Guided policy search for a lightweight industrial robot arm

Jack White

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology
Espoo, 16th October 2018

Thesis supervisor:
Prof. Ville Kyrki

Thesis adviser:
Jens Lundell

Aalto University
School of Electrical
Engineering

Funded by the
Erasmus+ Programme
of the European Union

Aalto University
School of Electrical
Engineering

Aalto University, P.O. Box 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Jack White

Title Guided policy search for a lightweight industrial robot arm

Degree programme MSc Space Science and Technology (Spacemaster)

Supervisor Prof. Ville Kyrki

Advisor Jens Lundell

Date October 16, 2018 **Number of pages** viii + 56 **Language** English

Abstract

General autonomy is at the forefront of robotic research and practice. Earlier research has enabled robots to learn movement and manipulation within the context of a specific instance of a task and to learn from large quantities of empirical data and known dynamics. Reinforcement learning (RL) tackles generalisation, whereby a robot may be relied upon to perform its task with acceptable speed and fidelity in multiple—even arbitrary—task configurations. Recent research has advanced approximate policy search methods of RL, in which a function approximator is used to represent an optimal policy while avoiding calculation across the large dimensions of the state and action spaces of real robots. This thesis details the implementation and testing, on a lightweight industrial robot arm, of guided policy search (GPS), an RL algorithm that seeks to avoid the typical need, in machine learning, for lots of empirical behavioural samples, while maximising learning speed. GPS comprises a local optimal policy generator, here based on a linear-quadratic regulator, and an approximate general policy representation, here a feedforward neural network. A controller is written to interface an existing back-end implementation of GPS and the robot itself. Experimental results show that the GPS agent is able to perform basic reaching tasks across its configuration space with approximately 15 minutes of training, but that the local policies generated fail to be fully optimised within that timescale and that post-training operation suffers from oscillatory actions under perturbed initial joint positions. Further work is discussed and recommended for better training of GPS agents and making locally optimal policies more robust to disturbance while in operation.

Keywords guided policy search, reinforcement learning, deep learning, machine learning, kuka, control engineering, optimal control, policy search, supervised learning, robotics, artificial intelligence

Acknowledgements

Thank you to Professor Ville Kyrki and to Jens Lundell of the Intelligent Robotics group at Aalto University, through whose tutelage this thesis has been accomplished, and also to Annika Salama and the other departmental administrators at Aalto University.

Thanks also to Dr Victoria Barabash at Luleå University of Technology, Sweden, and Professor Klaus Schilling at the University of Würzburg, Germany, for their academic organisation of the SpaceMaster degree, and also to the respective departmental administrators.

Many, many thanks to all the lecturers and teaching assistants at the three universities, especially for putting up with my often very demanding personality and for acceding to my requests more often than not. They have made a difference to the whole class.

Credit unsuitable for the reference list must go to Jan Peters and Gerhard Neumann of Technische Universität Darmstadt for their excellent presentation on policy search, which goes far beyond my needs and comprehension, but was nonetheless a much-needed introduction to the subject space.

Thank you to my parents, my uncle John White and my cousin Tom White for the support, particularly the vital pecuniam dedisset.

Finally, thanks and regards to the European Union, the best non-party political project going, and to the Eramus+/Erasmus Mundus Joint Masters programme.

This project has been funded with support from the European Commission.

This publication, *Guided policy search for a lightweight industrial robot arm*, reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Initialisms

Initialism	Meaning
ACE	Adaptive critic element
AI	Artificial intelligence
ANN	Artificial neural network
ASE	Associative search element
BADMM	Bregman Alternating Direction Method of Multipliers
DDP	Differential dynamic programming
DP	Dynamic programming
FRI	Fast Research Interface
GMM	Gaussian mixture model
GPS	Guided policy search
GUI	Graphical user interface
iLQR	Iterative linear-quadratic regulator
LQR	Linear-quadratic regulator
LQG	Linear-quadratic Gaussian
LWR	KUKA Lightweight Robot
LBR	Leichtbauroboter (see LWR)
MC	Monte Carlo (method)
MDP	Markov decision process
ML	Machine learning
NN	Neural network
PID	Proportional-integral-derivative
RL	Reinforcement learning
ROS	Robot Operating System
URDF	Universal Robotic Description Language
XML	Extensible Markup Language

Symbols

Symbol	Meaning
T	Total number of time steps
t	Current step
$p(z\|x, y)$	Probability of z given x and y
$p(\mathbf{x}\prime\|\mathbf{x}, \mathbf{u})$	Transition dynamics
$r(\mathbf{x}\prime, \mathbf{u}, \mathbf{x})$	Reward function
$\mathbf{x}$	Robot state
$\mathbf{u}$	Robot action
$\pi(\mathbf{x})$	Deterministic policy
$\pi(\mathbf{u}\|\mathbf{x})$	Stochastic policy
$v(\mathbf{x})$	Value function, aka state-value function
$q(\mathbf{u}, \mathbf{x})$	Action-value function
$v*, q*$	Optimal value functions
$\mathbb{E}$	Expectation
R	Return; sum of rewards along a trajectory; negation of cost
λ	Discount rate
$\phi(\mathbf{x})$	Function approximator feature vector
$\mathbf{w}$	Function approximator feature weight vector
α	Gradient descent step size
$g(\mathbf{x})$	Deterministic local optimal policy
$\pi_{\mathcal{G}}$	Stochastic trajectory distribution
a	Neuron activation function
c	Step cost
$\circ$	Hadamard, or element-wise, product of vectors
w_p	Weight given to final position cost
$\mathbf{e}$	Vector of robot final position errors

Contents

Section 1

Introduction

1.1 The automated world and artificial intelligence

Machines—labour-saving devices—are a hallmark of human civilisation. Even the earliest tool, such as a simple stone edge, is a machine that increases the pressure caused by the force of a human hand-strike.

Since the late 18th century, humans have expended vast fortunes in pursuit of industrial automation. Industrialisation resulted in rocketing productivity in, for example, England during the Industrial Revolution (1760-1840), Japan (1870-1910) and Russia(1928-40).

The mid-twentieth century brought cataclysmic war and bitter intercontinental rivalries. In this febrile and unsavoury atmosphere, the power of machines was given primary importance by advanced and advancing economies alike. Mass factory production became necessary to sustain both military and civilian needs.

Guided by the minds of futurists and fiction writers during the Golden Age of Science Fiction (late 1930s and 1940s), and under Cold War economic pressures, engineers developed the first robots—machines capable of performing sequences of task without regular human intervention. Unimate became the first industrial robot in 1961[1]. Since that time, robots have become common in industrial contexts (see Fig. 2).

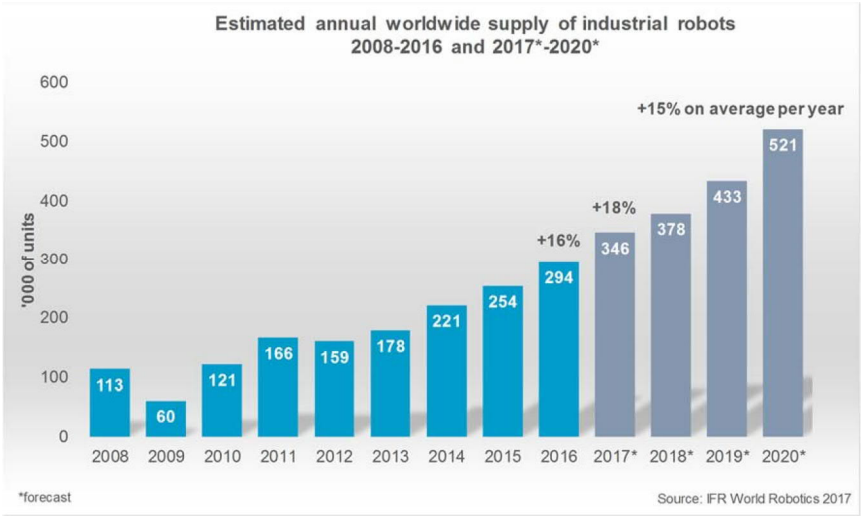


Figure 2: Robots are a growing concern in industrial contexts. In its 2017 industrial report World Robotics, the International Federation of Robotics suggests that, while the automotive industry has led in use of robots, the increasing human reliance on electronics means that this industry is seeing rapid growth in the number of robots.[2]

In recent decades, roboticists' attentions have, to some extent, turned away from the essential mechanisation and computerisation required to build and operate automatons, which rely upon a largely set programme of instructions, towards intelligent robots, able to react to their operational environments and determine for themselves best courses of action. This paradigm springs from the general field of artificial intelligence (AI).

Artificial intelligence, a term coined in 1955 by one of its trailblazers John McCarthy, broadly seeks to instil in machines an ability, similar to animals, to perform tasks based on abstract thinking. Robotics, with its fundamental place in science fiction in which machines can reason and act like a human being, has shared much history with AI. Certain aspects of AI have come to prominence in robotics in the last 20 years and it is informative to get an overview of three areas of AI that will be used in this thesis.

1.1.1 Supervised learning

The first two aspects of AI covered here are two related areas of machine learning, itself a major field of artificial intelligence. In supervised learning, a machine learns a hypothetic function that maps an input to an output. It is a function of a set of internal parameters and predictive features of the class of inputs[3].

During the learning process, the machine repeatedly is fed a series of training data, i.e. input/output pairs. It passes the training input to its hypothetical function and then compares the derived output with the example output and adjusts the set of internal parameters to make the function more accurately map input to output. In this way, the training dataset *supervises* the learning process, telling the learning agent when it is wrong, or when it right.

Supervised learning is commonly used as a machine pattern recognition tool in two important classes of task: classification and regression. For example, one might use supervised learning to train a machine to assert whether a photograph was taken at night or in daylight. This is classification because the output is that the input falls discretely in either the class of daytime photos or the class of nighttime photos.

Regression on the other hand maps inputs to continuous outputs. An example of a regressive supervised learner is an an agent learning to predict temperature from landscape photographs. It is regression that is used in guided policy search, the topic of this thesis.

Although both classification and regression seek to give definitive answers, the reality is that the answers will be given as correct, to a certain expected Bayesian probability. For example, the daytime/nighttime classifier might give results that there is a 90% probability that a particular photo was taken at night.

1.1.2 Reinforcement learning

Reinforcement learning (RL) grew out of supervised learning to cover a need for an ability for a machine to learn to function in processes, rather than series of single cases. The functional mapping in RL is between the situation the agent find itself in and an action that it should take to further the process. Instead of having a supervisor to tell it when it is successful, or how successful it is, the agent learns by being rewarded or paying a cost for its actions. The learning process is then a matter of minimising the costs (maximising the rewards) accumulated over the course of the process.

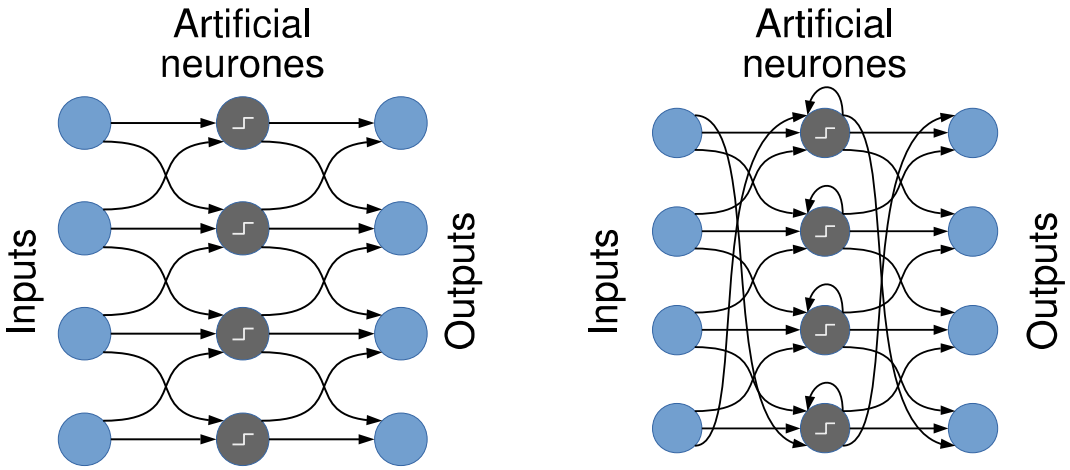
An important distinguishing feature of RL is that the feedback the agent receives during the learning process is incomplete. This marks RL as seperate from supervised learning, where the agent is told whether or not it is right. While in supervised learning the successful agent learns to extrapolate from the *correct* examples it is given, the RL agent is only rewarded on a step-by-step basis and may have only a limited view of the process—incomplete state observations. Its actions may have far-reaching consequences for its long-term performance in the process, of which the reward given now is unrepresentative [4].

1.1.3 Neural networks

From the early days of AI, efforts have been made to replicate the process of the animal mind in computers. Nowhere more obviously has this idea appeared than in the neural network, an attempted facsimile of animal nervous systems. In 1943, Warren McCulloch and Walter Pitts made a seminal study of the nervous system[5] with regard to propositional logic, in which a vast network of neuron cells passes signals from one neuron to another. The connecting organ between neurons, the synapse, is like a two-state tap that may be on or off, connected or disconnected, depending upon the truth of a proposition.

Artificial neural networks (ANN), which have grown from McCulloch and Pitts' work [6], [7], [8], are inspired by the nervous system, though some researchers note the glaring differences between the artificial and organic neurons [9]. AI has made much use of functional structures (that is, something which maps inputs to outputs) to represent decision-making. ANNs potentially represent an enormously complex function, mapping inputs (analogous to the stimulation of the brain) to outputs (analogous to the mental and physical reactions of an animal to stimuli) (see Fig. 3). The beauty of the ANN lies in its potential for extreme complexity and nonlinearity and, like its simplest predecessor the linear combination, the ANN can be adapted, based on past input-output pairs, to make better and better decisions over

time. These abilities make the ANN a powerful function approximator with applications in complex knowledge representation and computer learning.



(a) A feedforward (one-way) neural network

(b) A neural network with cyclical links

Figure 3: Artificial neural networks define a learned function that map an input vector to an output vector by means of a layered network of artificial neurons. Each neuron encapsulates a trigger function that works upon a weighted input. If the output of the trigger function passes a certain threshold, the neuron "fires", sending a signal to the next neuron in the chain. Many different trigger functions and many different network topologies exist.

A look at the use of an ANN in guided policy search can be found in Section 2.3.2.

1.2 The training data problem

The AI/ML techniques introduced in this section, supervised learning and reinforcement learning especially, suffer from the same problem. They both require large amounts of training data. In order to represent highly abstract identification tasks in supervised learning, the agent requires enough training examples to adequately describe all the features that might affect decision to offer an output in any one class, or at any point along the continuum.

Consider the previously visited scenario where a classifier wants to tell us whether the photo is taken at night, or during the day. How does one determine this? Is it a matter of the brightness of the photo? This might be a factor, however, a photo taken in the desert on a moonlit night might be almost as bright as photo taken on an overcast day. Could the colours play a role? Certainly, but a hazy early evening photo of a sun-drench landscape might look only as yellow as a night photo of a US American school bus.

Many examples must necessarily be fed to the supervised learning classifier in order for it to learn how to tell apart a night photo and a day photo. Similarly, in reinforcement learning, many trials of the process may be required before the agent adequately understands the cost/reward landscape enough to minimise the return over the whole process.

This thirst for trial data in reinforcement learning, where the environment may only be partially observable and samples unrepresentative, means that reinforcement learning can be complicated and complex. Guided policy search goes a long way to overcome these problems

by breaking down the reinforcement learning problem into a supervised learning component and a mechanism to feed this component easily and swiftly generated training data.

1.3 This thesis

1.3.1 Purpose

The aim of this thesis is to implement guided policy search on the KUKA LWR 4+ robotic arm, which is in use in Aalto University's Intelligent Robotics Group, and to demonstrate its utility in reaching tasks while solving the training data problem.

The main problems tackled during the preparatory work for this thesis are understanding the operation of GPS and its most basic variants, the transfer of the existing abstract controller for the PR2 robot to the newer robot-independent ROS Control software interface and configuration of GPS experiments to work with the dynamics of the KUKA robot.

The performance of GPS trajectory optimisation is evaluated using an experiment investigating the convergence of the learning agent on an optimal policy under a number of experimental conditions of varying difficulties, and an experiment investigating the performance of a single agent in post-learning operation under initial conditions perturbed from the training condition.

The deliverables are a novel controller plugin using the ROS middleware suite to allow easy modification for any similar robot, an agent to interface the GPS backend with the controller and a number of demonstrative example experiments.

1.3.2 Scope

The agent and experiments detailed in this thesis are limited to the use of GPS components implemented by Chelsea Finn in connection to her work with the discoverers of GPS. Within that scope, only the original linear-quadratic regulator using Gaussian mixture models to model transition dynamics[10] is used for trajectory optimisation.

Due to time constraints the neural network component of GPS, which implements policy generalisation, has not been experimentally evaluated.

The novelty in this thesis comes from the implementation of a robot controller for GPS using more up-to-date, flexible and widely used components of the ROS middleware than Finn has used in her work, rather than replacement of the GPS trajectory optimiser, which is a longer-term goal.

1.3.3 Structure of this thesis

Section 2 goes into more detail on the core AI topics introduced in this section, introduces trajectory optimisation, a fourth non-AI component of the standard GPS used in this thesis, and goes on to discuss the motivations and operation of guided policy search.

Section 3 describes the existing components used to implement GPS on the KUKA LWR 4+ and then goes on to detail the development of the new components required.

Section 4 details the operation of the experiments on the new platform and their results, with additional discussion on the limitations of these experiments and future tasks for the author and/or others.

Section 2

Technology review

This section introduces the primary techniques used by guided policy search (GPS) in the context of reinforcement learning (RL), of which GPS in an example. Section 2.1 gives a broad overview of RL, including the common description and classes of RL problem. Section 2.3 then introduces GPS in light of the problems with earlier RL techniques. The novel aspects of GPS are then discussed with reference to their past uses outside GPS. Finally, the specifics of GPS are discussed.

2.1 Reinforcement learning

(Successful) reinforcement learning enables a machine to take actions in an ongoing process in order to achieve a desired goal. It is a form of learning by doing in that the machine agent discovers how successful it is by going through the process, taking note of how well it did and then using this feedback (reward) to improve its own performance next time. The agent's actions themselves affect the course of the process. Therefore, in order to gain a wide view of the potential outcomes, the process must be undertaken again and again until the agent has sufficient information. In some respects, this learning process is similar to how human beings learn to control processes [11].

2.1.1 Environment, agent and Markov decision processes

Reinforcement learning emerged from the field of optimal control, a subfield of control engineering in which process controllers are developed by minimising a cost or maximising a reward function of a system's state, the cost/reward being some abstract measure of success in the control task. These concepts became key to reinforcement learning and resulted in a common notation for definition of an RL problem.

A process for which we are to perform RL then consists of two entities: the environment and the agent. The agent takes actions in its environment and then receives a new environment state and a reward. It is the maximisation of the sum of these rewards (termed the return) that marks a successful learning agent.

A complication in some RL processes is that the state returned to the agent after each action is incomplete; this is analogous to how, if we play the electric guitar loudly in our front rooms, it is not immediately obvious that the neighbour is getting upset and may soon call the police. The state returned to the agent are therefore termed *observations*. RL processes with this complication are called partially observable. Furthermore, since you might receive negative rewards due to the police being called, this example also introduces the concept of the delayed reward. The action that *really* caused the negative reward (playing loud music) may not be the action that *immediately* resulted in the negative reward, such as opening the front door to the police long after you stopped playing loud music. Reinforcement learning methods easily propagate later rewards back through a process, but assigning responsibility for reward to particular actions is a more complicated topic.

We end up, therefore, with the description of the process shown in Fig. 4.

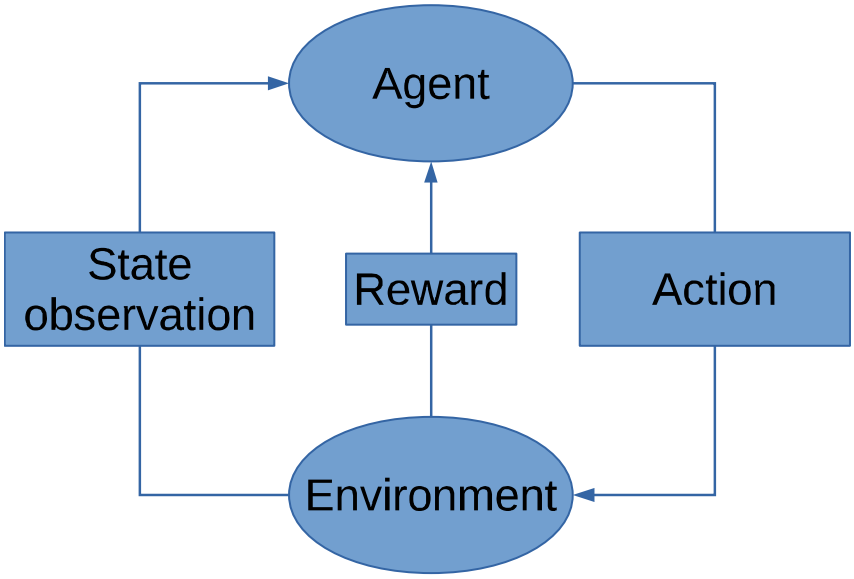


Figure 4: A visual description of a reinforcement learning process, in which an agent acts in an environment and receives rewards and observations afterwards

The breakthrough in optimal control came with Bellman's 1957 book Dynamic Programming [12], which described a recursive method of optimising the reward function by expressing the return value of a system state in terms of the immediate reward by taking a particular action now and the expected return value of the resultant state.

Later in the same year, Bellman formalised a discrete, stochastic variant of dynamic programming [13], which made the technique suitable both for noisy, real-world stochastic processes and computerised solution. The name Bellman gave these processes was the Markovian or Markov decision process (MDP).

Basic Markov decision processes are defined by:

- State space $\mathcal{X}$
- Action space $\mathcal{U}$
- Probability distribution of a particular initial state $p(\mathbf{x})$
- Dynamics - probability of an action $\mathbf{u}$ in state $\mathbf{x}$ resulting in state $\mathbf{x}\prime$, $p(\mathbf{x}\prime|\mathbf{x}, \mathbf{u})$
- Reward function $r(\mathbf{x}, \mathbf{u}, \mathbf{x}\prime)$

At each step t in the MDP, given an action $\mathbf{u}_t \in \mathcal{U}$ from a state $\mathbf{x}_t \in \mathcal{X}$, there is a *probability* that these will result in a:

- new state $\mathbf{x}_{t+1} \sim p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$
- reward $r_{t+1} = r(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})$

What formalises the MDP is that at each time step t, the probability distributions of new rewards and states depend only on the prior state and reward—the history of the process is encoded in the present state. Collectively, these probabilities are called the transition probabilities, or dynamics.

The goal of the learning agent is to develop a deterministic policy $\pi(\mathbf{x}_t)$ or stochastic policy $\pi(\mathbf{u}_t|\mathbf{x}_t)$ for traversing the MDP—a function mapping the current state $\mathbf{x}_t$ to an action $\mathbf{u}_t$—which optimises the expected return $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$, where γ is a return discount rate, i.e. a factor that places importance on the rewards in the next few steps. As the agent moves through the MDP, a trajectory $\tau = (\mathbf{x}_t, \mathbf{u}_t)_{t=0}^{\infty}$ is generated.

It should be noted that not all MDPs are ongoing. Many, including the robot MDPs on which GPS works, have an end—they are episodic. In this case, after the MDP passes the end, there are considered for the purposes of maintaining the same return equation, that there are an infinite number of zero-reward state transitions.

It is now possible to define a function known as the value function under the policy. The value of a state is the expected return starting in that state and following the current policy:

$$\begin{aligned} v_\pi(\mathbf{x}) &= \mathbb{E}_\pi(R_t|\mathbf{x}_t = \mathbf{x}) \\ &= \mathbb{E}_\pi(r_{t+1} + \gamma R_{t+1}|\mathbf{x}_t = \mathbf{x}) \end{aligned}$$

A secondary value function can also be defined, the action-value function $q_\pi(\mathbf{x}_t, \mathbf{u}_t)$. The action value is the expected return starting in state $\mathbf{x}_t$, enacting $\mathbf{u}_t$ and then continuing in the same way as with the state-value function $v_\pi(\mathbf{x}_t)$:

$$q_\pi(\mathbf{x}, \mathbf{u}) = \mathbb{E}_\pi(R|\mathbf{x}_t = \mathbf{x}, \mathbf{u}_t = \mathbf{u})$$

Bellman states that the optimal value functions are those value functions which maximise the expected returns under the policy at hand, i.e. the value functions where the step-in-hand reward is the reward for taking the action leading to the state with the optimal next value

function. The optimal policy π_* is the policy under which the value of each state is maximised for each and every state in the state space.

The value functions have recursive definitions for their optima called the Bellman optimality equations $v_*(\mathbf{x})$ and $q_*(\mathbf{x})$:

$$\begin{aligned} v_*(\mathbf{x}) &= \max_{\mathbf{u}\in\mathcal{U}(\mathbf{x})} q_{\pi_*}(\mathbf{x},\mathbf{u})) \\ &= \max_a \mathbb{E}_{\pi_*}(R_t|\mathbf{x}_t=\mathbf{x},\mathbf{u}_y=\mathbf{u}) \end{aligned}$$

$$q_*(\mathbf{x},\mathbf{u}) = \mathbb{E}(r_{t+1} + \gamma \max_a q_*(\mathbf{x}_{t+1},\mathbf{u}\prime)|\mathbf{x}_t=\mathbf{x},\mathbf{u}_t=\mathbf{u})$$

2.1.2 Tabular derivation of the value functions and policies

For MDPs with small, discrete state and action spaces, it is possible (because values/policies for each state and action can fit in a computer's memory) to derive piecewise optimal value functions and policies by progressively constructing a lookup table for each state or state-action pair. There are a number of these tabular methods, however, there are two (very) basic methods which are illuminating later on in this thesis. They respectively utilise *dynamic programming* (DP), in which a problem simplified by recursively biting off a chunk of the problem and dealing with that before moving on to the rest, and *Monte Carlo methods*, in which functions are constructed numerically from experimental data.

Using DP to tabularly solve MDPs is computationally expensive—prohibitively so in all but quite small MDPs [4]. It is a brute-force approach. Using the above equation for the value function, the value function may be found by the solution of simultaneous equations for each state, however iterative evaluation of a policy is a more practical alternative. In addition to its expense, the DP approach demands complete knowledge of the dynamics of the process.

The tabular value function is then derived by calculating interim values for each state until convergence, using arbitrary initial values.

$$\begin{aligned} v_{k+1}(\mathbf{x}) &= \mathbb{E}_\pi(r_{t+1} + \gamma v_k(\mathbf{x}_{t+1})|\mathbf{x}_t=\mathbf{x}) \\ &= \sum_{\mathbf{u}} \pi(\mathbf{u}|\mathbf{x}) \sum_{\mathbf{x}\prime,r} p(\mathbf{x}\prime,r|\mathbf{x},\mathbf{u})[r+\gamma v_k(\mathbf{x}\prime)] \end{aligned}$$

Note that the update in the DP method of policy evaluation is based on the *expected* next state.

Improvement of the policy relies on a simple operation. For each state, consider each possible action. If a particular action-value is higher than the state-value, then the policy should be changed to reflect that fact. In deterministic tabular policies, this is just a matter of replacing the entry for the current state with the new action. In stochastic policies, it is up to the programmer to determine how much to increase the probability of the new preferred action (see Section 2.1.3 below.

The repeated application of policy evaluation and policy improvement is called *policy iteration* and can be shown to converge on the optimal policy, however, it is practical to cut off the iteration once the changes to the value function grow smaller than a certain threshold.

A slightly different technique is *value iteration*, in which only one step of policy evaluation is conducted before a new policy is found. Although this may take longer to converge

than purely the policy improvement stage of policy iteration, it avoids the multiple value-function updates in the policy evaluation stage. Overall, therefore, value iteration tends to be faster and still converges on the optimal policy.

$$v_{k+1}(\mathbf{x}) = \max_a \mathbb{E}[r_{t+1} + \gamma v_k(\mathbf{x}_{t+1}) | \mathbf{x}_t = \mathbf{x}, \mathbf{u}_t = \mathbf{u}]$$

Monte Carlo methods (MC) are an alternative to the iterative DP approach. MC uses a set of experimental data to derive a value function and perform the policy update.

The expected value of a random experimental variable is the mean of the values it takes in an infinite number of trials. In simple MDPs, with a low number of states and a low number of actions possible in each state, finding the value function is a matter of running the agent through MDP a sufficiently large number of times that the mean return from each state in the sampled trajectories has tended towards the expected value enough that deviations from the true value function are so small that they make little or no practical difference.

First, run the MDP using a given policy and collect trajectories τ_i where $\tau_i = \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}, r_{t+1}$.

For each trajectory, calculate the empirical return $R_{t,\tau}$ in each state $\mathbf{x}_{t,\tau}$ for each time step t_τ. The MC value function is:

$$v_{MC}(\mathbf{x}) = \bar{R}_\mathbf{x} \; \forall \; \tau$$

A popular and graphic simple MDP is the gridworld game (Fig. 5), in which the agent starts in a particular box in a grid and learns to move towards the green goal states by maximising the return. If the agent reaches the target states, it receives either a +1 or +10 reward and the process ends. If it reaches a red state, it receives a -10 reward and the process ends.

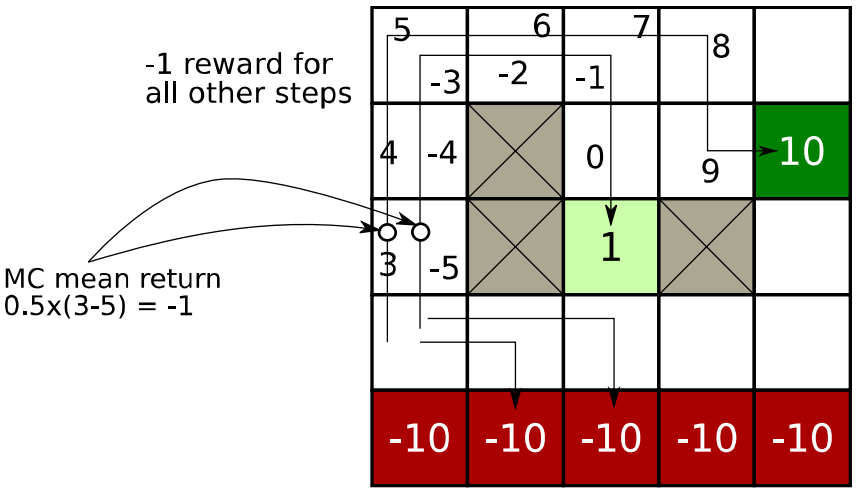


Figure 5: The gridworld illustrates how value functions for simple MDPs can be found by trial. The empirical returns for the highlighted state are 3 and 5, making the average return (value) -1.

At the end of each trial, a return can be found for each state in the agent's trajectory simply by counting back and dividing out the discount rate. At the end of the series of trials, the recorded return values for each state can be averaged.

Two variants exist of this simple Monte Carlo policy evaluation: first-visit MC and every-visit MC. In first-visit, only the return at the first visit to any state is included in the

calculation of the mean return, while in every-visit, all empirical returns are used. Both forms converge on the true value function, but every-visit MC is more suitable in some advanced RL methods [4].

The action-value function can be calculated using MC methods from the same data as the value function, however, it should be noted that for deterministic policies, only the greedy action (the action with the highest return) will be followed for each state. This may also happen for stochastic policies where certain actions almost always result in a certain successor state. The ignored action values will never, or rarely in the latter case, be improved from their initial value.

For more complicated MDPs, where, for example, states and actions are drawn from continuums, or where the sets of possible states and/or actions are otherwise so massive that an exhaustive evaluation of the value functions is impractical, approximations of the value functions must be made which are nonetheless *good enough*.

2.1.3 Policy improvement: Exploitation and exploration

In the gridworld example, an initial policy π in $\mathbf{u}_t = \pi(\mathbf{x}_t)$ might merely be uniformly random, i.e. the action is drawn randomly from a flat distribution across all possible actions. This is not an optimal policy, because it lends no importance to actions that will produce higher returns.

Having conducted the trials 10,000 times, or 100,000 times, we have now derived a value function which shows us which states, or which state-action pairs can be expected to lead to higher returns. This information can be used to improve on the existing policy. For each action-value in a particular state, we can test to see if it is higher than the state-value function for the current state. We can then increase the probability of actions with higher action-values and reduce the probability of actions with lower action-values.

Iterating this process of value function evaluation and policy improvement leads ultimately to the convergence of the policy on the optimal policy. As with the value function evaluation, this is a feasible process for simple MDPs, but for higher-complexity state-spaces, the exhaustive approach is infeasible.

Furthermore, how much should the probability of high-return actions be increased? The limiting increase, of course, is complete determinism: the policy always returns the action with the highest action-value. But this may lead to suboptimal, even bad policies. The more deterministic the policy becomes during the policy update, the less likely the agent is to explore *apparently* suboptimal states in the next round of value-function evaluation. Since the value functions change with each policy update, this "greedy" behaviour can lead to situations where high-return states are never explored by the agent because they were not high-return states in previous iterations.

A real-life analogy is the rise in business of short-termism [14], whereby company directors are encouraged by shareholders' greed for dividends to maximise short-term profit by slashing investment in research and development, infrastructure and jobs. The best than can be said is that it *might* work, but it is likely that near-term reward, rather than long-term return, is being maximised.

It is, therefore, important that the policy remain stochastic—that, even in the face of mountain evidence of a certain action-value's efficacy, lower-value actions still have a chance of being taken.

2.1.4 Approximation and the value function

While faster algorithms exist for the kind of table-lookup reinforcement learning covered above, the dimensionality of real-world MDPs can easily become so insurmountable that much faster methods of learning are required. These much faster methods involve the approximation either of the value function or the policy, using known interactions of the learning agent with certain environmental states to extrapolate to unknown interactions with states with similarities to the examples. This process is very similar to the kind of regression performed in supervised learning and, in fact, supervised learning and this form of reinforcement learning share many forms of function approximation.

In function approximation, we parameterise the function of interest with a limited set of features ϕ of the state. For each feature, we assign a weight w. The approximate value function $\hat{v}(\mathbf{x}, \mathbf{w}) \approx v(\mathbf{x})$ is some combination of the weights and the state features, for example, the linear combination:

$$\hat{v}(\mathbf{x}, \mathbf{w}) = \mathbf{w} \cdot \phi(\mathbf{x})$$

This form entered machine learning in 1958 with Rosenblatt's paper on the perceptron[6], an elementary neural network which used a biased linear combination to activate an artificial neuron and perform binary classification. Although Rosenblatt's ideas were debunked and pushed to one side 11 years later [15], the perceptron eventually became the progenitor of more flexible multilayer perceptrons, which can utilise various linear and nonlinear activation functions and which underlie today's deep and wide neural networks used in GPS and across machine learning.

The power of approximation for reinforcement learning lies in the reduction of the vastness of the state space to the minimalistic feature and weight vectors. This reduction in fidelity works, despite the update of the weight vector affecting the value of multiple states, because each state might share multiple features with many other states.

In value-function approximation, one attempts to approximate the true value function by minimising the difference between approximate values and their true counterparts. Typically, the measure to minimise will be the mean square value error.

Since the derivative of this error $e(\mathbf{w}) = \mathbb{E}\left[(v_\pi(\mathbf{x}) - \hat{v}(\mathbf{x}, \mathbf{w})^2\right]$ will reach zero at a minimum, a minimum may be found by applying gradient descent, i.e. repeatedly finding the gradient of the error and changing the weight vector in small steps in the direction that will reduce the error:

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_w e(\mathbf{w}) \\ &= \alpha\left(v_\pi(x) - \hat{v}(x, \mathbf{w})\right)\nabla_w \hat{v}(x, \mathbf{w})\end{aligned}$$

where α is the descent step size.

Since this method aims specifically to tackle large state-spaces, intermediate approximations of the true state-values (such as calculated in the tabular methods of Section 2.1.2) are unavailable and another target value must be used in its place. According to the Monte Carlo method, we can approximate the true value using the sampled return R_t, so that the weight update becomes:

$$\Delta\mathbf{w} = \alpha\left(R_t - \hat{v}(x, \mathbf{w})\right)\nabla_w \hat{v}(x, \mathbf{w})$$

However, other *update targets* are available. In an alternative to Monte Carlo called temporal difference learning [16], the sum of the next sampled reward r_t and the discounted

temporary value of the next sampled state $\gamma\hat{v}(x_{t+1}, \mathbf{w})$ can be used. This can further be extended to include the more than one step in a sampled trajectory.

2.2 Direct search for approximate policies

There exist two connected (big) problems with approximate value function-based methods of RL. While stochastic policies *can* be derived from the value function, the method of doing so is very much in the hands of the implementer. Really, building a policy from a value function lends itself most obviously to ultimately greedy, that is deterministic, policies. Sutton et al [17] decried value function approximation for its erratic policy updates and lack of convergence on optimal policies. They précised an emerging class of RL methods in which, instead of deriving a stochastic policy from an approximate value function, the policy itself becomes the parameterised function.

In their monograph on the topic (written, alas, before the announcement of guided policy search), Deisenroth, Neumann and Peters [18] champion policy search methods for their compact parametric representation in large state spaces, explorative properties and smooth trajectory generation, but note that they tend to result in only locally optimal policies, i.e. policies that work well around certain trajectories through state space.

They also praise policy approximation over value approximation for "safe" parameter updates, whereby only small policy updates are permitted. In value-function approximation, small changes to the value function may result in large changes to the policy, making the policy dangerous to use on real robots.

For a developing policy $\pi(\mathbf{x}, \mathbf{w})$, the essential form of policy update is similar to value-function update:

$$\pi_{k+1}(\mathbf{x}, \mathbf{w}) = \pi_k(\mathbf{x}, \mathbf{w}) + (\Delta\pi)_{k+1}(\mathbf{x}, \mathbf{w})$$

As opposed to the tabular policies in Section 2.1, which use a piecewise probability mass function, approximative policies in policy search may use any sort of probability density function, since it the *parameters* of the function that are being updated. Policy representations may be picked depending upon the type of MDP being controlled.

2.2.1 Model-based and model-free policy search

An important classification in policy-based RL methods is between model-based learning and model-free learning. Both classes use experimental data, but at different stages of the policy development and in often radically different quantities.

Model-based methods learn a dynamic model of the MDP $p(x\prime|x, u)$ and then develop policy updates based on these dynamics. The experimental data requirement comes during the learning of the process dynamics.

Model-free methods are usually used when it is difficult or impossible to acquire the process dynamics. For example, in scenarios where a robot is called upon to interact with unknown, changing external objects, the process dynamics are not simply the dynamics of moving the robot around. The robot dynamics are quite stable. However, introducing an obstacle to prevent a robotic arm's movement, or something that it must grip and move around, means a complete change in the process dynamics in the region of the object. Hence, the empirical data sample requirement for model-free methods comes at every iteration of policy update.

The difference in data sample usage makes model-based methods more sample efficient than model-free methods (on the whole). Nonetheless, model-free methods are popular in real-world applications[4] due to the proliferation of MDPs with unexpected and/or complex dynamics. Guided policy search, which in its original form is a hybrid of the model-free and model-based approaches, specifically reduces the complexity of its dynamic models using a model-based component, before feeding its results to a model-free general policy.

Whether one opts for a model-based or model-free RL method, there are a growing number of choices of algorithm.

2.2.2 Actor-critic reinforcement learning

Policy search methods have a notable problem—that the variance of the policy can be very high [4]. In real-world systems, this can result in frequent undesirable outcomes and unsafe stresses on physical components. Actor-critic architectures have been designed to overcome this problem.

The idea is that, similar to the idea of supervision in supervised learning, a process controller—an actor—has a check on its behaviour—the critic. This idea goes back to a 1983 paper by Barto, Sutton and Anderson [19] which detailed a neural control system called the associative search element (ASE), similar to the parameterised RL policy. A second neural device termed the adaptive critic element (ACE) further focused the control policies of the ASE by evaluating the worth of its current configuration.

In general policy-based RL, the actor is the parameterised policy system, such as the Monte Carlo policy gradient learner, while the critic is a parameterised approximate value function like the value function in Section 2.1.4. At each policy improvement step, the current policy is evaluated—run through—and state-action-reward samples are used to fit a value model of the policy. Once this model is sufficiently developed, instead of the policy updating according to its own gradient, the policy updates according to expected return given by the derived value function (see Fig. 6).

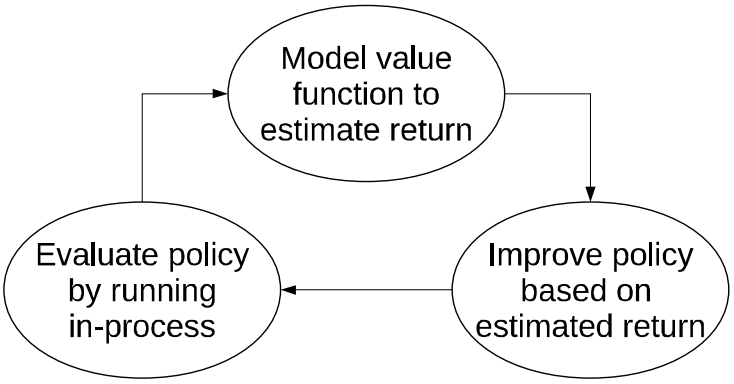


Figure 6: Policy update cycle of actor-critic RL

Although GPS is not an actor-critic RL method, it does benefit from the important features of this architecture. GPS specifically produces low-variance local policy examples and conducts supervised learning using these examples as supervisor.

Policy-search RL	complex dynamics	complex policy	HARD
Supervised learning	no dynamics	complex policy	EASY
Trajectory optimisation	complex dynamics	simple policy	EASY

Table 1: Levine's simple précis of GPS reasoning

2.3 Guided policy search

Guided policy search [10] is an approxmative policy-search RL method developed by Levine and Koltun to avoid a single problem common to many previous policy-search methods: that they easily converge prematurely into only locally optimal policies. While policy-based approximative RL methods are better than value-function methods at converging on locally optimal policies, the caveat that these optimal policies may only be local is a feature that can only be utilised in certain circumstances [20] where a more general policy is not required.

The algorithm can be considered (in the broad sense) as a two-part process (see Fig. 7a). The first part deals with generating a collection of local control policies; the second deals with training a general policy that interpolates between the local policies collectively to form a general policy. This general policy can then be used to improve development of local policies.

This structure allows reinforcement learning to be simplified, using only regressive supervised learning to train a general policy and as simple a reliable method as one wishes for the generation of local policies. Although it was not initially formulated by Levine and Koltun in such terms, looking at GPS in this way right from the start assists in understanding later variants of the algorithm.

Discussing vanilla GPS, Levine describes the simplification of RL with GPS, which is reproduced in Table 1. It shows how the two components of a model-based RL algorithm, the dynamics and the policy, can each be built up piecemeal from specific local parts. This turns the weakness of model-based methods—the need to build complex dynamic models—into a strength.

Levine and Koltun's original formulation of GPS is a model-based RL method, however, the dynamic model is used only in the first half of the algorithm to generate the guiding local policies which are then used to train the general policy. The general policy may be improved independently of the dynamic model, allowing the general policy to be built up from a number of dynamic models specific to subsets of the process statespace.

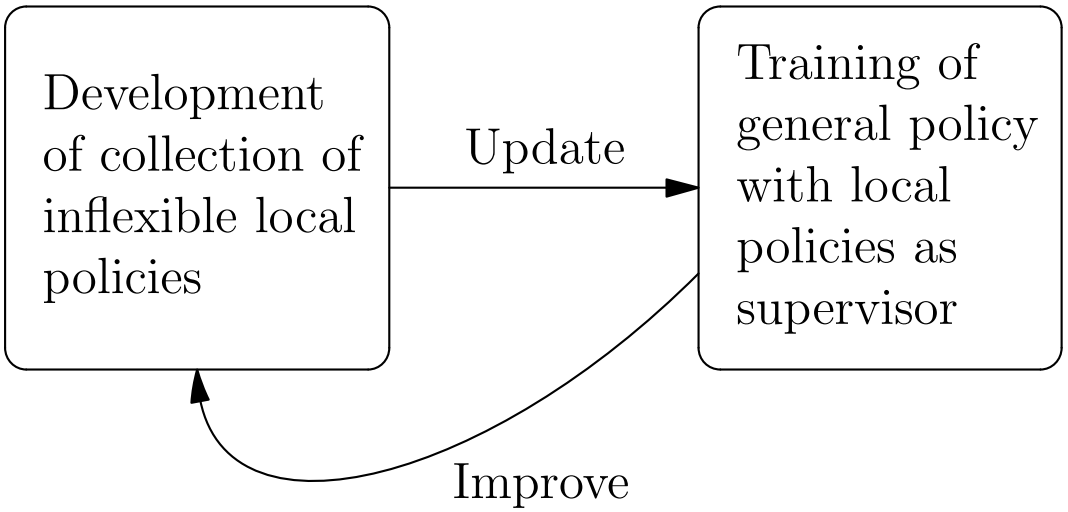
Figure 7b shows the full original GPS algorithm. The whole diagram belongs in the first (left-hand) part of the abstract diagram in Fig. 7a except that component labeled "train $\pi_\theta(\mathbf{u}|\mathbf{x})$", which is the general policy training component.

The algorithm begins with a default policy, which is run from a starting state to generate a set of trajectories of known length. The state transitions are then used to formulate the state-transition probability function $p(\mathbf{x}\prime|\mathbf{x}, \mathbf{u})$.

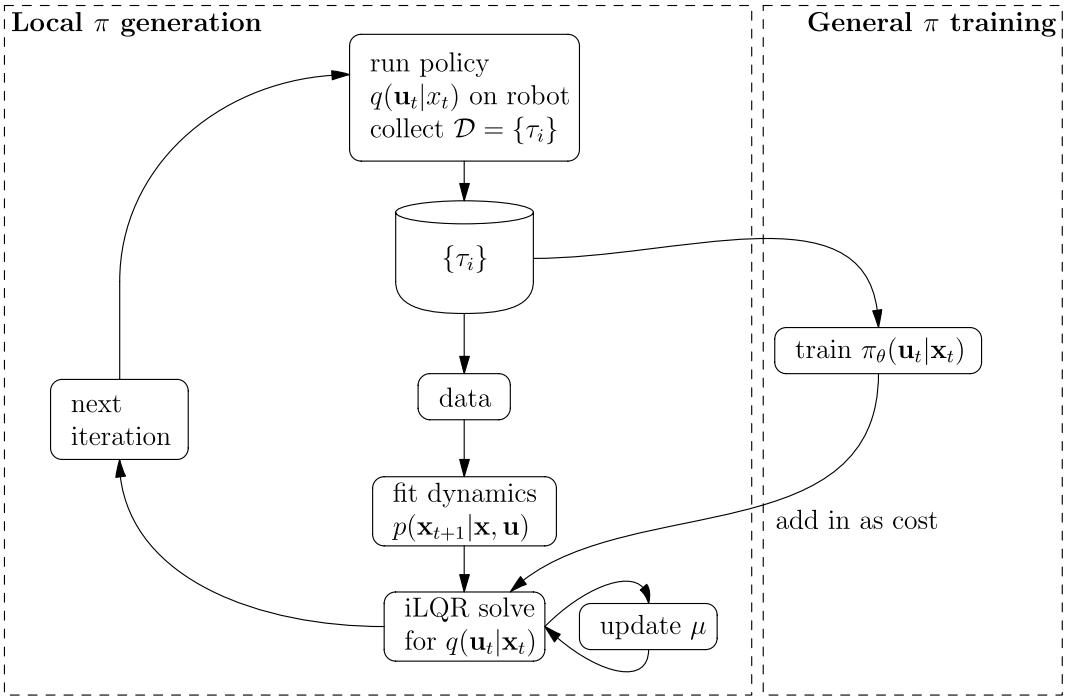
The new local policy is then run to generate a new set of empirical trajectory samples.

2.3.1 Trajectory optimisation

Trajectory optimisation is used to create a controller that will attain the optimal path through state space from a given initial state to a given target state. It is used by GPS to improve local policies. In the first instance, local policies are run to generate a set of trajectories. But



(a) Guided policy search at its most general is a two-part iterative process of local-policy generation and general-policy training



(b) The basic formulation of the GPS algorithm in which local policies are built up around guiding samples using trajectory optimisation, before the general policy is trained using supervised learning

Figure 7: Guided policy search

these trajectories are unlikely to be optimal for any suboptimal controller (local policy) and in fact may be poor.

This is a common problem in control engineering for which the entire field of optimal control was developed. How does one develop a controller that will attain that optimal trajectory?

First, we need to understand how bad it is when the controller fails to achieve the optimal path through statespace, how bad it is when the controller fails to attain the desired final state in terms of the distance between the desired and actual final states. Optimal controllers solve the optimisation problem by minimising a measure of this badness called the cost, just like the maximisation of the reward in the usual RL formulation.

The linear-quadratic regulator (LQR) is a form of controller that minimises a quadratic cost function for a linear system and produces a gain matrix derived from the cost of past state transitions, i.e. actions. The cost function is quadratic because this reduces the effects of small noises in the state signal.

Given a trajectory through statespace, the cost function that is minimised in GPS is a function of the actions and the state. These two terms correspond to the swiftness/smoothness of the trajectory and the fidelity—the error—of the actual final state.

The major problem with traditional LQRs is that they assume constant linear dynamics that may not hold true once the trajectory begins to be optimised—certainly the case in GPS. Tassa et al[21] developed a variant of the LQR called iterative LQR (iLQR), which applies LQR over and again, updating linearised dynamics that it works with on the basis of the new trajectory it generates and then reoptimising the trajectory using the new dynamics. If the difference between the return on the new trajectory and the return on the original trajectory is lower than a set threshold, then the trajectory has been fully optimised as is returned as the new local GPS policy. Otherwise, if the cost of the new trajectory is lower than the cost of the old trajectory, the new trajectory becomes the old trajectory and the process reiterates until convergence. If the cost of the new trajectory is higher than that of the old, then the trajectory has been changed too much and a more conservative attempt at optimisation is made.

The original GPS paper formulates iLQR thusly, the state and action subscripts meaning partial differentiation with respect to those functions:

Given a trajectory $(\bar{\mathbf{x}}_1, \bar{\mathbf{u}}_1), ..., (\bar{\mathbf{x}}_T, \bar{\mathbf{u}}_T)$, define:

$$\hat{\mathbf{x}}_t = \bar{\mathbf{x}}_t - \mathbf{x}_t$$
$$\hat{\mathbf{u}}_t = \bar{\mathbf{u}}_t - \mathbf{u}_t$$

The dynamics and reward are:

$$\hat{\mathbf{x}}_{t+1} \approx f_{xt}\hat{\mathbf{x}}_t + f_{ut}\hat{\mathbf{u}}_t$$
$$r(\mathbf{x}_t, \mathbf{u}_t) \approx \hat{\mathbf{x}}_t^T \mathbf{r}_{xt} + \hat{\mathbf{u}}_t^T \mathbf{r}_{ut} + \frac{1}{2}\hat{\mathbf{x}}_t^T \mathbf{R}_{xxt}\hat{\mathbf{x}}_t + \frac{1}{2}\hat{\mathbf{u}}_t^T \mathbf{R}_{uut}\hat{\mathbf{u}}_t + \hat{\mathbf{u}}_t^T \mathbf{R}_{uxt}\hat{\mathbf{x}}_t + r(\bar{\mathbf{x}}_t, \bar{\mathbf{u}}_t)$$

The derivatives of the Q-function and value function and linear policy terms are estimated:

$$
\begin{aligned}
Q_{xxt} &= r_{xxt} + f_{xt}^T V_{xxt+1} f_{xt} & \qquad Q_{xt} &= r_{xt} + f_{xt}^T V_{xt+1} \\
Q_{uut} &= r_{uut} + f_{ut}^T V_{xxt+1} f_{ut} & \qquad Q_{ut} &= r_{ut} + f_{ut}^T V_{xt+1} \\
Q_{uxt} &= r_{uxt} + f_{ut}^T V_{xxt+1} f_{xt} & & \\
V_{xt} &= Q_{xt} - Q_{uxt}^T Q_{uut}^{-1} Q_u t & \qquad k_t &= -Q_{uut}^{-1} Q_{ut} \\
V_{xxt} &= Q_{xxt} - Q_{uxt}^T Q_{uut}^{-1} Q_u xt & \qquad K_t &= -Q_{uut}^{-1} Q_{uxt}
\end{aligned}
$$

The iLQR then gives a deterministic optimal policy for the optimised trajectory:

$$g(\mathbf{x}_t) = \bar{\mathbf{u}}_t + \mathbf{k}_t + \mathbf{K}_t(\mathbf{x}_t - \bar{\mathbf{x}}_t)$$

However, for the purposes of GPS, a stochastic policy, a trajectory *distribution*, around the optimised trajectory is required. It is from these distributions that the general policy is built.

$$\pi_{\mathcal{G}} = \mathcal{G}(\mathbf{u}; g(\mathbf{x}_t), -Q_{uut}^{-1})$$

2.3.2 Local policy acquisition, training of general policies and neural networks

The local policy improvement portion of GPS, which constitutes the bulk of the algorithmic complexity, is only preliminary to the main goal: deriving a general policy. GPS trades for a reduction in the computational difficulty of the full algorithm by building up a general policy from a number of specific local policies. In more elementary RL methods based on the approximation of the policy function, notably the model-free Monte Carlo-based REINFORCE algorithms, the general policy approximation must be explicitly trained across the entire state and transition spaces. GPS avoids this by training its global policy approximation to mimic highly region-specific local policies that would be utterly insufficient to describe the entire state and transition spaces by themselves, and hoping that there are enough local policies that the global policy can effectively interpolate between them.

For each set of local policy improvements between a set pair of initial and final states, there is a long train of empirical trajectories emerging as each successive policy is run in the first component of Fig. 7b. These trajectories are used to calculate new transition dynamics, but they are also used to train the general policy approximator.

For each trajectory τ_i, there is a sequence of overlapping state-action pairs $(\mathbf{x}_t, \mathbf{u}_t)$. These are used as supervisor input-output pairs to the general policy approximator in a process that is, quite simply, supervised learning.

In vanilla GPS, the general policy approximator is a deep feedforward neural network. These networks are a much larger, nonlinear extension of the linear combination, in which layers of singly parameterised artificial neurons are used to represent the policy function. Each neuron is comprised by a activation function/weight pair, with the input being weighted. These weights are the policy parameters. With an activating input, the neuron "fires" and sends a signal to the next neuron. In the experiments conducted for the original GPS paper, the activation function used was a soft rectifying nonlinearity, or softplus function $a = \log(1 + e^x)$, which is used as an approximation of linear rectifier (see Fig. 8).

The neural network learns by comparing the global network output to the actions given in the supervisory samples—the actions generated in the running of the local policies—and then modifying the input weights of each neuron in the network in accordance with whether

the global output state matched the supervisory output. Since Levine and Koltun required a stochastic policy, but the plain neural network learns to produce a deterministic policy (i.e. to always match the supervisor outputs), Gaussian noise was added to the output (see Fig 9).

This process is similar to the policy gradient or value-function gradient update in simpler approximative RL methods.

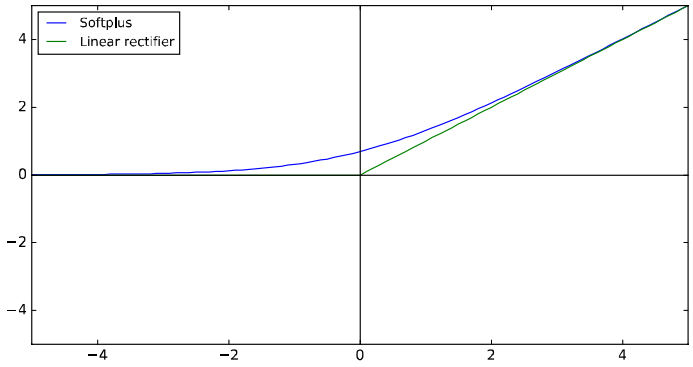


Figure 8: Comparison between the softplus function and a linear rectifier

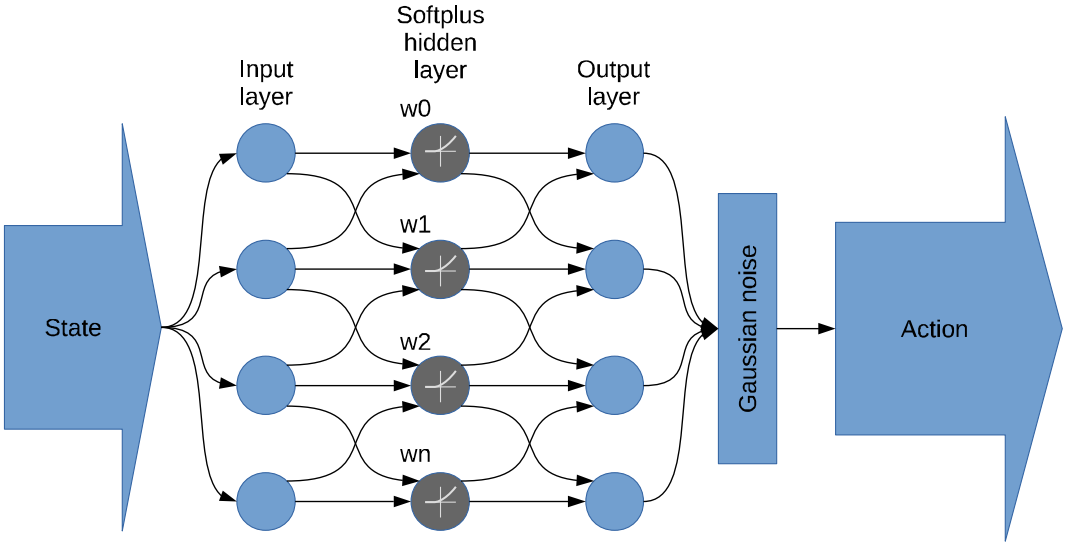


Figure 9: This simplified diagram of the neural network used to train the GPS general policy, with softplus activation functions in the single hidden layer and Gaussian noise added at the output to give a stochastic policy

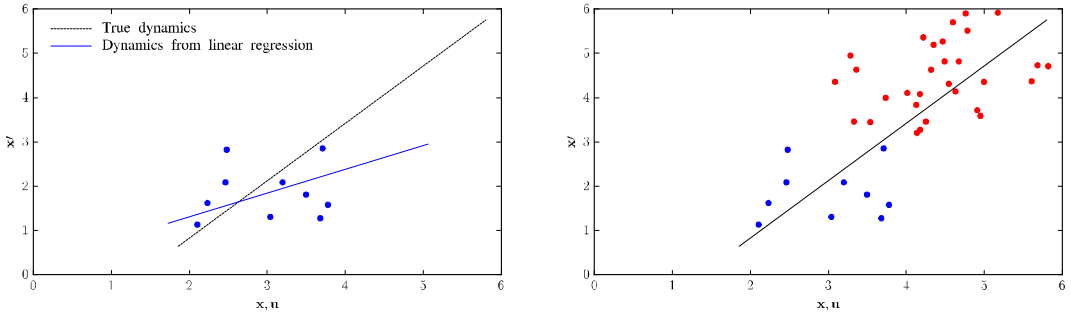
2.3.3 Criticisms and developments of GPS

GMMs for better dynamics: Levine and Koltun have continued to contribute to the development of GPS. A particularly helpful insight of Levine and Abbeel's [22] was that the derivation of the dynamics from newly sampled trajectories in the local policy generator relied on a small number of samples and that it could be too easy for a linear regression to these points to poorly represent the real dynamics (see Fig. 10a).

In an extension of GPS, the new trajectories are also used to train a Gaussian mixture model (GMM). New trajectories are *added* to the GMM in addition to being used to calculate the dynamics. After the first iterations of local policy development, the GMM becomes a prior in the derivation of the new dynamic model. This leads to the fit of the dynamics to the new samples being better (see Fig. 10b).

Adversaries for robustness: Although the GMM addition to GPS results in more robust linear fits for the dynamics, GPS is not designed with disturbance of the environment in mind and does not include a noise term in the model of the state update in the LQR. Ogunmolu [23] adopts a trajectory optimisation process similar to the DDP process suggested by Abbeel. GPS is augmented with an adversarial term in the optimisation cost function, and instead of linearising the dynamics, a 2nd-order Taylor expansion of the Bellman update is used. This was found to result in general GPS policies that were robust in the face of disturbance of the environment.

GPS under unknown dynamics: Following the addition of the GMMs and work on learning GPS policies without unknown dynamics [24], Levine, Wagener and Abbeel developed a GPS system for use on robots in contact-rich environments [25] such as assembling toys and screwing on bottle caps. Using sampled trajectories, like normal GPS, the new algorithm constrains the spatial divergence between the current trajectory and a new one and then optimises the current trajectory by minimising the Kullback-Leibler divergence by dual gradient descent. A dynamic model is then calculated. This process iterates a set number of times.



(a) In the original GPS, the small number of samples may lead to erroneous regressions

(b) Using a Gaussian mixture to keep past dynamic models as a prior to the dynamic fit results in better dynamic models in later local policy updates

Figure 10

Since standard GPS is unable to deal with the unknown dynamics inherent in contact of the robot with other objects, these extensions could be key to much more complicated contact-rich environments in future, such as assembly tasks on Earth and in space.

Model predictive control: Zhang et al. [26] exchange the LQR-based on-policy local policy generation of standard GPS for local policies generated by off-policy model predictive control. The motivation is that in training on MDPs such as their aerial drone tasks, poor initial dynamic models and policies and lack of access to the full environmental state can result in catastrophic failures of the hardware—the drone crashes. Model predictive control uses carefully controlled environments and extensive computation across the full state to generate policies. Once local policies have been generated, they can be used to train a general GPS policy that accepts as input, during operation, partially observable states, such as only the outputs on onboard sensors.

Guided policy search has received considerable attention since its publication in 2013 and the experiments on it and extensions of it are too numerable to be fully explored in this thesis. Other contributions include the use of generative motor reflexes to improve the performance of the general policy in areas of the state space that lie far from the local policies [27], the use of mirror-gradient GPS in the control of the highly non-linear tensegrity robots which have limited sensory input [28], the excision of initial-position resets during the local-policy generation in conditions of stochastic initial positions [29], further research into contact-rich, highly non-linear environments where the LQR trajectory optimiser is replaced with a model-free path integral stochastic optimiser [30] and collective, distributed GPS using multiple robots to reduce learning time [31].

2.4 Summary

Guided policy search has, in five years, grown into a diverse research topic of its own. While being far from the only algorithm of research in contemporary RL, GPS has overcome problems inherent in other RL methods, even those of considerable complexity, such as sample inefficiency and the local optimal policy trap. It is a robust and fast RL technique suited to a range of tasks in software and on robotic hardware. However, particularly in the areas of contact-rich environments and partially observable states, there is still work to be done.

Section 3

Implementation

This section introduces the existing technologies used in the implementation of guided policy search on the KUKA Lightweight Robot 4+ (LWR), before going on to present the new work done to update existing software to work not just on the LWR, but on similar hardware too. Section 3.1 very briefly presents the LWR. Section 3.2 introduces the Robot Operating System (ROS) middleware, used to control robots at an abstract level and specifically the ros_control component. Section 3.3 gives an overview of a ROS hardware interface for KUKA LWR robots. Section 3.4 describes a complex, but flexible implementation of the GPS core algorithm [32]. Section 3.5 discusses the development of the ROS controller, which interfaces GPS and ROS and which constituted the bulk of the work on this thesis. Section 3.6 briefly describes the learning agent, which brings together the components of the GPS backend and runs the experiments detailed in Section 4. Section 3.7 briefly describes a utility written to send commands to the GPS position controller during the setup of the experiments.

3.1 The KUKA LWR4+ robot

The KUKA Lightweight Robot 4+, known in German-speaking countries as the Leichtbauroboter or LBR 4+, is a low-power, light industrial robot (see Fig. 11). It weighs about 16kg and is therefore quite portable and suitable for lab research [33].

It has seven joints (degrees of freedom); six are required are to attain any end-effector position and orientation within reach. The seventh joint is therefore redundant and allows the robot to place its end-effector at any single point while changing its configuration.

The LWR may directly operated by human via a portable control panel called a pendant. In the context of extensions of this thesis, this allows human-directed trajectories to be recorded and used in third-party software applications (see Section 5.2.2).

Third-party systems may directly interface with the LWR via the Fast Research Interface [34] over Ethernet, which allows a programmer to control (using a C++ interface library) the robot by direct joint-space position or torque command, Cartesian position command or joint-space impedance command. The topology of command of the LWR is seen in Fig.12.

The FRI is used by the ROS hardware interface detailed in Section 3.3. In the work of this thesis, a simulation of the LWR was used, however, the hardware interface is intended to allow seamless switching between the robot and the simulation.

3.2 Robot Operating System

Robot Operating System, commonly known as ROS, is a middleware designed to offer abstract control of robots of all varieties, from arms like the LWR to mobile robots and drones.

The primary responsibilities of ROS are package management, process management (including launch), inter-process communication, logging and monitoring. Hundreds of ROS *packages* exist, containing things ranging from support files for specific robots to highly abstract trajectory generation and sensor visualisation tools. A package may contain any number of nodes—executable files, each intended to perform one specific purpose.

The `ros_core` package contains three key *nodes*: `master`, `parameter_server` and `ros_out`. Nodes are executables, usually performing very specific purposes. The key to getting things done in ROS is to connect nodes and configurations from a number of different packages, in order to maintain maximum portability across applications and robots.

The `master` node registers and controls the execution of all the other nodes that a user might load.

ROS *topics* and *services* are used for interprocess communication. Topics are simple one-way communication channels. They are opened by a node that is a *publisher*, which outputs data. *Subscribers* are used by other nodes to receive the information as it comes in. Service nodes are an encapsulation of a publisher and a subscriber. Nodes may query the service by sending data on a certain topic. The service will immediately pick up the data, process it and send a response back on the output topic.

The parameter server is a special part of the `master` node. It is an XML-based dictionary to which any node may upload data it shares with other nodes. All parametric data is visible to all nodes, but is separated into custom namespaces to avoid confusion and duplicate handles.

When ROS was first written, the authors built a robot named the PR2 to demonstrate ROS's capabilities. The implementation of GPS used in this thesis comes with a ROS demonstration using the original PR2 packages, however, this code is unusable for any other robot.

In more recent versions of ROS, an abstractive package called `ros_control` has been included, which aids in writing controllers. Programmers write a ROS controller, which loads a robot-specific *hardware interface*, which in turn exposes one or more of a number of standard or bespoke joint control interfaces. For example, the controller written for this thesis uses the LWR hardware interface and the effort joint interface—for direct torque control. The abstract joint control interface sits between the controller and hardware interface.

The experiments detailed in Section 4.3 utilise a simulated version of the KUKA robot. The simulator, Gazebo, was written for users of ROS and its development is tightly coupled with that of ROS. The Gazebo server node may be communicated with via ROS topics and models of robots (in URDF format) may be uploaded to the ROS parameter server for use by ROS.

Figure 11: A promotional image of the KUKA LWR 4+ [33]

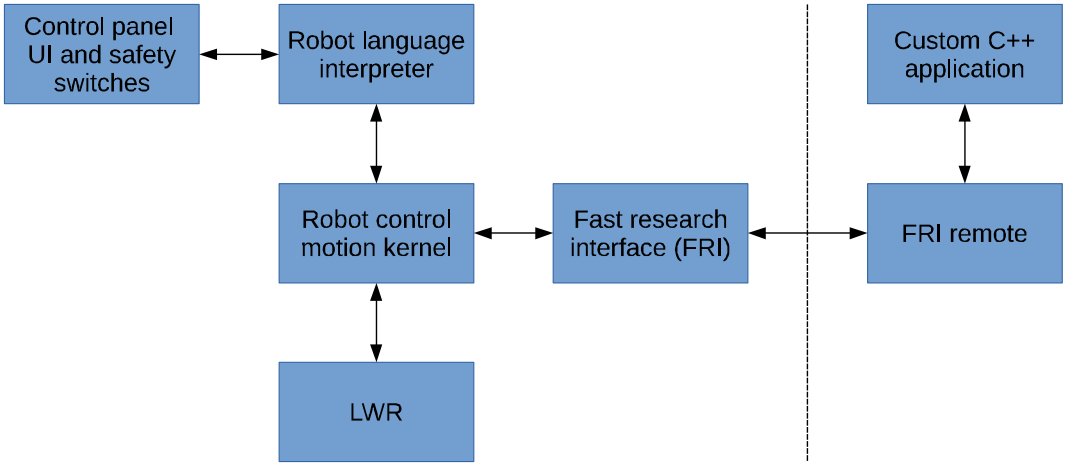


Figure 12: The Fast Research Interface means that the LWR 4+ is not limited to control by KUKA's
bespoke hardware and software

3.3 KUKA LWR hardware interface

An existing LWR hardware interface for ROS [1] was used in this thesis. It was written by staff at the Centro di Enrico Piaggio (Centro E. Piaggio), a bioengineering and robotics research centre at the University of Pisa, Italy. The hardware interface comes as part of a metapackage containing the hardware interface, a set of example controllers for `ros_control`, a URDF description of the robot and an example control scenario.

The hardware interface is built to control both the real LWR via the FRI and the Gazebo simulation. Which interface is loaded should be specified in the controller launcher.

During the development of the GPS controller it was important, during the extensive debugging phase, to be able to follow the torques being sent by the GPS backend all the way to the robot. It was found that there were differences in the hardware interface code between apparently intended behaviours and real behaviours. In particular, the Gazebo-specific hardware interface confused joint effort control, when pre-calculated torques are sent to the robot, with joint impedance control, when a desired torque is requested of the hardware interface and an internal spring-mass control system is used to gradually bring the torque to the desired level.

3.4 Guided policy search suite

The enormous (around 20,000 lines of C++ and Python), but flexible GPS implementation that was central to this thesis work was primarily written by Chelsea Finn, a student under GPS coinventor Sergey Levine, and is hereafter referred to as Finn's work. The component diagram of Finn's GPS can be seen in Fig. 13. To see the experimental configuration, refer to Appendix B.

The agent is the centrepiece of Finn's GPS. After loading up the experimental configuration, it manages the running of the local policy generator/optimiser and the general policy training. It also communicates with the controller. The choice of controller (and robot or

[1] `https://github.com/CentroEPiaggio/kuka-lwr`

other process) is entirely up to the user, but an agent class derived from Finn's base class must be written to communicate with the controller—to accept the transmission of state from the controller and to transmit actions to the controller.

In the case of the KUKA, these quantities are transmitted and received via ROS topics. The agent therefore needs to register as a ROS node and set up publishers for the GPS commands.

The commands that GPS sends to the robot are:

- Get data: sends a request to the controller for the latest state and expects a response
- Relax arm: tells the controller to stop sending torques to the robot
- Reset arm: tells the controller to return the robot to the initial position specified for this round of trajectory optimisations—does not expect a response
- Trial command: sends the controller a policy and expects the return of a trajectory

The controller is the next item of interest. Although it is called a controller, really, for the purposes of GPS, it simply acts as an interface between the agent and whatever robot or abstract control layer one is using. Finn's controller is a complex base class written in C++, named RobotPlugin. Since this base class has no knowledge of the robot, or the hardware abstraction, that will be used, separate classes which are instantiated in RobotPlugin, abstract components in the following ways:

- Sensors:
  - abstractions for physical state sensors, e.g. the joint states coming from the robot, photographs
  - abstractions for the method of communication with the GPS agent, e.g. ROS topics
- Controllers:
  - trial controller, for commanding the robot to conduct a series of trials and return a set of trajectories
  - position controller, an in-built PID controller for commanding the robot to reset

Despite the efforts in abstraction, Finn's software is incomplete. Most notably, the RobotPlugin class assumes two physical robots, due to her implementation of a derived controller for the PR2 robot, which has two arms. On one arm, a true PID controller is run and trial torques are sent, while on the other, a dummy PID controller is run and torques are not sent. Use of both PID controllers and torque commands is required for any ROS controller derived directly from RobotPlugin.

In the suite of examples that Finn provides is a derivative C++ class of RobotPlugin, called PR2Plugin, specifically for the PR2 robot. Instead of managing two instances of a one-arm RobotPlugin class, RobotPlugin contains the code for two arms and PR2Plugin merely extends this with more PR2-specific code. Dealing with this was one of the problems in implementing GPS for the LWR.

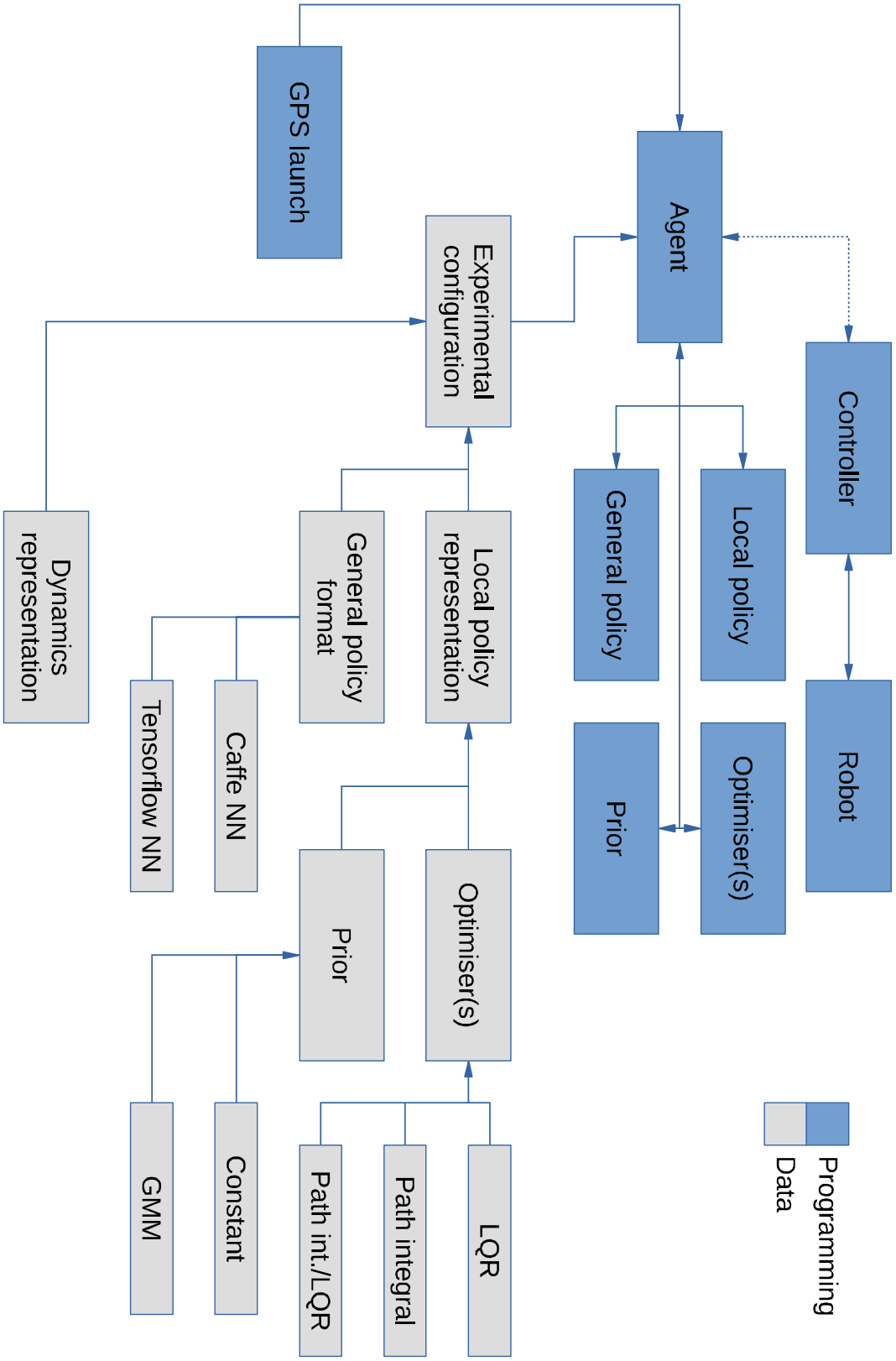


Figure 13: The structure of Finn's implementation of GPS

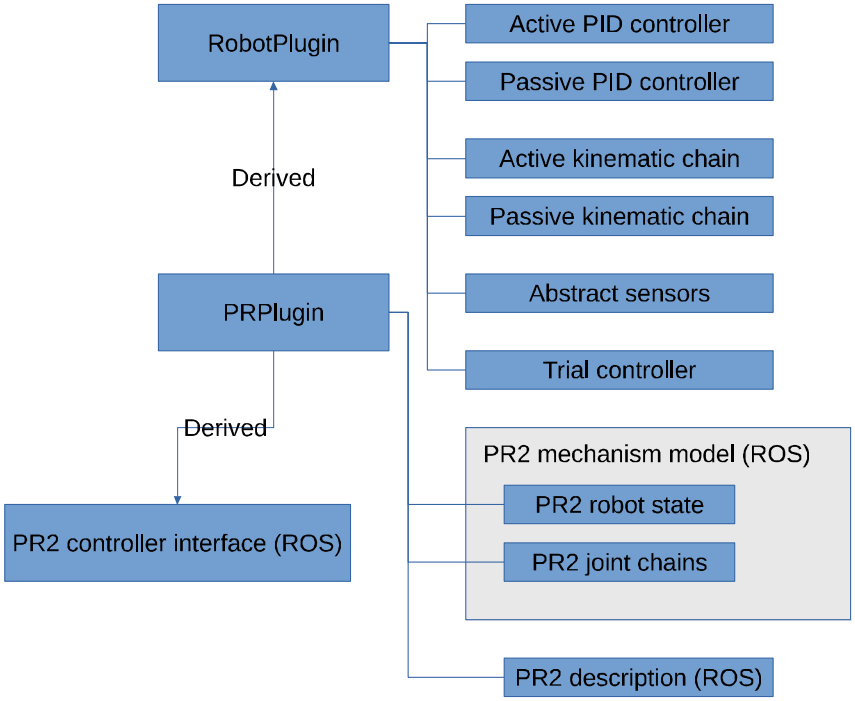


Figure 14: The structure of a PR2 controller based on the RobotPlugin class

3.5 GPS controller for the LWR using ROS Control

Writing a GPS controller for the LWR, based on the PR2Plugin class and derived from the RobotPlugin class was the major pre-experimental work of this thesis. While similar operations must be performed when setting up and updating a controller for PR2 as for the LWR, the specific ways of doing so are quite different. The PR2-specific mechanisms used by Finn's PR2 controller were written as a demonstration of early versions of ROS. Although the PR2 ROS packages have been kept up to date for the sake of backward compatibility, the preferred method for writing ROS controllers (including for the PR2) is now to write for the ROS Control package.

The structure of the ROS controller for interfacing GPS and the PR2 is found in Fig. 14. The PR2Plugin class inherits both from the RobotPlugin class, which abstracts the communication with GPS and the sensing of the state of the robot, and from the pr2_controller_interface class, which is found in the pr2_mechanism_model ROS package.

ROS controllers, including for the old PR2 packages, are based around two functions: the update function, which retrieves states from the robot, calculates everything needed for control and sends actions back to the robot, and the initialisation function, which is of particular interest.

In the initialisation of the PR2Plugin, the names of the key points on the robot (root and tip node names) are retrieved from the parameter server, having been uploaded at the launch of the controller. The controller then tries to initialise solvers for the forward kinematics and Jacobian, based on those parameters. Then, the controller manually iterates over the number of joints in the robot, constructing strings corresponding to the names of all the joints and checking them against the joint names on the parameter server. It does this for both the

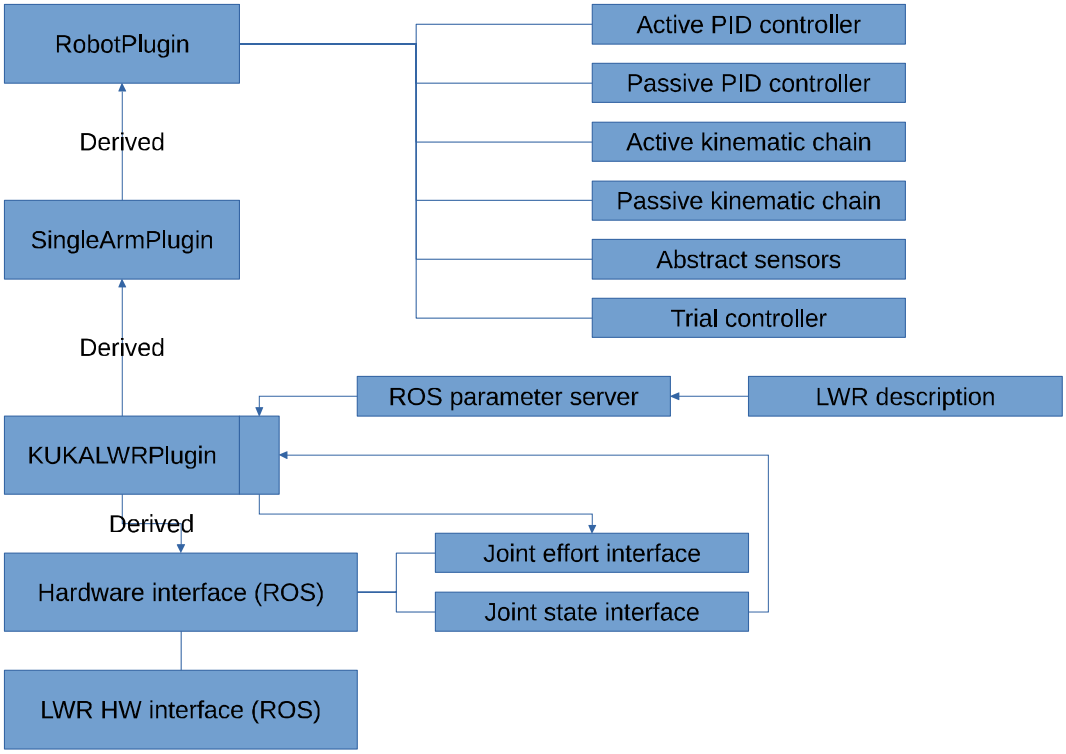


Figure 15: The structure of an LWR controller based on the RobotPlugin class

active and passive arms, partly because the PR2 has two arms, but also because dealing with both arms is a requirement of using the RobotPlugin base class.

Finally, the controller queries the PR2 model with the names of all the joints, and the model returns pointers to the joint states. The active and passive arm torques are explicitly set up as arrays.

ROS Control uses a very different system (see Fig. 15). In the initialisation of an instance of the KUKALWRPlugin class (written for this thesis), the root and tip names are also loaded from the parameter server, but then the XML representation of the URDF model of the robot is also loaded from the parameter server (having been uploaded during launch. The URDF model is then explicitly reconstructed from the XML and stored in URDF model object. The URDF model is then converted to a kinematic tree format. The tree object is then converted to the kinematic chain used by the PR2 plugin. After the solvers are initialised, joint handles, rather than joint states, are populated from the effort joint interface.

The documentation for the existing GPS code prescribes that new ROS controllers for GPS should directly inherit the RobotPlugin class (as with the PR2Plugin class), however, this is not, in fact, possible. Function appropriate to the PR2Plugin class has been included in the RobotPlugin class, namely the use of two arm trial controllers, because the PR2 has two arms. In order to handle this, an intermediate class, SingleArmPlugin, fits between RobotPlugin and KUKALWRPlugin. This third class initialises the member variables of RobotPlugin which pertain to the passive trial arm, but ensures no further interaction with those variables and generates errors if commands are sent to the passive arm, or if data is requested of it.

A second intermediary class was used polymorphously to initialise the GPS PID controller in the KUKALWRPlugin to combat recurring problems moving the end-effector joint in the Gazebo simulation which crashed Gazebo or caused the model to explode.

Both intermediary classes are discussed in Section 5.

3.6 GPS agent for the LWR

The difference between one GPS agent and another is largely just in the method of communication between the agent and the controller. In the case of the LWR agent, its only difference from the PR2 controller was to cease sending commands to the passive arm and expecting replies from the passive arm as part of general updates.

3.7 Target position command utility

Setting up experiments for ROS-based GPS systems demands the use of a GUI written by Finn. Several trajectory optimisation conditions (initial and final state) can be set to run sequentially, which is useful if the full GPS algorithm is being used to train neural network general policies. The GUI allows the user to move the robot to the initial and final states for each condition and allows the user to *set* the initial and final states, but does not offer any way of moving the robot to new states that could be set. For this purpose a small utility was written to publish commands on the GPS reset command ROS topic—desired positions, PID gains and a clamp for the integral term of the PID controller. Figure 16 shows the GPS target-setting UI, the Gazebo front-end and the utility for of commanding the simulated robot.

3.8 Summary

Implementing GPS on the KUKA LWR 4+ required the help of several third-party software packages, including the GPS back-end. However, the development of the ROS controller for GPS proved to be a difficult task.

Figure 16: Configuring experiments with the position controller command publisher

Section 4

Evaluation

This section presents issues around the configuration of the GPS agent (see Section 3.6) and two experiments conducted on the performance of GPS on the KUKA LWR 4+ and analyses the experimental results with respect to expected behaviour. The first experiment (Section 4.2) investigates the trajectory optimisation process detailed in Section 2.3.1 and the derivation of local optimal stochastic policies detailed in Section 2.3.2. The second experiment tests local policies of the type derived in experiment 1 under perturbed initial conditions. Finally, possibilities for future work are discussed.

4.1 Configuration of the GPS agent

The GPS agent (Section 3.6) is the core tool for connecting the components of GPS. Sometimes multiple implementations of a component are made available by Finn, each performing similar or analogous tasks to the others. For example, instead of using the linear-quadratic-Gaussian trajectory optimiser, one might use the Bregman Alternating Direction Method of Multipliers (BADMM) [35] [36]. In the following experiments, only a limited subset of these components is used and will be described in Appendix B. In the nomenclature of Finn's GPS code, these variables are called hyperparameters and are stored on a per-experiment basis. Unless otherwise specified, the following experiments use these hyperparameters.

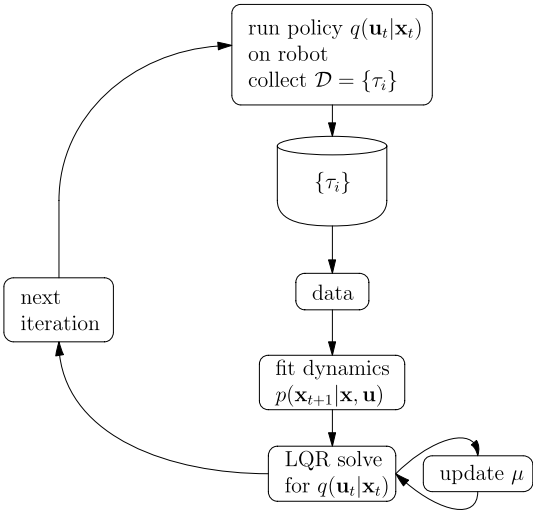


Figure 17: Local policy improvement cycle

4.2 Experiment 1: LQG trajectory optimiser performance

This first experiment uses the "inner" section of GPS, i.e. the local policy optimisation portion (Fig. 17). The experiment aims, first, to demonstrate that the controller and agent are sufficient for the GPS backend to learn optimal policies local to these initial and final configurations and, second, to examine the extent to which this experimental setup allows GPS to find consistent optimised trajectories.

4.2.1 Tasks

Six tasks were set up using the GUI and the default hyperparameters specified in Section 4.1. The tasks were intended to represent a variety of difficulties. In tasks 1 and 3, the initial joint positions are all approximately zero, not all joints change between the initial and final positions and there is a maximum movement of only half the joint range; in task 1, the range movement ranges are rather less than that. In task 2, the movement from initial to final position is little more than a single rotation of the base joint. In the other three tasks, some joints were specifically set close to their limits in order to restrict the possible actions the robot may take—to make areas of the trajectory distributions illegal. Images of the initial and target positions can be found in Appendix A.

Each task has been trained five times from start to finish. Within each task, there are 10 iterations of the optimisation process and within each iteration, seven sample trajectories are taken, from which the best optimised trajectory is chosen to form the optimal local policy. There are 200 steps in each trajectory.

Within each iteration, a mean cost c_i is found from the seven samples at each step and across each of the 200 steps in the trajectory. The cost comprises an action term, half the elemental sum of the three-way element-wise product of a weight vector $\mathbf{w}_u$ with the action $\mathbf{u}$ twice, and a state (position) term of half the squared L2 norm of the final position error in the trajectory, weighted with a scalar factor $w_p = 50$. The value was kept from the PR2 experiment on which these experiments were based.

$$c = \left[\frac{1}{2}\sum\left(\mathbf{w}_u \circ \mathbf{u}_t \circ \mathbf{u}_t\right)\right] + \left[\frac{1}{2}\sqrt{w_p}\,||\mathbf{e}_{t=T}||_2^2\right]$$

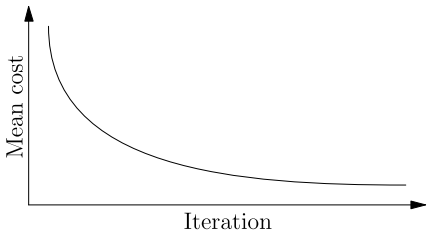


Figure 18: The expected profile of average costs in experiment 1

4.2.2 Performance metrics

Due to the naturally differing lengths of the trajectories, there is expected to be significant difference between the mean action-term costs. Therefore, the primary measures of performance will be the trajectory of mean costs *across tasks*—the mean of means—but also the standard deviation of costs at each iteration, taken across all trials of each task. For a task, a cost-trajectory comprises mean-of-mean costs μ_i.

If the agent is learning, μ will follow an overall downward trend, flattening out as the policy becomes optimal (Fig. 18). However, it should be borne in mind that the illustrative curve is ideal and not realistic. The tasks are stochastic processes and there is much room for deviation from the ideal. Furthermore, the more difficult tasks are expected to take longer to learn and μ will not go down so fast. There may also be flattening at certain points in the learning process, leading to false impressions of an optimal policy, when in fact, the agent can learn to furthere reduce the cost.

The degree to which the absolute final value of μ is representative of a optimal policy depends on the action cost, which depends both on the distance between the initial and final positions and on the difficulty of the task. However, since both the state and action costs rely on squared variables, as the costs approach zero, the policy must be approaching optimality.

The standard deviation of mean costs σ_i is a secondary measure. It is expected that it will reduce as the learned policy approaches optimality.

4.2.3 Results and evaluation

This experiment aimed to ascertain whether the GPS algorithm could reliably produce quality local policies, based on the qualitative shape of the curve of mean costs in each task, the mean costs itself as a measure of quality, and the variance/standard deviation of the actual mean costs from the mean of means. Figure 20 shows the results for each task. Figure 19 is provided to more clearly show the curve of task 1 without the outlying mean costs of the first trial.

While the graphs clearly demonstrate the agent is learning to move the robot in the right direction, the efficacy of this GPS configuration is extremely variable. In this limited test, tasks 1, 2, 5 and 6 do appear to, generally, converge on an optimal policy (Figs. 20a, 20b, 20e and 20f). This is pronounced in task 2. Nonetheless, the considerably lower-cost policies found in tasks 5 and 6 should that the means across trials are not yet representative of the optimal policy.

The final standard deviations in tasks 3, 4, 5 and 6 indicate that, while the agent does learn, it is not yet reliably able to optimise local policies. In particular, in task 6, the standard deviation actually continues to rise until the end of the trajectories.

These high-deviation tasks were specifically chosen to include positions close to the joint limits. This clamps the distribution of allowable trajectories to legal configurations and may, therefore, account for the increased standard deviation from the mean of means.

It is likely that where the cost-trajectories converge, the derived local policy comes close to the true optimal policy within the nine optimisation iterations. In the tasks where the convergence is limited, or the apparent convergence is on a considerably higher mean cost than the best case, the expectation is that further iterations would result in derivation of the optimal policy.

Each trial took approximately 15-20 minutes to run on the available hardware and it was impractical at that time to run further iterations over a variety of tasks.

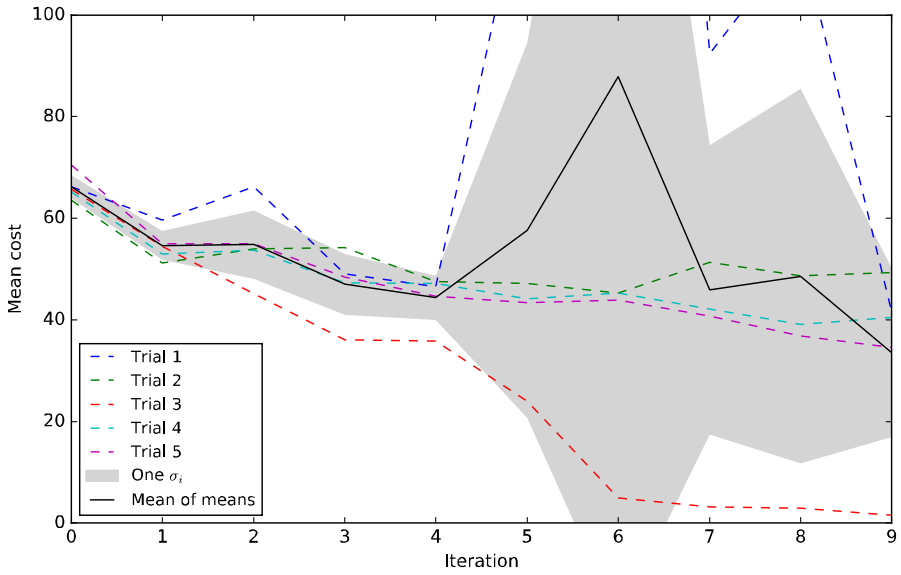
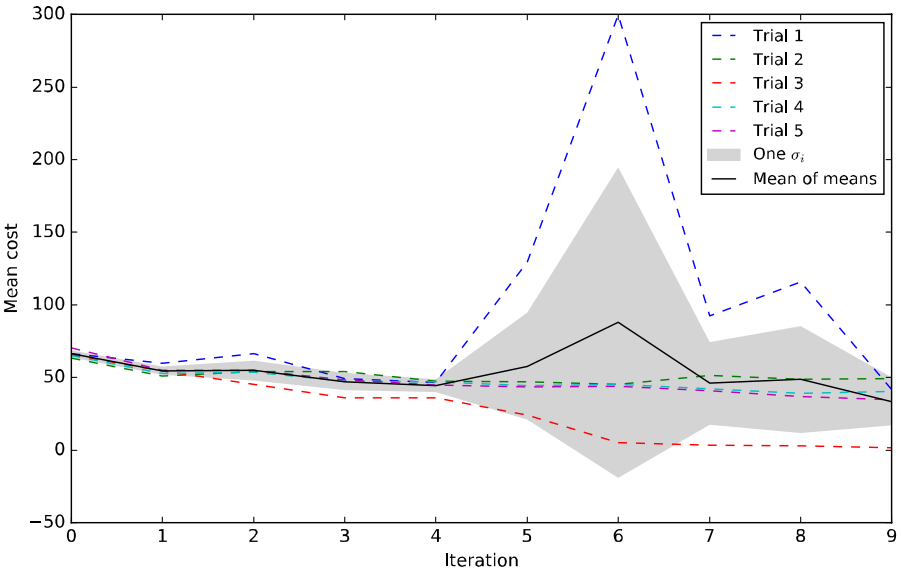
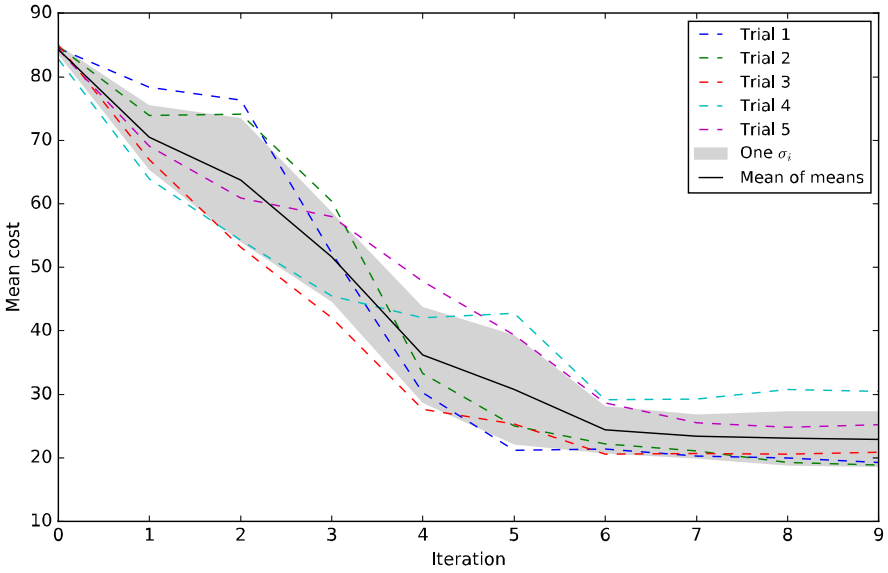


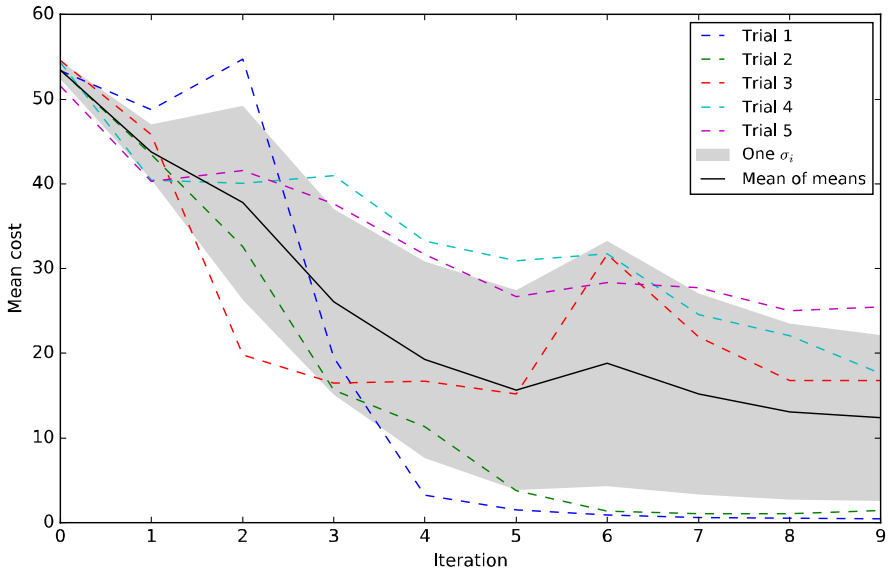
Figure 19: Task 1 results, zoomed in to exclude the high freak costs



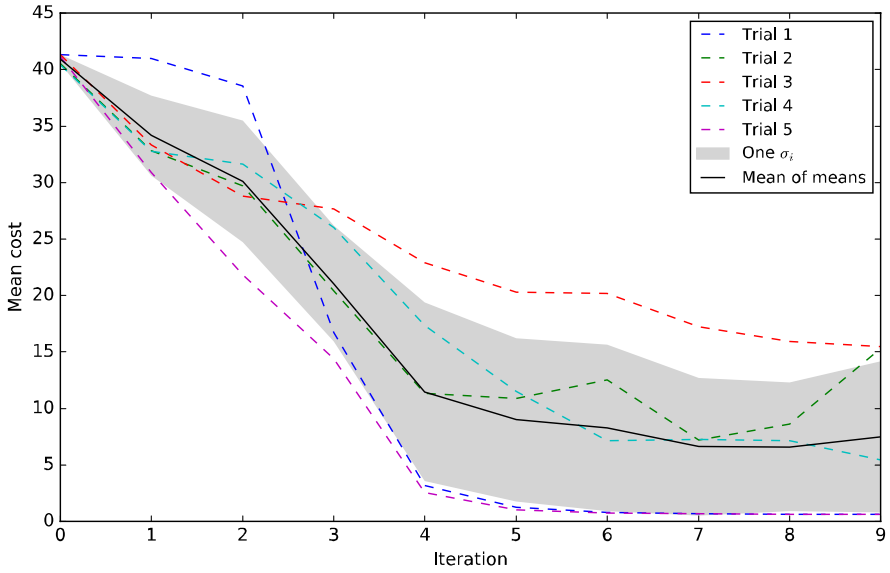
(a) Task 1 results



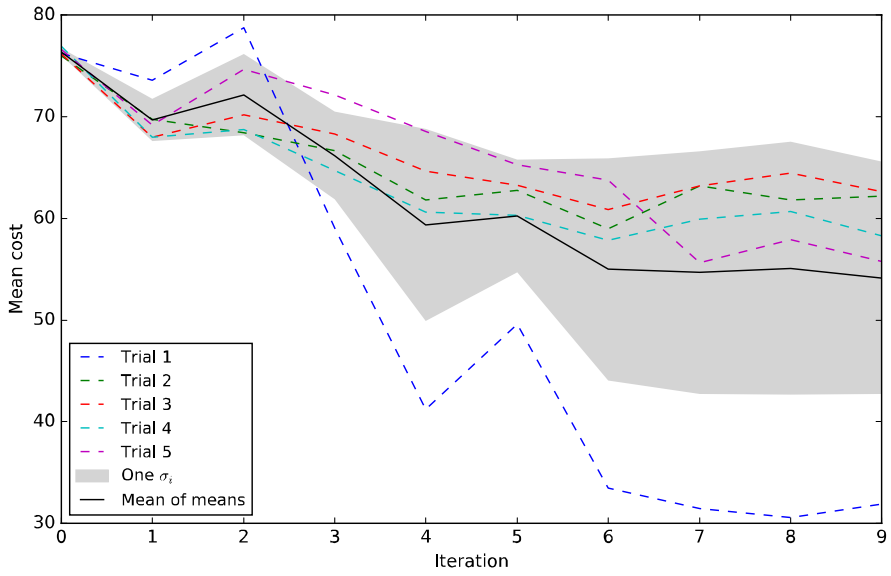
(b) Task 2 results



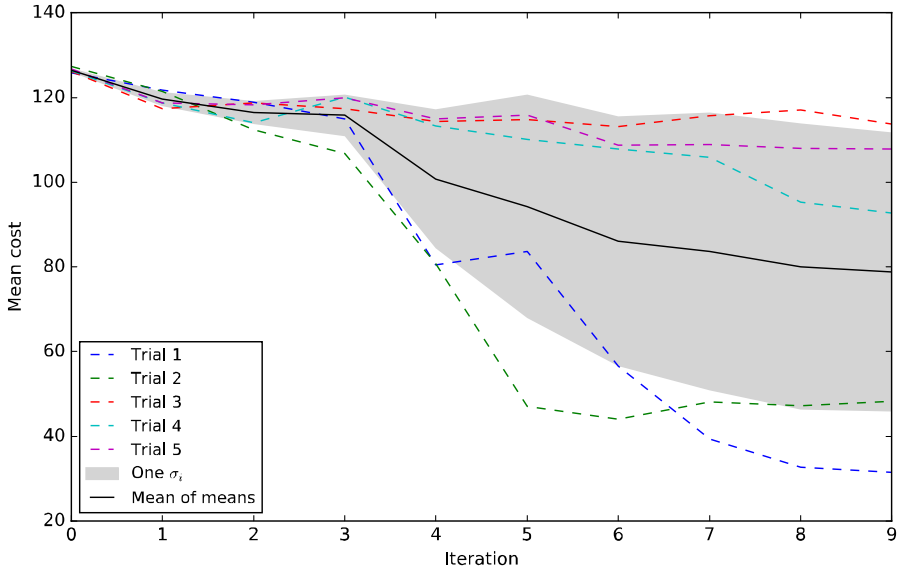
(c) Task 3 results



(d) Task 4 results



(e) Task 5 results



(f) Task 6 results

Figure 20

4.3 Experiment 2: Performance of local policies with regard to perturbed initial configurations

The purpose of this experiment is to evaluate a prior learned local policy in operation on noisy tasks—where the initial position has been perturbed. Once again, the portion of GPS being used can be seen in Fig. 17.

4.3.1 Tasks and expected behaviour

The nature of GPS is to find optimal policies in a *stochastic* setting, i.e. that the trajectories generated under the policy have a random element. The upshot of the stochastic policy is that, instead of generating a single trajectory, the policy will generate trajectories from a spatial distribution. The tasks in this experiment are intended to find how the policy is able to track the target end-effector position given perturbations in the initial end-effector position.

It is expected that the policy will perform well, that is, generally achieve a final position error broadly in line with the final error at the end of training, for initial positions leading to trajectories that fall within the trained trajectory distribution. Once the initial error rises such that this is no longer the case, the absolute final error and standard deviation from the final training error are expected to rapidly rise (see Fig. 21).

Caveat: In order to perturb the initial joint position, the GPS target setup GUI must be used in conjunction with an external position command utility. The position command utility (Section 3.7) sends desired positions to the ROS controller, which then uses GPS's internal PID controller to generate torques. The PID controller is stopped by a position error test against a hard-coded value of 0.385 and a velocity test against a hard-coded value of 0.03. The position figure could well be improved by use of better PID gains, however, finding appropriate gains becomes a trial and error search due to the Gazebo simulation's propensity to crash or explode. The position controller works in joint space only. This made perturbing the initial position exactly impossible. Additionally, the simulated Kuka had a further tendency to slowly fall over when the joints with horizontal axes were nonzero.

The agent configuration used in this experiment is based on task 2 in experiment 1, due to its reliable convergence. A local policy was trained with the configurational changes in Table 2 in order to maximise the quality of the policy.

The training profile of the policy can be seen in Fig. 22. Interestingly, the mean cost can be seen to achieve the same level approximated in experiment 1, task 2, collectively, by the 10th iteration, before improving further by the 20th, albeit at a slower learning rate.

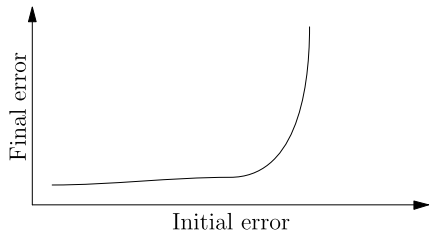


Figure 21: The expected profile of final error in experiment 2

Hyperparameter	Original value	New value
`agent.T`	200	300
`algorithm.iterations`	10	20
`config.num_samples`	7	10

Table 2: Modifications to experiment 1, task 2 for a high-quality, convergent policy in experiment 2

For each perturbation, the experiment will be run five times, allowing for a graph of mean final errors and standard deviations. The final errors are represented in the same way as the learning position error, where **e** is the final error:

$$\frac{1}{2}\sqrt{w_p}\,||\mathbf{e}||_2^2$$

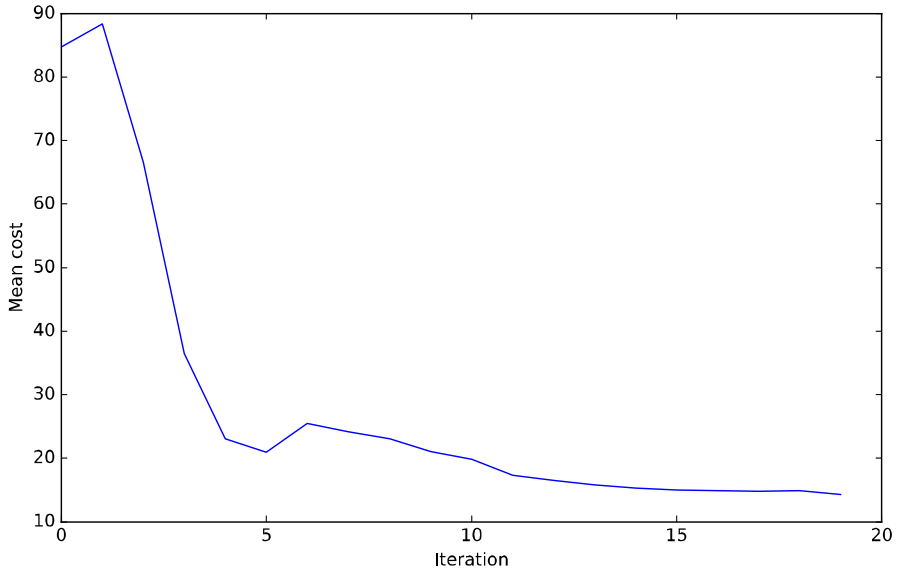


Figure 22: The training profile of the policy for experiment 2, showing the improvement from the changed hyperparameters

4.3.2 Results and evaluation

This experiment aimed to show that the agent can learn using one set of trajectories and still perform reasonably in off-policy control within the boundaries of the training samples' trajectory *distributions*. The results are seen in Fig. 23.

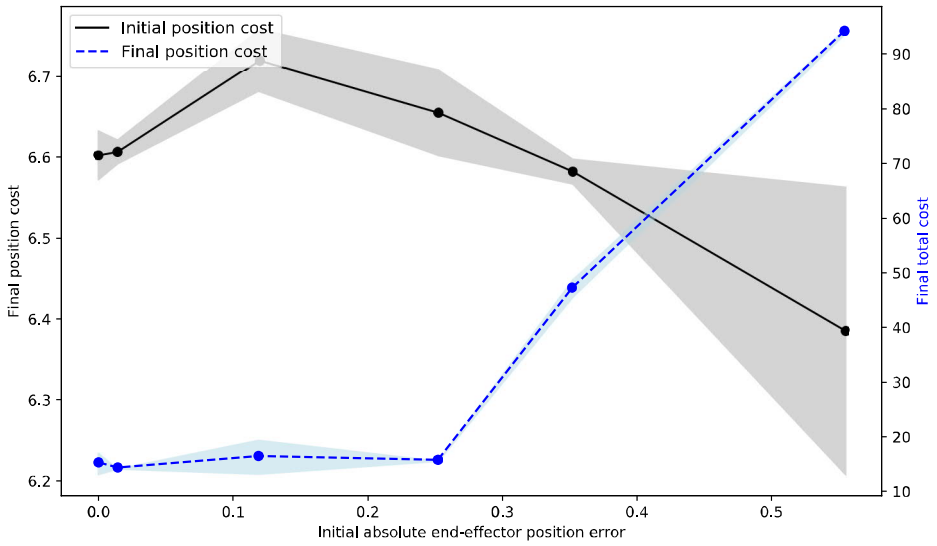
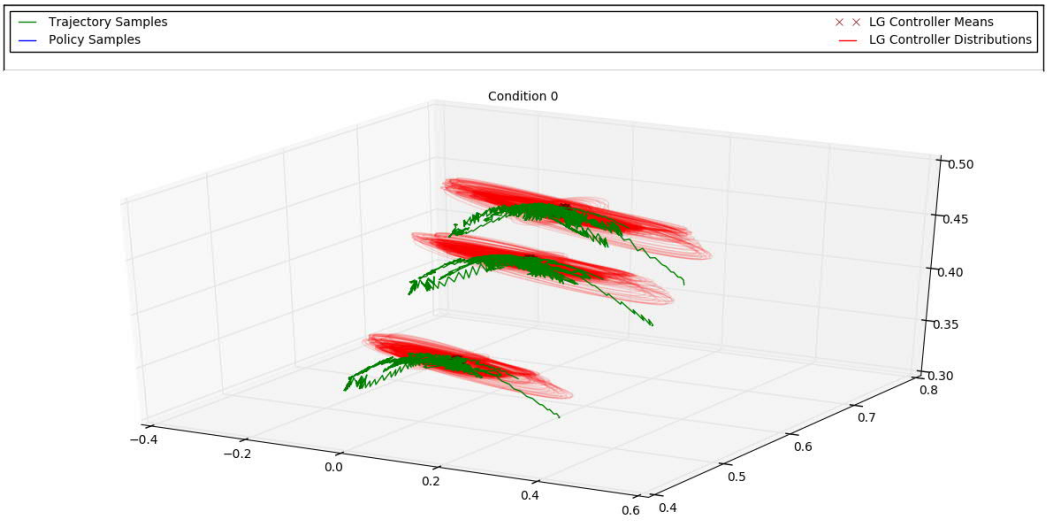


Figure 23: Experiment 2 shows that, contrary to expectations, the agent effectively guides the robot to its target configuration regardless of the deviation in initial position from the one used in training, however, when taking into account the full cost function of the trajectory used to optimise the trajectories in training, we see the expected performance profile in the final cost

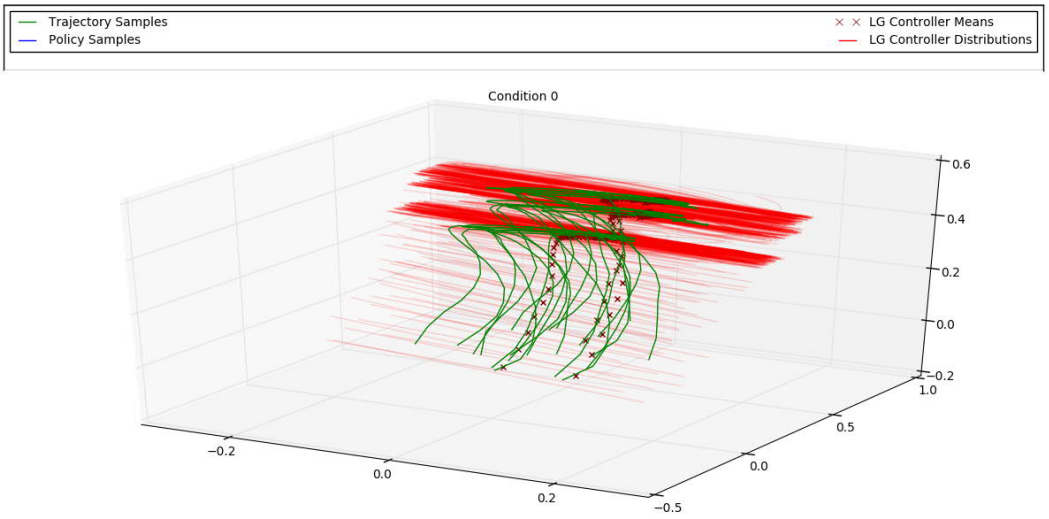
The results are contrary to expectations in that the optimiser can effectively learn the dynamics of the robot and that the agent can guide the end-effector to its destination to much the same degree as when the canonical initial positions are used. In fact, the position cost is slightly reduced. The expected profile does emerge when the full final *cost* is graphed (Fig.23 right-hand axis). The difference in the full cost and the position error cost is the action cost. This means that although the target position can be reached, the joint trajectories get progressively worse with initial position deviation, i.e. the sum of the squared torques sent to the robot is larger than it needs to be.

Watching the simulation in early trials of the KUKA controller would indicate that these sub-optimal trajectories are noisy—that is, jerky, with vibration of the joints throughout. This could have severe physical consequences if these policies were to be run in this manner on a real LWR. A comparison between bad, jerky trajectories and good, smooth trajectories is seen in Fig. 24.

It is interesting that the final position errors do not follow the expected profile. This may mean that the trajectory distributions are large in comparison with the size of the robot. It may be that better local policies are found with narrower distributions.



(a) Jerky trajectories, taken from the first iteration of experiment 1 task 1



(b) Smooth trajectories, taken from the tenth iteration of experiment 1 task 1

Figure 24: Poor policies and poor dynamics result in trajectories in which the LWR joints chop backwards and forwards at high speed, possibly increasing wear and tear or breaking the joints outright

Section 5

Discussion

5.1 Difficulty in writing the ROS controller

Neither ROS Control nor the existing GPS controller code is well documented and the comments in the latter are often cryptic. Working out how to accomplish the relatively simple task of setting up the LWR controller took weeks of poring over the code and the ROS help files.

Three big problems also occurred. First, the controller continually crashed when the LWR controller plugin was derived straight from the RobotPlugin class. This was eventually found to be because the passive arm PID controller was receiving commands at the PR2Plugin level, but the KUKALWRPlugin class was not passing them on to a trial controller. Part of the attraction of using ROS Control is that it is abstract: what works for one robot ought to work for another similar robot, just as long as the similar robot has a ROS hardware interface. The use of two arms in the RobotPlugin class, which was not written with ROS Control in mind, breaks that mechanism for any one-armed robot.

The solution was to insert an intermediate class between RobotPlugin and KUKALWR-Plugin, which was derived from RobotPlugin. The sole purpose of the SingleArmPlugin class is to feed the objects pertaining to the passive arm dummy data, or to force it to not expect updates. In theory, this means that any single-arm robot can now use the KUKALWR-

Plugin class as a GPS controller, so long as it has a ROS hardware interface that exposes an effort joint interface.

The second big problem was that upon testing the GPS internal PID controller on the simulated LWR, the tip joint would barely turn. A large variety of different PID gains were tried, but the integral and derivative gains had little effect and raising the proportional gain above about 7 (in comparison with 500, or up to 1200, for the base joint) resulted in the simulation either crashing, or the simulated model exploding, or both.

Third, sometimes the trajectory charted by the PID controller would simply not finish. In initial tests with the position commander (see Section 3.7), this made it impossible to send further position commands to the robot, however, it was worse once experiments with the trial controller began. At the end of each trajectory generation, the trial controller uses the PID controller to return the robot to its initial state for that experiment. The agent expects a report from the ROS controller on the state of the robot and when it does not receive one, because the movement is not finished, the agent crashes.

In fact, the second problem was causing the third. The PID controller decides when it has finished testing the norm of the joint position errors against a hard-coded threshold and because the tip joint would not turn, the threshold was never reached, the PID control never finished and the state of the robot was never sent to the agent.

The cause of the sticky joint was not discovered. At first it was thought that, since this was a simulation, the joint might be turning a zero-mass end-effector link, however, on inspection of the Gazebo model, this was found to be incorrect.

Because the tip joint simply would not turn properly, a hack was introduced. A subclass of the position controller was written where the test for completion of the movement used the norm of only the first six joints. This is not satisfactory because both the position and trial controller do attempt to control the sticky seventh joint. Sometimes it moves one way, but not the other. This will introduce errors into the initial position when the trial controller resets and will result in suboptimal local policies with higher final costs.

5.2 Further work within the thesis mandate

The pratical work of thesis has avoided two major features of GPS: the training of general policies and the seeding of the local optimal policy generator with qualitatively better guiding trajectory distributions, from human demonstration with a real LWR robot, for example. Furthermore, the experiments detailed above have thrown up two things which should be investigated: a cost function that takes into account non-final position errors, and the unexpected phenomenon that perturbing the initial state of the robot does not appear to affect how close the agent can get to the desired final state.

5.2.1 General policies

First and foremost, the KUKA GPS controller needs to be tested in totality. In the experiments conducted in this section, only the local policy optimisation has been tested. Full GPS relies upon the local policies generated being used to train a more general policy. In Finn's implementation of GPS, the general policy may be a neural network maintained by the Caffe or Tensorflow libraries, but it may in fact be any any sort of general policy. Due to the length of time it has taken to develop the controller and agent, and to run the experiments conducted so far, this has not been possible for this thesis.

It is expected that the neural network policy would perform well around the training trajectories, merely quite well elsewhere in the configuration space covered by the training trajectory distributions and poorly outside those distributions. As evidenced in experiment 2, this may, in reality, result in jerky trajectories and high action costs outside the distributions.

It is recommended that future workers on this software experiment with different policy representations, both outside neural networks (for example, one might ask, just how much better is a deep NN than a simple linear combination) and in the use of different forms of neural network. Particularly once the agent is in operation in changeable environments, it would be interesting to see how its policy might be improved in-situ by use of long/short-term memory networks to give weight to the emerging environment.

5.2.2 Effects of human-demonstrated guiding distributions

An aspect of GPS inherent to the local policy improvement is the ability to accept prior guiding samples. This is exactly what happens when the accumulating dynamic Gaussian mixture model is used as a prior to the determination of the transition dynamics. What this thesis does not explore is the possibility for totally external priors to be used, for example smooth trajectories generated by human movement of the real KUKA LWR 4+ robot. Much research is being undertaken right now on learning from human demonstration and GPS neatly encapsulates this. In Levine and Koltun's original paper, external initial trajectories were used for their walking and running demonstrations.

5.2.3 Cost function improvement

The experiments in this thesis pulled out both a logarithmic positional term of the cost function used by Levine and Finn on the PR2 robot [36] (leaving only the L2 norm term) and all the positional costs for non-final steps in the trajectory. Both aspects were removed while experimenting to achieve something close to convergence of the final costs.

If these costs can be returned to the cost function, it is expected that learning rate would increase because they encourage the policy to move more directly towards the target state. This may also iron out some of the joint vibration mentioned in Section 4.3.2.

5.2.4 Determination of the reason for low final position error for deviating initial positions

That the policy in experiment 2 managed to consistently move the end-effector to its targeted final position was a significant surpise. It is not clear exactly why that was the case and the wide spatial representation of the local policy distributions alluded to above is something generated internally by the policy improvement process, rather than something that is explicitly set. It was expected that errors in final position would be diverse within a large interval. An informal test was undertaken of an initial-position deviation that was small in Cartesian space, but very large in joint space, with the same results—the position cost term was similar to all the other trials, while the action cost was understandably huge. This phenomenon ought to be explored, because it calls into question the need for a neural-network policy.

Section 6

Conclusion

The purpose of this thesis was to demonstrate the effectiveness of the guided policy search (GPS) method of reinforcement learning (RL) for learning reaching tasks with the KUKA LWR 4+ robotic arm, both in qualitative (fidelity of the reaching movement) and quantitative (sample efficiency) terms. To this end, a ROS controller was developed to transfer commands from an existing implementation of GPS, via an existing hardware interface, to a simulation of the robot, and to transfer position and velocity data from the robot back to the algorithm back-end. Additionally, a protocol-specific agent was written for the GPS back-end and a set of experiments was written to test the system (Section 3).

Guided policy search is an advanced RL algorithm comprising two main components: an interchangeable locally optimal policy generator and general policy approximator (Section 2). This thesis details the standard GPS algorithm, in which the local policy generator is a linear-quadratic regulator (LQR) that iteratively optimises a collection of previously generated trajectories, arriving at a dynamic trajectory distribution in a local area of the robot's state-space, and in which the general policy approximator is a feedforward neural network. This arrangement is seen to achieve sample efficiency, despite the need to ultimately train a model-free general policy, by using a small number of samples drawn from nominally inefficient trajectory distributions to feed the LQR and thereby derive better local policies without recourse to running general policies on the robot.

Nevertheless, the extensions to GPS discussed in Section 2.3.3 clearly show that by replacing the LQR component with different locally optimal policy generators, GPS can be improved not only in the sample efficiency which this thesis specifically aimed to demonstrate, but also in combating issues not targeted by the original GPS formula, such as contact-rich environments.

Two experiments were undertaken on the finished software system. First, local policies were trained for six different reaching tasks (Section 4.2). These policies were shown to improve in broadly the expected manner, thus confirming that GPS is suitable for basic reaching tasks on the LWR 4+. However, the learning time was insufficient, particularly for the more difficult tasks, for the policies to reach full optimisation. This was graphically shown in the second experiment, where a policy was trained on one of the tasks in the first experiment, and continued to improve for the increased number of optimisation iterations.

The second experiment studied the performance of a trained policy under perturbed initial conditions, where the expectation was, in the average, a final position error increasing exponentially with the deviation from the initial position used in training. The agent defied this expectation, however, in that the final position errors were similar regardless of the initial position error. This phenomenon threw into question the need for training a general policy, at least with the tasks the LWR 4+ is capable of performing. Nonetheless, the deviation in the online initial position from the training initial position did affect the smoothness of the trajectory distributions.

This thesis shows that GPS is capable of training an agent to complete elementary reaching tasks in a matter of 15 minutes on mid-range PC hardware using fewer than 100-150 sampled trajectories. The basic GPS configuration used in this thesis has been shown to produce slow trajectories under deviating initial conditions after training has finished. Nonetheless, the data gathered for this thesis is not yet adequate to fully assess GPS's efficacy. The speed with which the local policies were learned, even under these restricted conditions, makes GPS an attractive research topic; deeper exploration of the experimental hyperparameters and additions to the algorithm may provide increased adherence to the desired target position, increased sample efficiency and smoother operational trajectories further away from the central trajectory of the trained trajectory distribution.

In particular, the use of an adversarial component, as discussed in Section 2.3.3, has not only been successful in increasing GPS's robustness to disturbances, but also in generating smoother, lower-energy trajectories in humanoid robot gaits [37]. Therefore, an adversary in the local optimal policy generator may be able to iron out the non-smooth trajectories that local policies generated in experiment 2, under perturbed initial conditions.

In summary, GPS is a powerful tool for the practitioner of intelligent robots, due to its sample efficiency and extensibility. However, it may be that standard GPS is only appropriate in test environments and that further research is required before it is ready for operation.

References

[1] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, 1st ed. Springer-Verlag Berlin Heidelberg, 2008, ISBN: 978-3-540-30301-5.

[2] I. F. of Robotics, *World Robotics 2017 (Executive Summary)*, Sep. 17, 2017.

[3] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24, ISBN: 978-1-58603-780-2.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning*, 2nd (draft). MIT Press Cambridge, MA.

[5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943, ISSN: 1522-9602. DOI: 10.1007/BF02478259.

[6] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain," *Psychological Review*, pp. 65–386, 1958.

[7] E. R. Caianiello, "Outline of a theory of thought-processes and thinking machines," *Journal of Theoretical Biology*, vol. 1, no. 2, pp. 204–235, Apr. 1, 1961, ISSN: 0022-5193. DOI: 10.1016/0022-5193(61)90046-7.

[8] (). Rev. Mod. Phys. 34, 123 (1962) - The Perceptron: A Model for Brain Functioning. I, [Online]. Available: https://journals.aps.org/rmp/abstract/10.1103/RevModPhys.34.123 (visited on 09/26/2018).

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[10] S. Levine and V. Koltun, "Guided Policy Search," in *Proceedings of Machine Learning Research*, Feb. 13, 2013, pp. 1–9.

[11] C. J. C. H. Watkins, "Learning from Delayed Rewards," King's College, Oxford, 1989.

[12] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 2010, ISBN: 978-0-691-14668-3.

[13] ——, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957, ISSN: 0095-9057.

[14] M. J. Roe, "Stock Market Short-Termism's Impact," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 3171090, Aug. 13, 2018.

[15] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969, ISBN: 978-0-262-63022-1.

[16] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug. 1, 1988, ISSN: 1573-0565. DOI: 10.1023/A:1022633531479.

[17] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, *et al.*, "Policy Gradient Methods for Reinforcement Learning with Function Approximation.," in *Proceedings of the Neural Information Processing Systems Conference*, vol. 99, 1999, pp. 1057–1063.

[18] M. P. Deisenroth, G. Neumann, and J. Peters, "A Survey on Policy Search for Robotics," *Found. Trends Robot*, vol. 2, no. 1–2, pp. 1–142, Aug. 2013, ISSN: 1935-8253. DOI: 10.1561/2300000021.

[19] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834–846, Sep. 1983, ISSN: 0018-9472. DOI: 10.1109/TSMC.1983.6313077.

[20] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, Robotics and Neuroscience, vol. 21, no. 4, pp. 682–697, May 1, 2008, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2008.02.003.

[21] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *Proceedings of the ... IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 1, 2012, pp. 4906–4913, ISBN: 978-1-4673-1737-5. DOI: 10.1109/IROS.2012.6386025.

[22] S. Levine and P. Abbeel, "Learning Dynamic Manipulation Skills under Unknown Dynamics with Guided Policy Search," *Advances in Neural Information Processing Systems*, vol. 27, p. 1, 2014.

[23] O. Ogunmolu, "A dynamic game approach to training robust deep policies," Feb. 15, 2018.

[24] S. Levine and P. Abbeel, "Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 1071–1079.

[25] S. Levine, N. Wagener, and P. Abbeel, "Learning Contact-Rich Manipulation Skills with Guided Policy Search," in *International Conference on Robotics and Automation (ICRA)*, 2015.

[26] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with MPC-guided policy search," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden: IEEE, May 2016, pp. 528–535, ISBN: 978-1-4673-8026-3. DOI: 10.1109/ICRA.2016.7487175.

[27] P. Ennen, P. Bresenitz, R. Vossen, and F. Hees, "Learning Robust Manipulation Skills with Guided Policy Search via Generative Motor Reflexes," Sep. 15, 2018.

[28] J. Luo, R. Edmunds, F. Rice, and A. M. Agogino, "Tensegrity Robot Locomotion Under Limited Sensory Inputs via Deep Reinforcement Learning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, Australia: IEEE, May 2018, pp. 6260–6267, ISBN: 978-1-5386-3081-5. DOI: 10.1109/ICRA.2018.8463144.

[29] W. Montgomery, A. Ajay, C. Finn, P. Abbeel, and S. Levine, "Reset-Free Guided Policy Search: Efficient Deep Reinforcement Learning with Stochastic Initial States," Oct. 4, 2016.

[30] Y. Chebotar, M. Kalakrishnan, A. Yahya, A. Li, S. Schaal, and S. Levine, "Path integral guided policy search," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3381–3388. DOI: 10.1109/ICRA.2017.7989384.

[31] A. Yahya, A. Li, M. Kalakrishnan, Y. Chebotar, and S. Levine, "Collective robot reinforcement learning with distributed asynchronous guided policy search," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver, BC: IEEE, Sep. 2017, pp. 79–86, ISBN: 978-1-5386-2682-5. DOI: 10.1109/IROS.2017.8202141.

[32] C. Finn, M. Zhang, J. Fu, X. Tan, Z. McCarthy, E. Scharff, and S. Levine, "Guided Policy Search Code Implementation," 2016, Software available from rll.berkeley.edu/gps.

[33] K. AG, *KUKA LWR*.

[34] G. Schreiber, A. Stemmer, and R. Bischoff, "The Fast Research Interface for the KUKA Lightweight Robot," May 3, 2010.

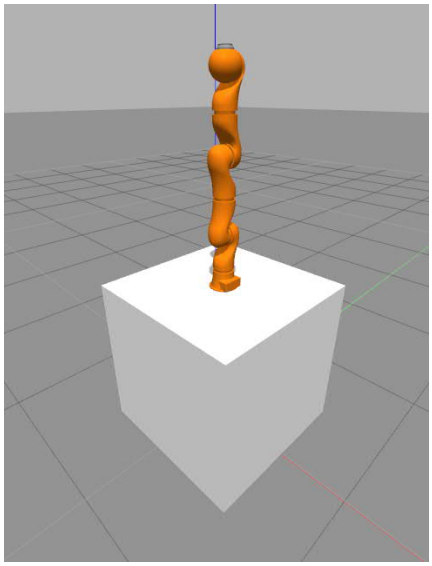
[35] H. Wang and A. Banerjee, "Bregman Alternating Direction Method of Multipliers," Jun. 13, 2013.

[36] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-End Training of Deep Visuomotor Policies," Apr. 2, 2015.

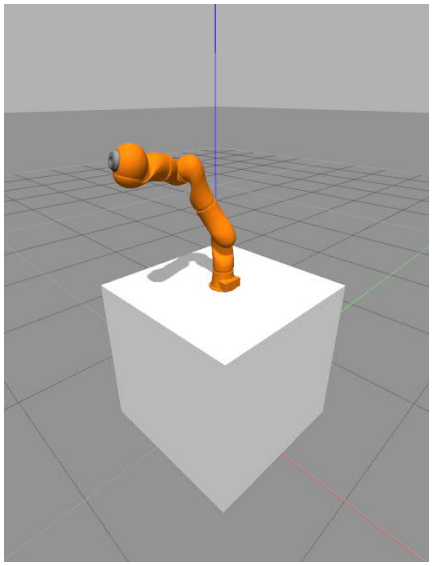
[37] J. Merel, Y. Tassa, D. TB, S. Srinivasan, J. Lemmon, Z. Wang, G. Wayne, and N. Heess, "Learning human behaviors from motion capture by adversarial imitation," Jul. 7, 2017.

Appendix A

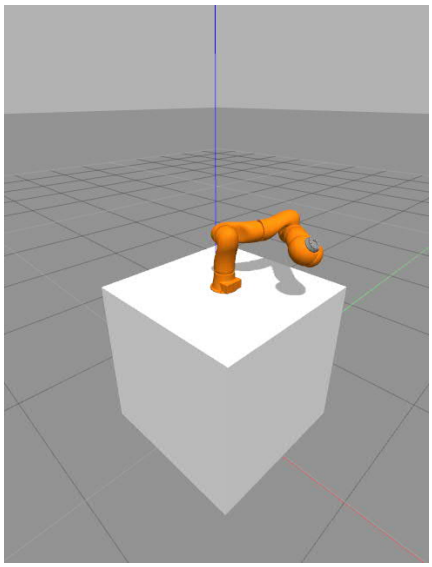
Experiment 1 initial and final configurations



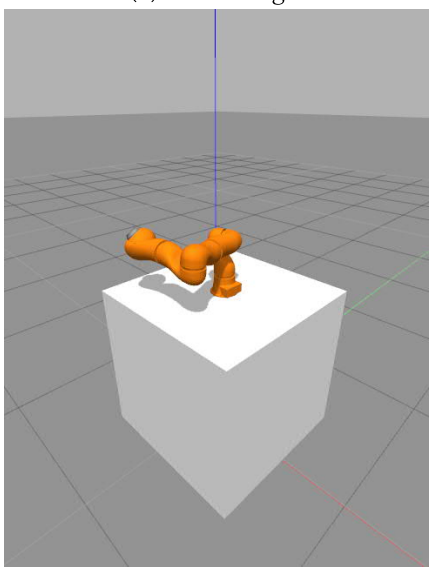
(a) Task 1 initial



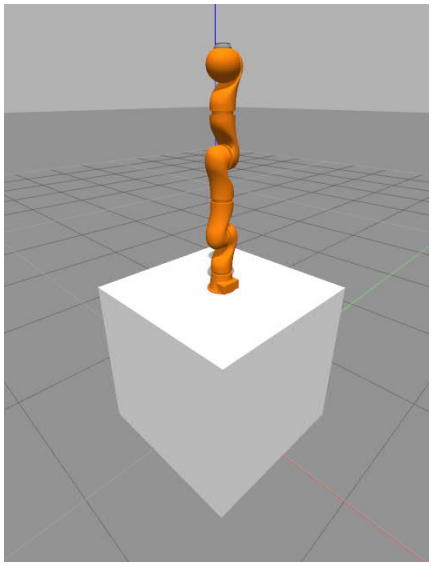
(b) Task 1 target



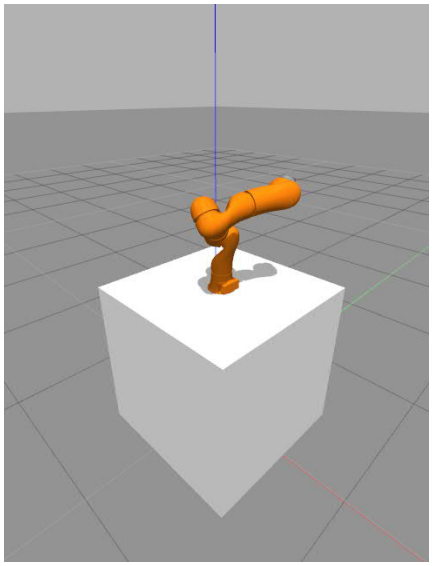
(c) Task 2 initial



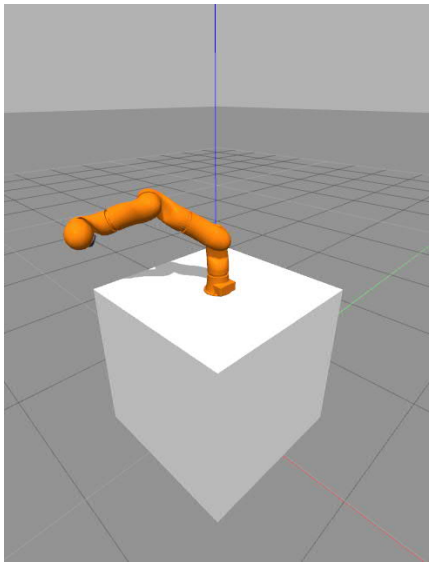
(d) Task 2 target



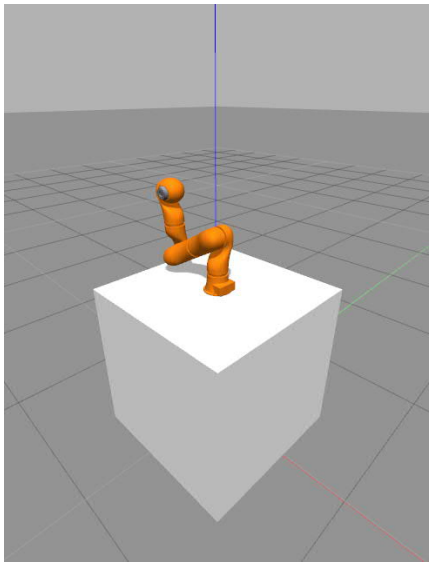
(e) Task 3 initial



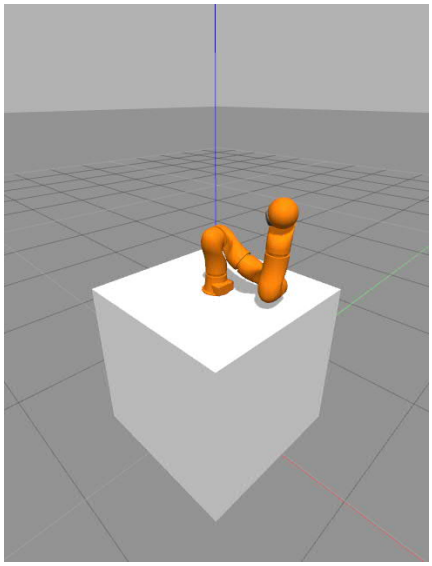
(f) Task 3 target



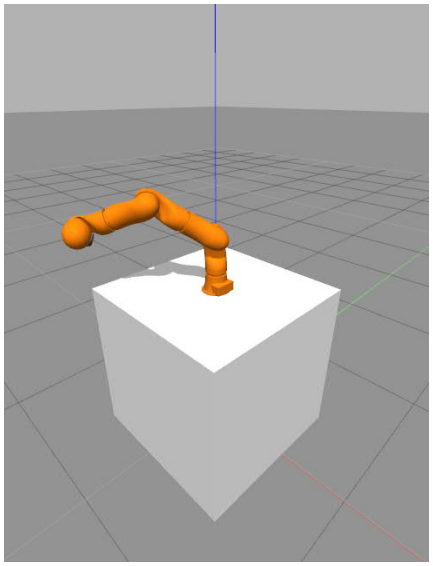
(g) Task 4 initial



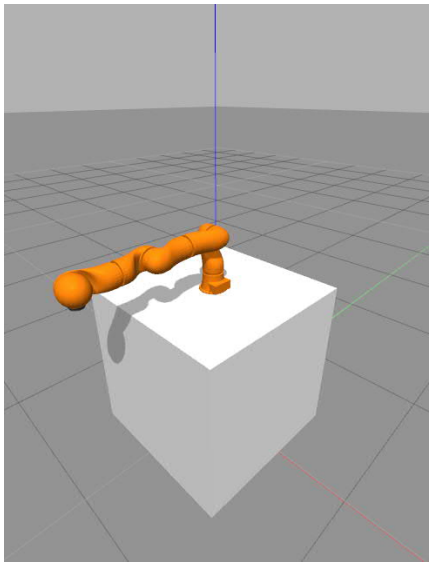
(h) Task 4 target



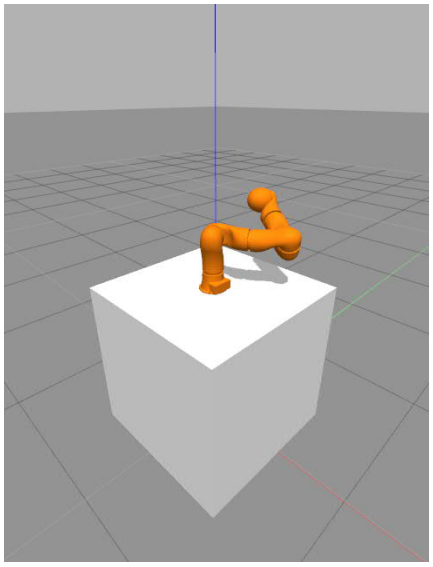
(i) Task 5 initial



(j) Task 5 target



(k) Task 6 initial



(l) Task 6 target

Appendix B

Agent configuration

Component/ Variable	Description	Default value(s)
EE_POINTS	At least three point offsets are required by the GPS backend to ensure that the end-effector not only reaches the correct position, but attains the correct orientation. The world-space end-effector position is subtracted from the positions of these points at the end of local policy training.	$\begin{pmatrix} 0.02 \\ -0.025 \\ 0.05 \end{pmatrix}$, $\begin{pmatrix} 0.02 \\ -0.025 \\ -0.05 \end{pmatrix}$, $\begin{pmatrix} 0.02 \\ 0.005 \\ 0.0 \end{pmatrix}$
PR2_GAINS	A 7-vector of scalar gains, one for each torque/joint of the LWR. The GPS backend is extremely sensitive to these gains; too high a gain causes the Gazebo model to explode, while too low a gain prevents the joint from moving at all. These were set by trial and error. The gains are subsequently updated by GPS internally.	$(96, 48, 20, 14, 6, 6, 12)$
agent	Details of which agent to use and top-level configuration	
type	Name of the agent script—written for this thesis	AgentROSControlArm
dt	Step size [s]	0.05
T	Length of the trajectory [steps]. In initial tests, considerably better results were achieved using 200 steps, rather than the 100 used by Finn's PR2 demonstration, however, this does increase the time required to run each iteration of the trajectory optimisation stage and reduces the data efficiency of the algorithm	200

Component/ Variable	Description	Default value(s)
state_include	A list of the internal variables comprising the GPS state	Joint angles, joint velocities, end-effector points, end-effector point velocities
algorithm	Details of the policy-improvement algorithm to be used by GPS	
type		AlgorithmTrajOpt
iterations	The number of full iterations of optimisation	10
init_traj_distr	Set-up for the DDP initialisation of the linear quadratic regulator	
init_gains	The initial joint gains	$\frac{1}{\text{PR2\_GAINS}}$
init_var	The variance of the initial trajectory distribution. This is key to both getting the LWR moving at the start of the learning process and maintaining enough stability that the Gazebo simulation does not explode. The default value was found by experimentation.	30
stiffness	Initial stiffness of the joints. Important to get the joints turning in the initial distributions before the true dynamics begin to be discovered.	60
stiffness_vel	Initial velocity stiffness. Unchanged from the PR2 demonstration.	0.25
cost	The weighted sum of the cost terms defined below.	
weights	The external weights of each cost term.	[1, 1]
dynamics	Specifies the type of dynamic model prior used to optimise the trajectories. Permanently set in this thesis.	Maximum 20-cluster Gaussian mixture model.
torque_cost	A term of the LQR cost function intended to minimise the L2 norm of the action. This results in smoother trajectories after optimisation.	
wu	The internal vector weight of the action per joint	$\frac{5 \times 10^{-4}}{\text{PR2\_GAINS}}$

Component/ Variable	Description	Default value(s)
fk_cost2	A term of the LQR cost function intended to minimise the L2 norm of the final end-effector position error. This term can be configured to minimise intermediate error costs too, however, including those terms was experimentally found to destabilise the learning process.	
wp	The internal vector (length T) of weights per trajectory step.	[1..1]
l1	The internal weight of the L1 norm sub-term.	0
l2	The internal weight of the L2 norm sub-term.	1
wp_final_multiplier	The internal weight of the norm on the final step of the trajectory	50
ramp_option	Specifies additional weighting profile across each time-step. May be a linear ramp, placing more emphasis the later steps in the trajectory, or a final-only "ramp", where only the final step is considered.	RAMP_FINAL_ONLY
config	Connects above options, the optimisation algorithm and the agent.	
num_samples	The number of trajectory samples used on each iteration to improve the dynamic model.	7 (changed from 5 experimentally)