uses a policy function that can approximate optimal policies for nonlinear control problems.

[41]For simplicity, assume that the action vector has a single element. The results of this section can be extended to the p-action setting by replacing w with a matrix $W \in \mathbb{R}^{p \times n}$.



(a) A linear policy learnt by ARS on the Pendulum environment. Plotting torque as height, this surface is an inclined plane. It cannot be oriented so that it accurately approximates a policy like that plotted in Figure 12b.

(b) A nonlinear policy that solves the pendulum environment. Φ corresponds to the pendulum's angle relative to the vertical and $\dot{\Phi}$ is its angular velocity. The size of each data-point is proportional to the torque magnitude applied by the policy.

Figure 12

The key lies in changing the policy function in a way that preserves the simplicity of Augmented Random Search. Radial basis functions (rbfs) have been shown to deliver good performance on nonlinear control problems, but have only previously been applied in the context of complex learning algorithms [21]. An rbf returns a scalar value that depends on the distance of its input from some value c. An example of an rbf is the exponential kernel,

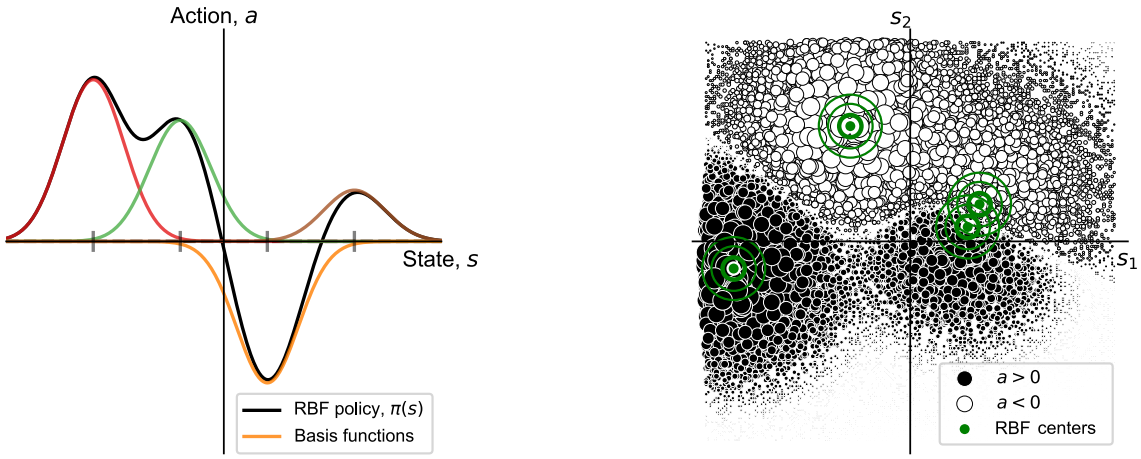
$$\xi(s;\sigma_l,c) = \exp\Big(-\frac{\|s-c\|_2^2}{\sigma_l^2}\Big) \tag{8}$$

where σ_l is a scalar, c is a vector of the same size as s, and $\|\cdot\|_2$ is the Euclidean norm. For $s \approx c$, $\xi(s)$ is approximately equal to 1. As s moves away from c, the rbf's output tends to 0 at a rate that depends on σ_l. $\xi(s)$ can therefore be thought of as indicating when s is in the region of state-space surrounding c. The intelligent controller proposed in this project replaces ARS's linear policy with a weighted sum of rbfs. The revised policy is therefore

$$\begin{aligned} a = \pi(s) &= w_1\xi_1(s) + w_2\xi_2(s) + ... + w_n\xi_k(s) \\ &= w^T\xi(s) \end{aligned} \tag{9}$$

where $\xi(s)$ is now a vector $\big(\xi_1(s),\ ...,\ \xi_k(s)\big)$ and $\xi_i(s) : i = 1, ..., k$ are k exponential rbfs, each with unique centers c_i and common length scale $\sigma_1 = ... = \sigma_n = \sigma$.

An example policy of the type described by Equation 9 is plotted in Figure 13a. Each colored curve corresponds to a unique basis function, $\xi_i(s)$. The curves' heights are equal to their associated weights w_i. The black line is produced by summing the weighted basis functions together to produce $\pi(s)$. Figure 13b shows the kind of policy surface that can be generated for environments with two states. As the number of basis functions increases, so too does the complexity of the surfaces the policy can represent.



(a) A one-input, one-output rbf policy.

(b) A two-input, one-output rbf policy.

Figure 13: Examples of rbf policies.

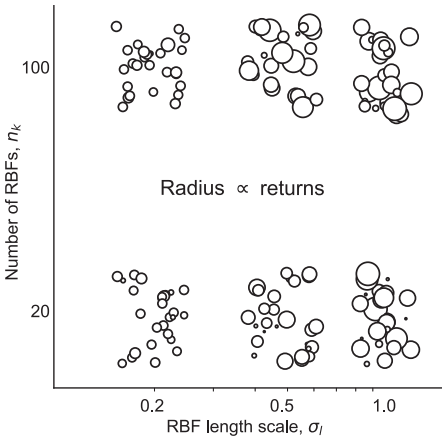
Replacing ARS's linear policy with an rbf policy is simple: instead of searching for the best weights to apply to the elements of the state vector, the controller seeks out what weights to apply to a set of basis functions. The centers of the basis functions are randomly initialized at the beginning of training[42], and the rbf width is specified as a hyperparameter.

3.8.2 Stage 3 (b) - Hyperparameter Sensitivity Study

To determine how the performance of the proposed controller - referred to as ARS-rbf - was affected by the number of basis functions used and the chosen length scale, a sensitivity study was run. 3 seeds of the controller were trained against the Pendulum environment for 500 episodes across the 162 different hyperparameter settings defined by the grid of values listed in Table 14b. After 500 episodes of training, each controller's parameters were frozen and its returns across a further 25 episodes were averaged. The median and mean of this average across the 3 seeds were then computed. The controller's runtime was also measured to verify that it was similar to ARS's.

The results of this study are shown in Figure 15b. The controller's learning curves are shown in the first facet of Figure 14a and will be discussed in the next section alongside the validation study's results. The large plot in Figure 14a highlights that an appropriate parameter configuration was crucial to achieving acceptable performance. ARS-rbf's returns were most sensitive to three of the six hyperparameters modulated: the rbfs' length scale σ, ARS's perturbation magnitude ν, and ARS's update scale α. As can be seen in Figure 14a and the green facet of Figure 15b, the controller's returns scaled rapidly with the rbf length scale and weakly with the number of rbfs. Small length scales mean that more rbfs are needed to cover an environment's entire state-space. Larger length scales allow fewer rbfs to be used to produce a non-zero action across the entire space, at the expense of lower-resolution control over the policy surface's shape.

[42] Using uniformly distributed basis functions may produce better performance. Initializing the policy's parameters with random values also injects variance into the controller's learning rate, which is undesirable. Future work should investigate alternative basis function initialization schemes.

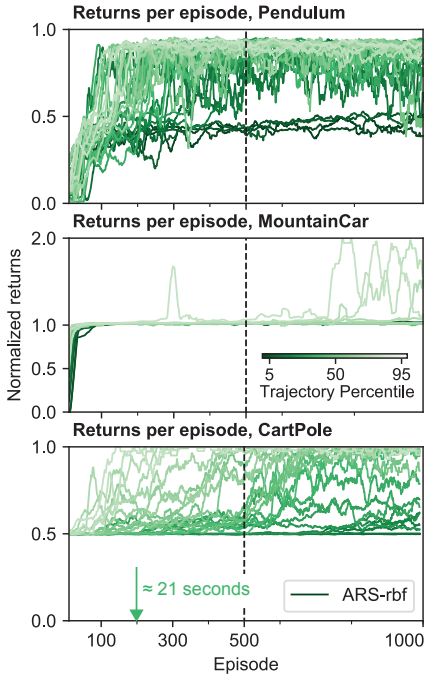


(a) RBF returns on Pendulum as a function of number of rbfs / rbf length scale.

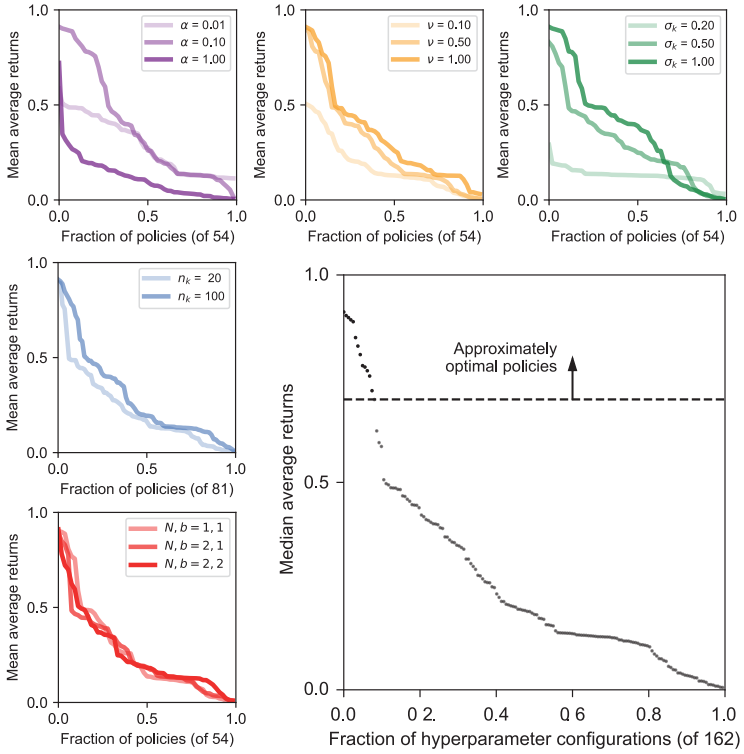
Hyperparameter	Values
Number of rbfs, n_k	20, 100
Rbf length scale, σ_k	0.20, 0.50, 1.00
ARS update scale, α	0.01, 0.10, 1.00
ARS perturbation scale, ν	0.10, 0.50, 1.00
ARS perturbation directions, N	1, 2
ARS retained directions, b	1, 2

(b) The set of hyperparameter values that were permuted in the sensitivity study.

Figure 14



(a) RBF returns on each of the testbed tasks. Each curve corresponds to a separate seed.



(b) Hyperparameter sensitivity study results.

Figure 15: ARS-rbf controller sensitivity and validation study data.

The experiment's results indicated that a middling parameter update size α produced better performance. This may have been because a large step-size was excessively sensitive to the effects of environment stochasticity on returns. In this case, the pendulum's initial state was randomly generated at the beginning of each episode: a favourable/unfavourable initialization pairing for perturbations[43] $+\nu\delta$ and $-\nu\delta$ might have produces a misleading returns gradient,

[43] It may be useful to refer Section 3.4.1 here.

causing the controller to update in a way that rubbished the parameter improvements of preceding updates. By contrast, too small an update length scale may have meant that the controller searched the parameter space too slowly and failed to approach an optimal policy before reaching episode 500.

3.8.3 Stage 3 (c) - Testbed Validation

Having determined a set of hyperparameter values that produced acceptable performance on the Pendulum environment (α=0.1, ν=1, σ_k=1, n_k=100, N=1, b=1), the ARS-rbf controller was evaluated against the remaining two testbed tasks. The learning curves generated by these experiments are shown in Figure 15a. These curves demonstrate that the controller was capable of learning successful policies on all three of the problem environments, validating the conjecture that an rbf policy would allow ARS to succed in solving the pendulum swing-up problem. The metrics in Table 11b indicate that ARS-rbf achieved similar mean returns to DDPG and NAF on Pendulum, albeit with policies that had a higher variance in returns. On MountainCar and CartPole, ARS-rbf produced substantially better policies, both in terms of the returns' mean and variance. A surprising result among the MountainCar seeds was that two of the controller's seeds discovered a policy that almost doubled their return relative to the maximum earned by any of the controllers from Stage 2. They did this by learning to park the car in front of the termination flag. Some of ARS-rbf's policies failed on Pendulum, an outcome that was likely caused by an unfavourable set of rbf locations.

3.8.4 Stage 4 (a) - Validation of Controller w.r.t. Performance Metrics

The validation study suggested that ARS-rbf is a compromise. It matched the performance of DDPG and NAF on nonlinear control tasks, but several initializations were required before it could find a robust solution[44]. In a commercial context, multiple initializations may be feasible: the controller could be trained on the testbed environments in between 10 seconds and 1 minute, almost as little time as it would take to train ARS. All thirty of the seeds shown in Figure 15a were trained in less time than it took to train a single seed of DDPG or NAF, highlighting that

- persistent estimates of $\hat{Q}^{\pi}$ limit the rate at which a controller can converge, as was explored in the optimizer study of Stage 2 (b),
- the frequent updates associated with TD-error-based learning are much more time-consuming than the intermittent ones applied by Monte-Carlo methods.

ARS-rbf is simpler than DDPG and NAF, both in terms of implementation complexity and explanation complexity. This is another benefit that it carries over from ARS.

[44] Robust in the sense that the variance in returns across several episodes is small.

3.9 Stage 4 - Appraisal

This section highlights ARS-rbf's limitations, specifies how the controller could be used in the Year 5 system, and suggests how the project's outcomes could be developed in future research.

3.9.1 Stage 4 (b) - Limitations of the Proposed Controller & Suggestions for Future Work

The controller performed well on the testbed environments, but suffered from limitations that would hinder its deployment in a commercial system. It required several hundred episodes' worth of experience before it could learn an optimal policy for a simple task. In practice, the controller would need to learn in simulation. This could either be provided beforehand, for example in the form of a physics engine, or learnt from observations. RL controllers that learn an internal model of the environment generally have much better sample efficiency than the model-free variants that were considered in this project[45] [14] [10]. They represent a promising avenue of enquiry for future work.

Another shortcoming of the current control scheme is that the environment's reward function was specified by an external agent. The implications of this for the Year 5 project were considered when designing the testbed, which used only binary rewards over a closed region of state-space (as opposed to complex rewards that had been manually tuned by other researchers). Simple binary rewards could arguably be generated automatically by a machine vision system or manually via a user interface.

The controller provided no guarantees on its robustness or on its ability to generalize to new environments, or even similar environments under different reward functions. In Year 5, the controller will likely have to respond to changes in its goal[46]. One way of achieving this without necessitating retraining might be to re-express the learning problem: extend the state vector to include a goal state s_g as well as the current state s_t, then define the reward function as returning +1 if $|s_t - s_g| < \epsilon$, where ϵ is a small vector, and -1 otherwise. That way the controller would need to learn to transform its policy depending on the current goal state, and could be more successful in generalizing what it had previously learnt. A similar idea has been applied successfully to estimating $\hat{Q}^\pi(s, s_g, a)$ in the past [17].

Aside from its evident limitations, there remain many unanswered questions regarding the controller's performance. This project did not consider advanced obstacles to learning such as sparse reward functions, credit assignment issues, and incomplete observations[47]. Nor did it address the issues associated with high-dimensional state and action spaces, which are known to degrade an RL controller's performance, or hierarchical tasks, where a problem must be solved at several levels of abstraction[48]. While these extensions to the basic problems considered here are of research interest, simpler shortcomings of the proposed controller should first be resolved. It needs to be evaluated on a larger range of environments, and statistics for predicting its performance on unseen environments need to be developed, as do comprehensive robustness tests. Learning curves are also a crude measure of performance, since performance is unlikely to be directly proportional to a controller's returns. In many cases, a controller is unacceptable unless it can consistently achieve maximal returns.

[45] RL controllers that learnt a model of the environment were considered out-of-scope for this project, as they would have complicated the experiments and report.

[46] For example, an alternative goal on MountainCar would be to drive the car up the left hill instead of the right one.

[47] An environment with a sparse reward function only generates non-zero reward on a tiny fraction of state-space (e.g. a checkmate position in chess). The credit-assignment problem occurs when a controller receives reward that lag its state transitions and actions - it is not clear which state transition was responsible for the reward. An observation is incomplete if it cannot be used to fully describe the state of the environment

[48] Hierarchical task representations come naturally to human beings. 'Make a pancake', for instance, consists of 'Whisking pancake mix' and 'Frying panfuls of mix', tasks which can be further decomposed still.

3.9.2 Stage 4 (c) - Applications of this Project's Outcomes in the Year 5 System

Specific applications of the controller in a home care system were not explored directly. The benefit of this is that the project's outcomes can be applied to problems across many sectors of engineering and business. The controller could be used to minimize power usage in a building's ventilation system, maximize the quality or throughput of a manufacturing process, or optimize the gait of a locomotive robot. At a process level, the controller is a useful tool for rapidly prototyping new behaviours. It could be used during the design process as well as in a final product, offering a lightweight alternative to existing controller design packages, particularly if integrated into a graphical application that gamified the design process.

Whether learning occurs in advance or in situ in the Year 5 product, the controller could be applied to either internal operations or user-facing behaviours. In the context of the systems developed by other members of the Project group, it is likely to perform planning-type operations: generating trajectories for the inverse kinematics model, destinations and routes for the SLAM system, and user feedback signals for the human-robot interface. The controller could also be combined with the contact-detection system to learn actuation policies that are safe, and the fall-monitor's accelerometer to help a user retain their balance.

4 Conclusions & Recommendations for Future Work

Autonomous learning is necessary if a home-care system is to help someone recover their physical independence, particularly if that system takes the form of a robot. This project developed a sound basis for a commercially viable intelligent controller. The proposed design expanded the class of problems that could be solved by an existing system, but did so in a way that preserved its $\times 50$-150 learning rate relative to modern alternatives. The final design was relatively simple, making it attractive to businesses and non-specialist engineers that are disconcerted by the complexity of current offerings for autonomous learning. The optimizer study and associated evaluation showed that an idea behind many of the RL controllers developed within the last four years - maintaining persistent estimates of Q^π between policy updates - is a constraint that limits the rate and consistency with which a successful policy can be learnt. The analyses contained in this report suggest that simpler, more robust methods similar to those applied by Augmented Random Search are more likely to lead to intelligent controllers that are better-aligned with practical needs.

The Project Objectives provided a useful framework for structuring the controller's development. They specified that the requirements of the Year 5 system should be analyzed, distilled into a set of performance metrics, and used to evaluate controllers from the literature: this was achieved. The associated work revealed that many of the metrics pertinent to a learning system's commercial relevance - convergence time, robustness at convergence, and generalization capacity - were either absent or rarely mentioned in journal papers. The project's later stages successfully explored how neglecting these performance measures may have contributed to the popularity of an impractical control abstraction, and redressed an important shortcoming of a faster, simpler alternative.

A range of practical problems will be encountered in Year 5 that were avoided in this project by using idealized learning environments. The controller was flexible because it had many free variables that needed specifying. In the Year 5 product, it will be essential to produce a rigorous set of performance requirements for the learning system and give careful thought to how best to express the system's learning problems. Very few case studies describing how to integrate an intelligent controller into a commercial product exist, meaning that the Year 5 Design Project is an opportunity to pioneer good design practice in this respect.

Design is a necessary component of successful product development. Reinforcement learning is a field that might be criticised for neglecting practical considerations and failing to provide adequate tools for exploring solutions. If future

efforts towards commercial reinforcement learning are to be successful they will need to place greater emphasis on design process and experimental practice as opposed to technical detail and brute computation.

References

[1] Office for National Statistics. Disability in england & wales: 2011 and comparison with 2001. 2011.

[2] National Audit Office. Adult social care in england: Overview. 2014.

[3] NHS Information Centre. Survey of carers in households. 2010.

[4] Lucinda Beesley. Informal care in england. *Wanless social care review*, 2006.

[5] UK-RAS Network. Robotics in social care: A connected care ecosystem for independent living. 2017.

[6] Unimation, the first industrial robot. https://www.robotics.org/joseph-engelberger/unimate.cfm. Accessed: 05/12/2017.

[7] SemanticScholar, query term 'machine learning'. https://www.semanticscholar.org/. Accessed: 11/04/2017.

[8] IEEEExplore, query term 'control'. https://ieeexplore.ieee.org. Accessed: 11/04/2017.

[9] R.E. Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 2003.

[10] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. A Bradford book. Bradford Book, 1998.

[11] Andrzej Jezierski, Jakub Mozaryn, and Damian Suski. A comparison of lqr and mpc control algorithms of an inverted pendulum, 07 2017.

[12] H.M. Power and R.J. Simpson. *Introduction to Dynamics and Control*. McGraw-Hill Electrical and Electronic Engineering Series. McGraw-Hill, 1978.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[14] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *I. J. Robotics Res.*, 32:1238–1274, 2013.

[15] Dylan Hadfield-Menell, Stuart J Russell, Pieter Abbeel, and Anca Dragan. Cooperative inverse reinforcement learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3909–3917. Curran Associates, Inc., 2016.

[16] Kun Li and Joel W. Burdick. Large-scale inverse reinforcement learning via function approximation for clinical motion analysis. *CoRR*, abs/1707.09394, 2017.

[17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *Computing Research Repository*, abs/1509.02971, 2015.

[19] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[20] Paul W. Glimcher. Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis. *Proceedings of the National Academy of Sciences*, 108(Supplement 3):15647–15654, 2011.

[21] Aravind Rajeswaran, Kendall Lowrey, Emanuel V. Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6550–6561. Curran Associates, Inc., 2017.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[24] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017.

[25] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *ICML*, 2016.

[26] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[27] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *ICML*, 2016.

[28] H. Mania, A. Guy, and B. Recht. Simple random search provides a competitive approach to reinforcement learning. *ArXiv e-prints*, March 2018.

[29] Shixiang Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. *CoRR*, abs/1610.00633, 2016.

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[31] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[32] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger B. Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017.

[33] D. Himmelblau. *Applied Nonlinear Programming*. McGraw-Hill, 1972.

[34] A. Cauchy. *Méthodes générales pour la résolution des systémes d'équations simultanées.* Comptes Rendus de l'Académie des Sciences, 1847.

[35] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, second edition, 2006.

[36] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[37] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[38] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[39] Github, ikostrikov's ddpg implementation.

[40] Emami, p. https://github.com/fchollet/nelder-mead/blob/master/nelder_mead.py. Personal blog item, 'Deep Deterministic Policy Gradients in TensorFlow'. Accessed: 19/03/2017.