Data-Efficient Reinforcement Learning for Autonomous Helicopters

MEng project report

Pedro Antonio Martínez Mediano

Supervised by Marc Deisenroth and Murray Shanahan

December 2, 2014

Abstract

This work is framed within the 2014 Reinforcement Learning Competition, an annual gathering in which researchers and students in the field of Reinforcement Learning compete in a variety of problem domains. In this report we describe the basic aspects of Reinforcement Learning and the problems proposed by the Competition. Of those, we tackle the problem of autonomous helicopter control, which consists of learning a controller in an environment with unknown, non-linear dynamics and a black-box reward function without any previous knowledge. We build upon the state-of-the-art, data-efficient PILCO algorithm for Reinforcement Learning and propose modifications to suit this specific problem. We were able to successfully learn a controller to perform different aerobatic trajectories. Our method provides an improvement of several orders of magnitude in terms of number of agent-environment interactions compared to the winners of previous Competitions. This approach drastically reduces the experience time needed to learn a controller, while achieving a performance similar to the most sophisticated methods available, showing once again that model-based Reinforcement Learning is a feasible option to learn effective controllers for real-world applications.

Acknowledgements

I can't overstate how grateful I am to Marc Deisenroth for accepting me as a student for this project. Not only he accepted me as an external, international and different-background student, but he seemed to have found some kind of joy in it, for which I feel deeply flattered. Marc's succinct (and always correct) comments have guided my way through this project from the very beginning. His patience and advice are greatly appreciated.

Many thanks to Murray Shanahan for his assistance and for putting up with all my unusual cases in the PhD admission process. I will do my best to prove worthy of these efforts in the years to come.

Thanks to Duncan Gillies and Danny Segal for their assistance during the administrative procedures. Thanks to Krysia Broda, Amani El-Kholi and the Department of Computing PhD admissions committee for patiently waiting for the result of this report.

Thanks to Julio Parra for drawing me to the art and business of Machine Learning, and to Paula Sola for keeping my drive for this project alive. Cheers to my colleagues Ibles Olcina, Arturo Miquel, Luke Wahley, Sergio Hernández, Daniel Hernández, Hlynur Sveinbjornsson and Sally Dacie, who helped me survive this year of intense work and scarce rock'n'roll.

Finally, special thanks to Juan José (JJ) Gómez Cadenas for his unconditional support from the start of my career. His guidance and advice will always be invaluable.

Contents

1 Introduction

The development of intelligent algorithms and autonomous machines has been and still is a long-standing goal in the field of Computer Science. In this project we explore and study the state-of-the-art of a branch of Artificial Intelligence known as **Reinforcement Learning**.

Reinforcement Learning is a particular approach to the development of intelligent, autonomous agents that learn through interaction with their environment. Reinforcement Learning was initially inspired in the way humans learn, by interaction, trial-and-error and associations between what we can see in our environment and the outcomes of the actions we take.

For instance, Reinforcement Learning has been remarkably successful in combination with game theory. The application of Reinforcement Learning techniques to gaming has resulted in intelligent agents that can outperform the most talented humans [37][38][39][40]. However, conventional methods require a very large amount of practice to learn (e.g. the TD-Gammon backgammon player [38] takes several million games).

The framework of this project is set around the Reinforcement Learning Competition, an annual gathering of experts and students of Reinforcement Learning that compete in a variety of problem domains. The problem domains selected for the Competition usually provide an important and challenging testbed for learning algorithms, and the Competition itself helps researchers around the globe compare and understand in more detail how their algorithms perform in different problems.

Of the three domains proposed by the Competition we focus on the problem of **autonomous helicopter control**. This is a well-known problem in the field of Machine Learning that has been tackled by many researchers, and that has a high number of practical applications including rescue tasks, aerial filming, access to hazardous zones and others [36].

As mentioned above, one of the most restrictive aspects of Reinforcement Learning is the large number of interactions between the intelligent agent and its environment that are needed to distil a successful controller. For the cases in which no prior knowledge is assumed (e.g. no expert advice and unknown dynamics), conventional methods learning from scratch are very slow learners. Although they can achieve impressive performance, this limitation renders conventional Reinforcement Learning out of many practical, real-world applications in which agent-system interactions are scarce or expensive, like robotics or systems control.

With this problem in mind, in this work we develop a data-efficient learning method that can provide a successful helicopter controller using a small amount of interaction with the system.

The helicopter problem has been part of the Competition for several years, and several groups have published their results on it [25][27][28]. This gives us an opportunity to contrast our results and know what to expect of this Competition. However, we differ in method from many of the previous attempts to solve the helicopter problem within the Competition. We take a different approach and choose a model-based Reinforcement Learning algorithm with the specific goal of minimizing the experience needed by the agent to learn an effective controller.

As a result, we provide a method to train a helicopter controller to realize any aerobatic task that achieves good performance with a drastic reduction of the number of trials needed to train the controller. While previous attempts range between several thousand and several hundred thousand trials [27][30], the proposed method takes on average between 5 and 8 trials, which usually contain less that 1 minute of experience. Furthermore, we achieve better performance and more stable flight than any previous attempt to solve the generalized helicopter control problem without using prior knowledge.

2 Background

Reinforcement Learning has its roots on the work of optimal control in the late 1950s, with the appearance of dynamic programming [14]. Although different in their assumptions, the goal of both Reinforcement Learning and optimal control is to extract the maximum reward from a system, which is usually formulated as a Markov Decision Process (see sec. 2.1.1). This close relation moves some authors to draw strong connections between the two fields [1].

Later in the century, around the 1980s, the modern concept of Reinforcement Learning emerged when dynamic control was applied in combination with trial-and-error learning methods, to learn controllers for systems for which little or no information is available. This view of Reinforcement Learning, which has now become broadly studied, is the one in which we focus in this study.

In this section we describe the fundamentals of conventional Reinforcement Learning and the elements that describe a Reinforcement Learning problem (sec. 2.1). Next, we apply the introduced Reinforcement Learning techniques to a toy model example, the *gridworld* problem (sec. 2.1.7), to illustrate one the problems of typical Reinforcement Learning methods. Finally, we describe the aims and objectives of the 2014 Reinforcement Learning Competition, together with the proposed problem domains (sec. 2.2) and the software support they provide to run Reinforcement Learning experiments, RL-Glue [10] (sec. 2.3).

2.1 The problem

Reinforcement Learning (RL) is an area of Machine Learning inspired by behaviourist psychology, in which a learning *agent* interacts with an *environment* through certain *actions* that may modify the *state* of both the agent and the environment. The agent must be able to retrieve some information about the state through the *state signal* (or *observation*) in each time step. The goal of a RL agent is to perform a certain *task*.

For instance, consider a robot in a square grid that must travel from the start to the end of an unknown maze. In this simple example (often referred to as the *gridworld* problem), the task usually is to successfully exit the maze, the actions might be moving one step in different directions and the state could be the position of the robot in the grid.

Two of the main differences between the RL problem [1] and other machine learning problems are the concepts of *reward* and *decision making*. At each time step, after performing an action a in state s, the agent will receive a scalar numerical reward r. The goal of the agent is to maximize the reward received during the task by making the right decisions. Usually the action of the agent at a certain time affects the states the agent finds thereafter, such that the decisions made by the agent have a mid- or long-term effect. In this case we talk about a *sequential decision making* problem. In the previous example, exiting the maze could have a large positive reward whereas stepping into a trap could have a negative reward, and decisions made by the robot at a crossroads might affect which is the right path to take thereafter.

The most basic schematic of an RL problem is shown in figure 1. At each time step, the agent causes an effect on the environment by taking an action, and then it measures its new state and the received reward. Sometimes it is more convenient to study a system in terms of *state-action* pairs – that is, pairs formed by an action a and the state s in which it is taken.

In RL, unlike other kinds of machine learning, the agent is never told explicitly what to do. For instance, in supervised learning the learning agent has a collection of labelled examples provided by an external source of information. The supervised learning agent can then compare these examples with its own predictions, and adjust itself in consequence. However, in RL the agent

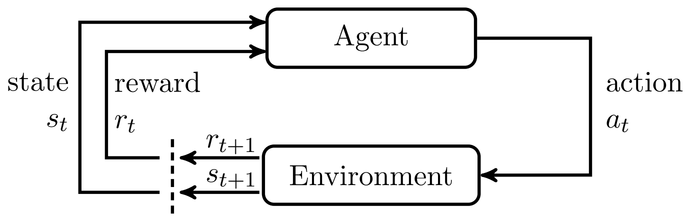


Figure 1: A simple diagram of the RL problem. An agent interacts with its environment by taking actions, that may give the agent a reward and change its state. The goal of the agent is to obtain the maximum reward possible. Extracted from [1].

has no other source of information other than its own experience and the reward as a learning signal, and must perform a trial-and-error search to maximise the total reward obtained.

We call *reward function* the function that maps states or state-action pairs to rewards. As shown in figure 1, r_{t+1} is the reward that follows the state-action pair (s_t, a_t). The reward function is an indicator of the immediate desirability of a certain state (or state-action pair), what makes the agent give it preference over other less desirable states. The relationship between the reward function, the actions of the agent and the desirability of the states are explored in section 2.1.3.

2.1.1 The Markov property

In general all parts of the RL problem can be stochastic. The reward or the new state following an action might have some random component, or might be influenced by another process not observed by the agent.

In the most general case, the probability distribution for the new state-action pair (s_{t+1}, a_{t+1}) can depend on the whole history $\mathcal{H}_t$ at time t of the trajectory followed by the agent, i.e.

$$P(s_{t+1} = s', a_{t+1} = a' | \mathcal{H}_t) = P(s_{t+1} = s', a_{t+1} = a' | s_t, a_t, r_t, s_{t-1}, ..., s_0, a_0) \ .$$

A key concept for RL is that of a Markov Decision Process (MDP). We say that a problem is Markov if the random variables involved obey the *Markov property*, and, thus, the expression above can be replaced by

$$P(s_{t+1} = s', a_{t+1} = a' | \mathcal{H}_t) = P(s_{t+1} = s', a_{t+1} = a' | s_t, a_t) \ . \tag{1}$$

That is, events at time $t+1$ only depend on events at time t. We should clarify that in reality s_t can depend on other information that does not necessarily come from time step t. For example, we would expect the dynamics of a physical system to be highly dependent on its velocity, which usually depends on the position of the system at t and at $t-1$. However, as long as this information is contained in the state signal perceived by the agent at step t the system is still perfectly Markovian.

In the context of the MDP we introduce the quantities $\mathcal{P}^a_{ss'}$, the state transition probability, and $\mathcal{R}^a_{ss'}$, the expected reward

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a), \tag{2a}$$
$$\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} = r | s_t = s, a_t = a, s_{t+1} = s'], \tag{2b}$$

respectively. $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ contain an almost complete description of the MDP. The only missing piece of information is the specific distribution $P(r_{t+1} = r | s_t = s, a_t = a, s_{t+1} = s')$ — the distribution of which $\mathcal{R}^a_{ss'}$ is the expected value.

2.1.2 Continuing and episodic learning tasks

Of the many different classifications one could think of for the set of RL problems, we will first introduce the distinction between *episodic* and *continuing* tasks.

An episodic task has a set of *terminal* or *absorbing* states, that conclude the episode. After a terminal state the agent can be sent back to the starting state to start a new episode. Exiting a maze or wining (or losing) a chess game are examples of terminal states.

On the other hand, a continuing task does not break naturally into episodes, but keeps going indefinitely. For example, controlling a system with a long life span can be considered a continuing task. Although there usually is the possibility of resetting the system and going back to a starting state, they are not the goal in the normal operation of the system.

On the basis of these two kinds of problems we define the concept of *expected return*, or return for short. The return at time t is the sum of all the rewards obtained from t on, weighted by a *discount rate* γ, which usually satisfies $0 \leq \gamma < 1$. The return is defined as

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{3}$$

In order for the expected return to be finite, we need to meet certain convergence conditions. First, the r_i sequence must be bounded. If the task is episodic, we know the sum will terminate at some point T when the agent reaches a terminal state, and thus the return[1] $R_t = r_t + r_{t+1} + ... + r_T$ is just the plain sum of a finite number of rewards and is perfectly convergent. However, if the task is continuing, the discount rate must be strictly less than one to ensure convergence.

For simplicity, we assume that in an episodic task the sum (3) is truncated at time T, or that all the rewards r_{T+k} following any terminal state are null. This convention will allow us to simplify the mathematical formulation of the problem and writing the summation up to infinity in all cases.

2.1.3 Policies and value functions

Another important element of an RL set-up is the agent's *policy.* The policy is the protocol the agent follows to select action a when measuring state s. The policy is usually denoted by the letter π. To make the dependence on s and a explicit it is sometimes written as $\pi(s, a)$. The policy can be a deterministic rule, a look-up table, or even stochastic.

After briefly mentioning the concepts of reward, return and policy, we are in position to introduce another fundamental concept of RL: the *value function.* The value function can be seen as an extension to the reward function: Whereas the reward function determines the immediate reward, the value function determines the long-term desirability of a certain state s. We usually denote the value function as $V(s)$.

Clearly, long-time desirability is a weak definition. A more robust definition would be to estimate $V(s)$ as the expected return of the agent when in state s (see equation (3)). However, the return in general depends on the trajectory and the decisions made by the agent after visiting state s, and the decision-making protocol of the agent is summarized in the policy. Thus, the value function and the policy are closely related and provide a solid definition of the value function,

[1]Some authors use the term return for the $\gamma = 1$ case only, where $R_t = r_t + r_{t+1} + ... + r_T$, and use the term discounted return for the $0 < \gamma < 1$ case.

i.e.

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t | s_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s\right] \ . \tag{4}$$

$V^{\pi}(s)$ now represents the expected return of being in state s and following the policy π. We call it the *state-value function for policy* π.

Similarly, we can be slightly more specific and define the *action-value function* $Q^{\pi}(s, a)$ as the expected return starting from s, performing action a and following π thereafter:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s, a_t = a\right] \ . \tag{5}$$

2.1.4 Optimality and the Bellman equation

So far we have introduced a significant number of new concepts that belong to the RL jargon and have pointed out a few inter-relationships between them, but we have not drawn any conclusion or followed any argument. In this section, we discuss more formally the mathematical structure of the RL problem.

First we begin by exploring the definition of the value function. As mentioned above, $V(s)$ is the expected return of the agent starting in s. Unless s is a terminal state, the agent will follow to s' after performing action a in s and gaining a reward r in the process. In turn, $V(s')$ is the expected return in state s'. With a quick examination we might suspect a recursive relation between $V(s)$ and $V(s')$, with a contribution from r.

More formally, the relation is described in the following equations:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s\right] \\ &= \mathbb{E}_{\pi}\left[r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s\right] \\ &= \sum_a \pi(a,s) \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \Big| s_t = s'\right\}\right] \ , \end{aligned}$$

where the sum $\sum_a \pi(a, s)$ accounts for the case in which the policy is stochastic, such that $\pi(a, s)$ is the probability of selecting action a in s.

The last term in the equation above is exactly the definition of $V(s')$. Thus we arrive at the so-called *Bellman equation*,

$$V^{\pi}(s) = \sum_a \pi(a,s) \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')\right]. \tag{6}$$

This equation represents a major milestone in most methods developed to solve the RL problem. In general, the Bellman equation is central to many aspects of control theory and to the mathematical optimisation method known as dynamic programming [14] (see section 2.1.5).

At this point, we recall that the exclusive goal of our RL agent is to maximize the reward obtained over time, R_t, or equivalently $V(s)$. The Bellman equation has a unique solution V

for a determined policy π [1]. Thus, it is sensible to consider that a policy π' is better (in all senses) than π if $V^{\pi'}(s) \geq V^{\pi}(s)\ \forall s$ [2].

Following this argument we can think of a policy that is better than or equal to all the other policies. We call it the *optimal policy* and denote it by π^*. The unique solution of the Bellman equation for the optimal policy is the *optimal state-value function* $V^*(s) = \max_{\pi} V^{\pi}(s)$.

In close relation we define the *optimal action-value function* $Q^*(s,a)$ as

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right]. \tag{7}$$

The optimal state- and action-value functions satisfy the relation

$$V^*(s) = \max_{a} Q^*(s,a). \tag{8}$$

For a finite MDP one can always find V^*, and it is independent of the policy [1]. A fundamental property of V^* is that, since it already takes into account the future consequences of choosing any possible state, any policy that chooses the action that takes the agent to the neighbouring state with highest $V^*(s)$ is an optimal policy. That is, any greedy policy with respect to V^* is an optimal policy. Through V^* the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state.

In this section, we have introduced the concept of optimality and the Bellman equation. These are very fundamental concepts in RL, since the goal of the agent is to maximize its reward by approaching the optimal policy through learning. It is common in several RL algorithms to use the agent's experience to estimate terms in the Bellman equation or apply other techniques to solve it and calculate the optimal value function.

2.1.5 Methods to solve the RL problem

Now that we have defined the RL problem and determined our goal through the Bellman equation, we will describe the three general families of methods that are used to solve a RL task. For this section we assume that the states and the actions are discrete and there is a finite number of them. The case of continuous state- or action-spaces requires different methods and is briefly described in section 2.1.6.

The most complete way to solve a RL problem is through **Dynamic Programming** (DP) [14]. DP is a collection of algorithms that can be used to compute optimal policies, but which have a major drawback: a perfect model of the environment as a MDP (that is, $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$) is needed.

Provided we do know the true values of $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$, it is easy to evaluate the problem. Following the discussion on the Bellman equation in the previous section, given a policy π we can calculate its associated $V^{\pi}(s)$ by iterating on the Bellman equation[3]. This method, shown in equation (9), is known as the policy evaluation method,

$$V^{\pi}_{k+1}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^{\pi}_k(s')\right]. \tag{9}$$

[2]The policies are equivalent if the equality holds for all s and one is better than the other if the equality breaks for at least one state.

[3]Knowing $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ we have all the information needed to determine V^{π}, but in large problems often the amount of computational resources needed is impractical and one must resort to other kinds of methods.

In every iteration, the policy evaluation method considers every possible one-step transition from s to update $V_k(s)$, and does this for all s. In other words, it *backs up* the value of every state to produce the new estimator V_{k+1}. We say that DP methods are *bootstrapping* in the sense that they compute a series of estimators based on previous estimations.

The policy evaluation method can be easily modified to achieve optimal results. As mentioned in the previous section, a policy which is greedy with respect to V^* is always an optimal policy. It is straightforward to modify equation (9) so that the policy is always greedy with respect to the last estimator of V. This method is the value iteration method, and has guaranteed convergence to optimal behavior,

$$V_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi_k(s') \right]. \tag{10}$$

The next type of RL-solving methods are the **Monte Carlo** (MC) algorithms. MC methods are usually more appropriate for a RL application because they assume no prior knowledge of the environment.

In a standard MC algorithm, a certain policy π is to be evaluated. Then an episode is generated following π and $V^\pi(s)$ is estimated as the sum of all the rewards obtained after the first visit to s. This process is iterated until convergence criteria for $V^\pi(s)$ are satisfied. We say MC is a *non-bootstrapping* method because the estimators of $V^\pi(s)$ are all independent and do not rely on previous estimators.

The MC method we just described is guaranteed to converge to V^π for any policy, but if our first guess for the policy is poor, convergence can be slow and V^π can be far from optimal [1]. In order to make a more robust method, we can update the policy after each episode according to the value estimators from previous episodes. For example, we can choose a policy that assigns a probability of choosing action a proportional to its Boltzmann factor $e^{\beta Q(s,a)}$, where β acts as the inverse of the temperature. This way the policy approaches the optimal policy as $Q(s,a)$ approaches Q^*. This method for assigning probabilities is known as the *softmax* method and is depicted in algorithm 1 assuming an episodic task.

Algorithm 1: Softmax MC algorithm

```
Initialize $Q(s,a)$ arbitrarily
Initialize $\pi$ to a random policy
while $Q(s,a)$ has not converged do
    Generate episode following $\pi$
    for each $(s,a)$ in episode do
        Update $Q(s,a)$ with return following the first occurrence of $(s,a)$
    end
    Update policy as $\pi(s,a) = \frac{e^{\beta Q(s,a)}}{\sum_{a'} e^{\beta Q(s,a')}}$ for all $s$
end
```

The last main type of methods are the **Temporal-Difference** (TD) methods [15]. TD is a combination between the ability of MC to learn without prior knowledge and the bootstrapping features of DP.

If policy π chooses action a in s_t obtaining a reward r_{t+1} and leading to s_{t+1}, then the exact solution of the Bellman equation will satisfy $V^\pi(s_t) = r_{t+1} + \gamma V^\pi(s_{t+1})$. However, if our estimator of V is not accurate there will be a non-zero error $\delta_t = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$. Then we can use this error to modify our estimator $V^\pi_{k+1}(s_t) = V^\pi_k(s_t) + \alpha\delta_t$.

A very effective TD method is the SARSA algorithm [16], which uses TD back-ups to estimate $Q(s,a)$. As in the previous case, plain SARSA can be enhanced with any policy improvement method. For instance, we can use an *ε-greedy* method, which chooses the greedy action with probability $1-\varepsilon$ and a random action with probability ε. Thus, since Q is continuously updating, π is also improving. This method is illustrated in algorithm 2, again for an episodic task.

Algorithm 2: ε-greedy SARSA algorithm

```
Initialize $Q(s,a)$ arbitrarily
Initialize $\pi$ to a random policy
while $Q(s,a)$ has not converged do
    Initialize $s$
    while $s$ is not terminal do
        Evaluate greedy policy with respect to $Q$ and select action $a$
        Take $a$ and observe $s'$, $r$
        Choose $a'$ to perform in $s'$
        $Q(s,a) = Q(s,a) + \alpha\,[r + \gamma Q(s',a') - Q(s,a)]$
        $(s,a) = (s',a')$
    end
end
```

This classification in 3 main types of algorithms is not completely solid. It is easy to develop algorithms that incorporate elements from the 3 types of algorithms and that combine different policy improvement methods to achieve optimal behaviour as fast and effectively as possible.

2.1.6 Learning in continuous spaces

So far we have been dealing with a finite set of states (actions), which the agent can measure (take). However, there are many applications and possible tasks that cannot be formulated in terms of finite sets, but instead take values in a continuous state or action-space. In the previous sections the action- and value-functions had a determined value for each state, and since the number of states was finite these could be stored as a table containing one value of V for each s. However, if the state signal includes real variables (or more generally, *features*) this is not a possibility.

We can generalize most of our previous discussion by representing the value function V not as a table but as a parametrized functional form with parameter vector $\vec{\theta}$. For example, $\vec{\theta}$ can be the weights of a neural network or the coefficients in a set of splines. Any change in $\vec{\theta}$ can affect the value estimation of many states, so the back-up process is not so trivial.

Fortunately, the literature on function approximation is extensive. We can overcome this problem by considering every step in the agent's trajectory as an estimator for the value function at that point. That is, in a discrete state space, every step we (usually) update the value of $V(s_t)$ and just replace it with a new estimator, say $r_{t+1} + \gamma V(s_{t+1})$. In a continuous space we can take the point $(s_{t+1}, r_{t+1} + \gamma V(s_{t+1}))$ as a conventional training example and use it together with any supervised learning method to learn the parameters $\vec{\theta}$ in $V^{\pi}(s;\vec{\theta})$.

In comparison, however, the problem posed by a continuous action-space is more serious. One of the possible solutions is to learn the action-value function of the problem, $Q(s,a)$. If the state- and action- spaces have dimensions $\mathcal{D}_s$ and $\mathcal{D}_a$ respectively, we can model Q as a function of $\mathcal{D}_s + \mathcal{D}_a$ variables with parameters $\vec{\theta}_Q$. In each step we can use a supervised learning method to estimate the value of $\vec{\theta}_Q$ that estimates a better Q and then use an optimisation method to calculate the optimal a given the current estimator of Q and subject to the constraint of being in the state s. However, this method involves an optimisation routine for every step, and is certainly inefficient. More sophisticated methods are known [13], but are out of the scope of this introduction.

Another approach to the continuous action-space problem is to ignore the value function and use a *direct policy search* method [6]. Direct policy search methods involve evaluating a policy by the return (or reward) it provides and updating it every episode (or step) to achieve a higher performance.

2.1.7 An example: the gridworld problem

The gridworld problem setting we consider consists of a 2-dimensional grid whose sites can be empty, trap, or obstructed. The state of the system is uniquely determined by the position of the agent in the grid, which the agent knows with absolute certainty. The goal of the agent is to move from a start position to a goal position.

Arriving to the goal state has a positive reward of 10 and finishes the episode. Moving through a trap state has a -6 reward. The agent can move freely through the empty sites with a reward of -1 each step, thus encouraging the agent to reach the goal state as quickly as possible. The obstructed states end the episode without any additional reward.

The gridworld problem can be completely formulated as an MDP. The state space is discrete and finite, with the number of states equal to the number of tiles in the grid, plus a fictitious terminal state. The only possible actions are going up, down, left or right. At every time step, the new state of the system only depends on the previous state and the action taken by the agent[4].

In order to compare the rest of the algorithms it will be useful to know the optimal solution to the problem. Since the model is completely known, we can easily work out the solution with a DP value iteration algorithm as outlined in section 2.1.5. The requirement of a model makes DP algorithms not suitable for the RL-C, but make a perfect benchmark for a completely determined problem like the gridworld.

The approximate (but almost exact) solution of the value function optimality equation for this grid is shown in figure 2. The greyscale map represents the value function $V^*(s)$ for each site. White represents the highest value and black the lowest. Red forms indicate the type of state (start, goal, trap or obstructed).

It can be easily computed that the optimal policy will achieve a total return $R = -4$. With this knowledge we can compare other algorithms in the same grid. For this example we evaluate the two algorithms outlined in section 2.1.5, SARSA and MC.

Results are shown in figure 3. All the areas drawn are 95% confidence intervals for the total reward received in each episode, averaged over 200 runs. The blue area corresponds to the ε-greedy SARSA algorithm (described in algorithm 2) using $\varepsilon = 0.1$. That is, the policy is updated each step to be greedy with respect to Q but selecting a random option with probability 0.1.

The green and red areas were computed with the same algorithm, a softmax MC method (as shown in algorithm 1). The difference between the two is the start condition. The red area was computed with an *optimistic start* — the initial values for Q were higher than the actual values obtained by the system, encouraging the *exploration* of new, unvisited states. Conversely, the

[4]Actually, two different actions in two different states are equivalent if they lead to the same state — the same site in the grid. This is known as an *afterstate* formulation, in which the state- and action-values are estimated the same for every (s, a) pair sharing s'.

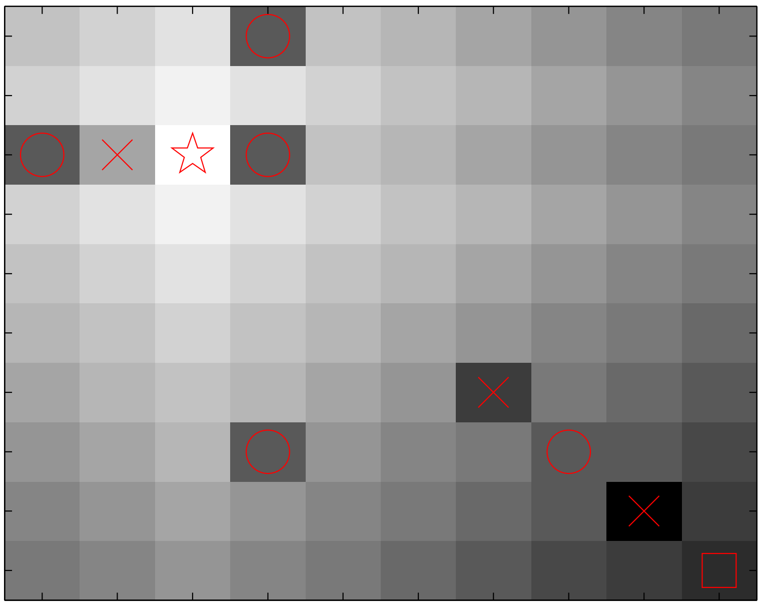


Figure 2: Approximate solution of the Bellman optimality equation for the gridworld considered. The red square is the start, the star is the goal, the circles are the obstructed and the crosses are the trap states.

blue area had a *pessimistic start*, in which the initial value for Q was much less than the real values, what forces the algorithm to find a way to the goal state and stick to that path thereafter, encouraging *exploitation* — the use of a known strategy to obtain a reward without exploring. However, the first choice of path in the pessimistic case is highly random, what explains the wide confidence level.

Despite the simplicity of this example we can already see some general consequences of the features of a RL algorithm.

- The first thing to notice is that an algorithm that overrates exploitation over exploration is likely to get a higher result in the short term, but a poorer result in the long term. In the case of this problem, the exploitative agent is choosing a path to the goal state which is safe (in the sense that it is already known), but which is far from optimal because no better paths have been discovered due to the lack of exploration.
- We can also see that in this problem softmax MC learns faster than ε-greedy SARSA, specially due to the policy improvement method. The ε-greedy method usually selects the greedy action and the rest of the times it has equal probability of selecting the best or the worst action, whereas the softmax method weights all the actions and assigns probabilities accordingly.
- Last, we note that the result obtained by SARSA is slightly lower than the result obtained by the optimistic MC. That is because the ε in the ε-greedy remained constant, whereas the temperature in the Boltzmann factor for the softmax method (recall section 2.1.5) slowly decreased during learning. For an infinite temperature all the actions are equally likely, and for low temperature the policy is always greedy. The incorporation of an *annealing schedule* that lowers the temperature as time passes allowed the agent to combine an exploratory policy at first and converge to an exploitative policy later on to obtain a maximum return.

Note that the best considered method so far, the optimistic softmax MC, took more than 100

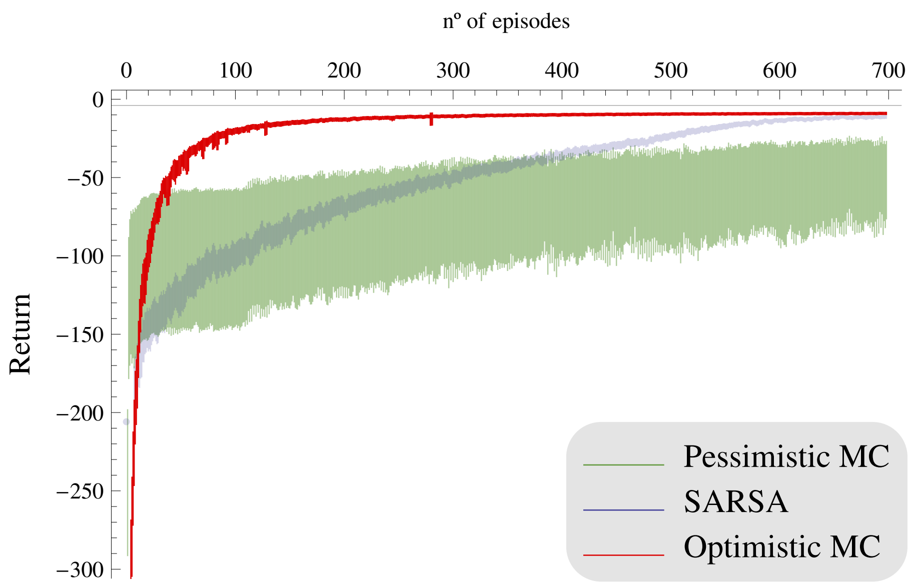


Figure 3: Performance of the 3 algorithms discussed: ε-greedy SARSA in blue, pessimistic softmax MC in green and optimistic softmax MC in red. Total return is shown in the vertical axis. An episode starts when the agent is placed in the initial state and ends when it arrives at the goal. The grey horizontal line at $R = -4$ is the optimal solution as determined by value iteration. In each case, the shadowed area is the 95% confidence belt for expected return.

episodes to perform partially well on the simple 10x10 grid of figure 2. This illustrates the problem of RL mentioned in section 1 — conventional RL algorithms need a high number of agent-environment interactions to learn an effective policy. While this is not a problem in toy models like the gridworld problem, it is very restrictive for real-world applications or in those applications in which simulation of the system is prohibitively expensive.

In these value learning methods, the data samples obtained through agent-system interaction are used once to update the value function, and then are discarded. To overcome this problem one can use the same data multiple times [18], train a supervised learning predictor that can act as a model of the environment, or do both things at the same time. Indeed, in section 3.1 we show an algorithm that incorporates these two features to extract as much information as possible from the data, and thus achieve unprecedented results in data-efficient RL [2].

2.2 The competition

This project is framed in the context of the 2014 Reinforcement Learning Competition (RL-C) held by the RL Community. The RL-C is aimed at RL students and researchers, and gives them the opportunity to test their algorithms in well-defined problem settings, as well as to create new specifically designed algorithms.

All the documentation about this and past RL-Cs can be found in the website

`http://www.rl-competition.org/` .

The competition is divided into 3 problems domains. In this section, we give a brief description of each domain and the challenges that they represent in the context of this project. Complete documentation can be found in RL-Cs website.

2.2.1 Helicopter

The Helicopter domain is based on the work of A. Ng's group at Stanford University [20][21][22][23]. The goal of the agent is to control a simulated helicopter and perform a certain task without crashing it. The task can be hovering the helicopter, flying at a constant stable speed or performing other more sophisticated aerobatics like flips and rolls. The simulator is based in a XCell Tempest helicopter, the same model used by the group at Stanford University and shown in figure 4.

Figure 4: XCell Tempest helicopter used by the Stanford University group [20] and simulated in the Helicopter domain in the RL-C 2014.

The observation or state space for this problem has 12 continuous variables, corresponding to the X, Y, Z components of the helicopter's velocity, position, angular rate and orientation. The action space has 4 continuous variables: longitudinal and latitudinal cyclic pitch and main and tail rotor collective pitch.

The goal of the Helicopter problem is to be able to safely control the helicopter. A large penalty is given if the helicopter moves too far from equilibrium (crashes), which should be avoided at all times. The task is run for 6000 steps, which simulates 10min of real flight. The simulator provided by the Competition implements 10 different tasks of unknown content, that are identified by a (0-9) integer.

The main challenge of the Helicopter domain is its relatively high-dimensional continuous state- and action-space and its noisy non-linear dynamics. Although we have all the physical information needed to characterise a 3D rigid body like the helicopter, the noise in the observation and external effects like the wind might make this problem hard to model as a MDP.

2.2.2 Invasive species

The Invasive Species domain is a biologically inspired problem set-up in which the goal is to find the optimal decisions to control a spatially spreading process. In this case, an invasive species (namely the Tamarisk tree) is competing against a native species in the ecosystem of a river network [24].

The environment has a binary tree network structure simulating a river network. Each of the E edges (or *reaches*) of the network has H slots (or *habitats*) that can be empty, occupied by native plants or by Tamarisk plants. That makes a total of 3^{EH} discrete states.

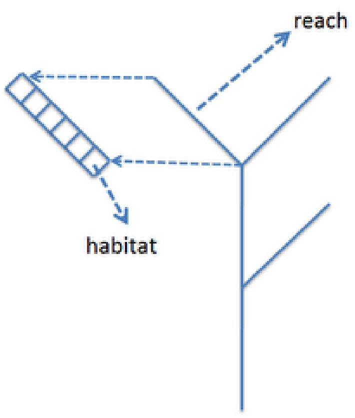


Figure 5: Schematic of the Invasive Species problem. Extracted from the Competition's website, accessed July 2014.

At each time step, a number of phenomena might occur. A tree might die spontaneously, and alive trees might spread their seeds and compete to occupy an empty habitat. Propagation is much more likely downstream than upstream.

There are four actions that can be taken in each reach: doing nothing, eradicating Tamarisk plants, restore native plants or eradicating Tamarisk and restoring natives. That makes a 4^E discrete action space for the whole system. Each action has a deterministic cost associated with it, but the output is always stochastic.

The goal of the problem is to reduce the spread of the Tamarisk tree while keeping the cost of the actions carried out as low as possible.

This a continuing discounted task, so we should aim for an on-line learning algorithm. The large size of both the state and action spaces imposes certain computational restrictions on the agents. In the default case $E = 7$, $H = 4$, there are more than 2×10^{13} states, so the computation time might be a significant factor in developing an algorithm.

2.2.3 Polyathlon

The Polyathlon domain is designed to be the most generic RL set-up. The problem is divided in an unknown number of unknown tasks, of which the only information available is that the tasks are episodic, approximately Markov and stochastic. The state space has 6 unknown continuous variables and there are 6 discrete actions available.

With such a generic description, the algorithm for a successful agent should be able to quickly adapt to any kind of task posed. The small amount of information about this domain is certainly the major obstacle (as well as its main feature).

2.3 The software: RL-Glue

To solve these domains we will need a robust software support. All three of the RL-C domains are built on the **RL-Glue** software package [10].

RL-Glue is a language- and platform-independent protocol for evaluating reinforcement learning agents with environment programs. RL-Glue separates the agent and environment-development process so that each can be written in different languages and even executed over the Internet from different computers.

The RL-Glue architecture is divided in the RL-Glue Core and 3 separated programs: the experiment, the agent and the environment. All of them have functions which interact with the RL-Glue Core during the execution.

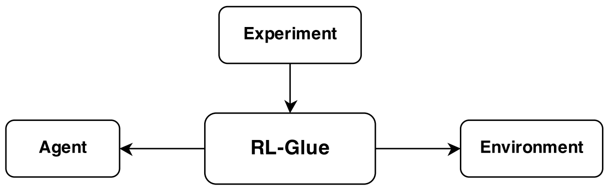


Figure 6: High-level diagram of the RL-Glue architecture [11]. The experiment calls RL-Glue Core functions, and the Core calls the functions provided by the agent and the environment.

The following lines are the most simple, yet illustrative description of the agent, environment and experiment as used in RL-Glue. They are extracted from the RL-Glue Overview Manual [11].

- In RL-Glue, the agent is both the learning algorithm and the decision maker. The agent decides which action to take at every step.
- The environment is responsible for storing all the relevant details of the world, or problem of your experiment. The environment generates the observations/states/perceptions that are provided to the agent, and also determines the transition dynamics and rewards.
- The experiment is the intermediary which (through RL-Glue) controls all communication between the agent and environment. This structured separation is by design, division of the agent and environment both helps create modularized code and captures our intuitions about how much the agent and environment should “know” about each other.

More specifically, each program must implement certain methods to enable interaction with the Core. The necessary methods are illustrated in diagram 7.

Following this architecture, a sample RL-Glue experiment would proceed as depicted in algorithms 3 and 4. We should clarify that what we call methods are indeed just abstract methods, and do not refer to implementation details (like classes, class methods or procedures). The RL-Glue Core is a language-independent software that interacts with different languages through their specific codecs. The details of the codecs are not relevant for the purpose of this report.

Algorithm 3: A sample RL-Glue routine

```
RL_init()
RL_start()
i ← 0
MaxSteps ← 100
terminal ← false
while i < MaxSteps and not terminal do
    terminal, reward, observation, action ← RL_step()
    Collect information from current step
    Increment i
end
RL_cleanup()
```

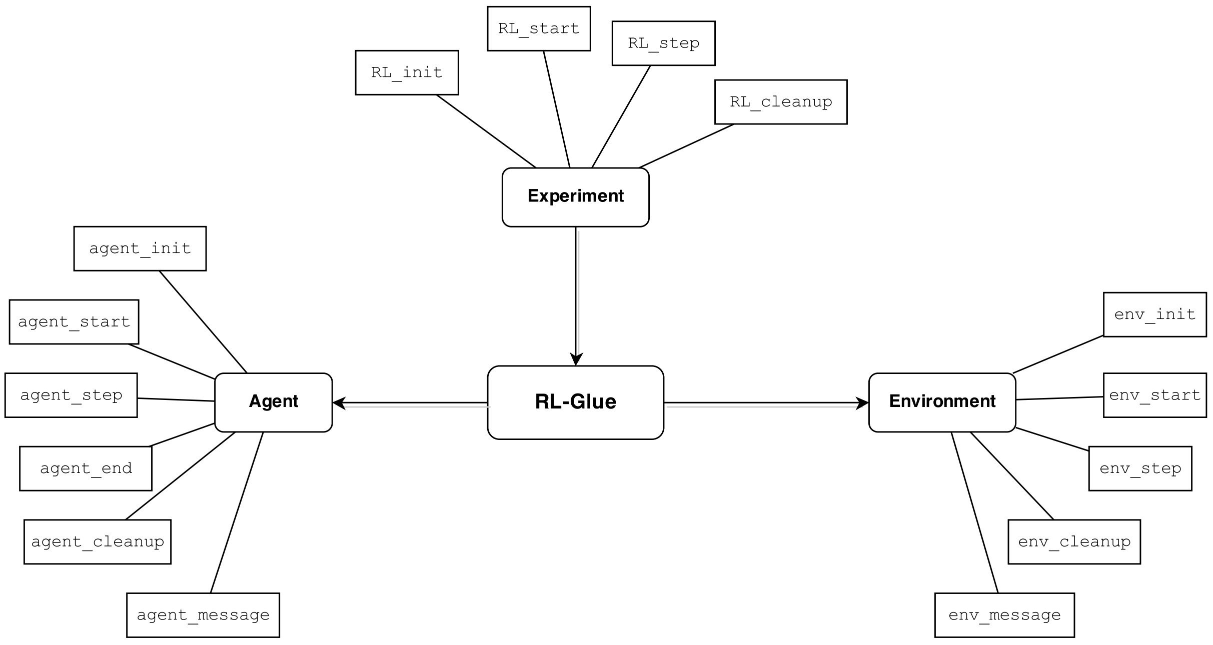


Figure 7: Schematic diagram of the full RL-Glue architecture [11]

Algorithm 4: Schematic of `RL_step()` function

```
Function RL_step():
    reward, observation, terminal ← env_step(action)
    if terminal then
        agent_end(reward)
        return reward, observation, terminal
    else
        action ← agent_step(reward, observation)
        return reward, observation, terminal, action
    end
end
```

3 Solving the helicopter domain

In the rest of this report we will describe the methods used to solve the helicopter problem as posed by the Competition, and the results and performance obtained.

To solve this problem we use a method fundamentally different from those described in section 2.1.5. To introduce it, we give a broader picture of the methods proposed to solve the RL problem.

Value learning: These methods explore the state-action space to gather information about the reward and estimate a value function, either V or Q. By transforming long-term desirability into a local property that only depends on s (or on s, a pairs), these methods can effectively ignore the dynamics of the system. All methods mentioned in section 2.1.5 fall within this category.

Direct policy search: As their name indicates, direct policy search methods do not attempt to build any estimate of the value function. Instead, they proceed by direct testing of the policy on the system. That is, given a class of policy functions, their performance can be assessed based on their expected return and thus the policy can be optimized to achieve good performance.

Model-based learning: All model learning methods are divided in two basic steps. First, they estimate the dynamical model of the system (often via a conventional supervised learning method) and then use it to learn a policy (i.e. an indirect policy search). The policy learning steps of these methods works in a similar fashion to a direct policy search, but is usually more effective[5], since more information about the original system is used in addition to the reward.

Our proposed method falls in the category of *model-based learning.* To the best of our knowledge, this is different from the common approach of previous winners of the Competition, that have opted for other value learning or direct policy search methods [25] [27] [28].

3.1 High-level steps

The main algorithm we use in this study is an adapted version of PILCO [2]. This is a model-based learning RL algorithm that relies on a *Gaussian Process* (GP) for the model learning step and on gradient-based optimization methods for the policy improvement step. The high-level steps are summarized in algorithm 5 and detailed in sections 3.2 and 3.3.

Algorithm 5: PILCO

```
Generate trajectories using random actions and collect dataset $\mathcal{D} = \{(s, a, r, s')\}$
repeat
    Train GP dynamics model from current dataset $\mathcal{D}$
    Learn policy via policy search
    Run latest policy and collect new dataset $\mathcal{D}_i = \{(s, a, r, s')\}$
    Aggregate datasets, $\mathcal{D} = \mathcal{D} \bigcup \mathcal{D}_i$
until task learned
```

As a model-based learning method, PILCO is exposed to *model errors* — the policy search algorithm assumes the dynamics model is a perfect model of the environment, which is usually not the case. This effect is particularly noticeable when there is a limited number of samples and multiple hypothesis are similarly likely.

To address this problem, PILCO takes advantage of the *probabilistic* outcomes of the GP. By considering the uncertainty in the estimations, the algorithm is less prone to model bias [32]. In the policy search step, the uncertainties are consistently propagated and taken into account during the optimization process.

In the following we outline the mechanisms behind the two main steps in PILCO, *Gaussian Process training* and *policy search.*

3.2 Gaussian Process regression

To perform the model learning step in algorithm 5, we use a state-of-the-art, data-efficient supervised learning algorithm based on a Gaussian Process (GP). In this section we will only outline the features of this method, for an extensive description of GPs and their applications we refer to [3].

As their name implies, GPs are based on the properties of Gaussian distributions. More specifically, it relies on the fact that *conditionals and marginals of a multivariate Gaussian distribution*

[5]More effective in terms of the number of agent-environment interactions needed.

are also Gaussian distributions. As an illustration, let $\mathbf{x}, \mathbf{y}$ be two random vectors that follow a joint Gaussian distribution, i.e.

$$p(\mathbf{x}, \mathbf{y}) = \mathcal{N}\left(\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\end{bmatrix}, \begin{bmatrix}A & C\\ C^\top & B\end{bmatrix}\right) , \tag{11}$$

where $\mathbf{a}, A$ are the mean and covariance matrix of $\mathbf{x}$, respectively; $\mathbf{b}, B$ are the mean and covariance matrix of $\mathbf{y}$, respectively; and C contains the cross-terms of the covariance.

To marginalize over a part of the variables in the Gaussian distribution, e.g. $\mathbf{y}$, we perform the integral

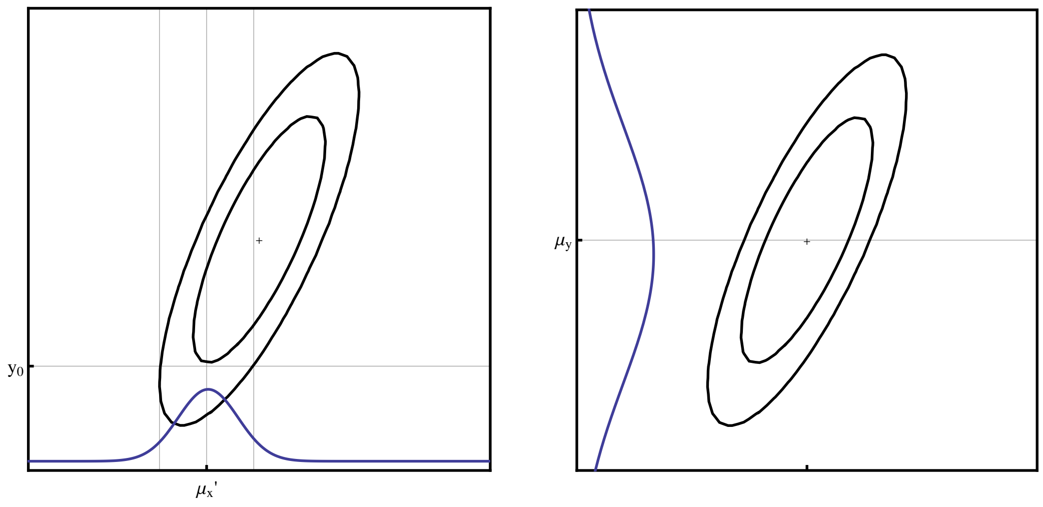
$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{y}) d\mathbf{y} = \mathcal{N}(\mathbf{a}, A) , \tag{12}$$

which yields the desired result that $\mathbf{x}$ also follows a Gaussian distribution.

Similarly, using this result and Bayes' rule one can compute the conditional probability

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{y})} = \mathcal{N}(\mathbf{a} + CB^{-1}(\mathbf{y} - \mathbf{b}), A - CB^{-1}C^\top) , \tag{13}$$

which again is a Gaussian distribution. For a proof of these properties, see any standard Multivariate Statistics book, e.g. [33].



(a) Conditioned Gaussian distribution **(b)** Marginalized Gaussian distribution

Figure 8: Schematic of conditioned and marginal distributions obtained from a two-dimensional multinormal distribution

Once reviewed the relevant properties of the Gaussian distribution, we proceed back to the description of the Gaussian Process. The formal definition of a GP [3] is as follows:

A Gaussian Process is a collection of infinitely many variables, any finite number of which have (consistent) Gaussian distributions.

Thus, a GP is the extension of a multivariate normal distribution to the infinite-dimensional case. To this we can add the idea that a function can be thought of as a vector with an infinite number of components. This way we can build the concept of the GP as a "distribution of functions", i.e. a probability distribution from which we can sample functions. For instance, in the one-dimensional ($f : \mathbb{R} \to \mathbb{R}$) case, we could say that

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')) . \tag{14}$$

As the common multivariate normal distributions, a GP is completely specified by a mean and a variance, or in this case, a *mean function* $m(x)$ and a *covariance function* $k(x, x')$. In the multidimensional case, where $f : \mathbb{R}^n \to \mathbb{R}$, the covariance function k is a function that maps two vectors of the n-dimensional space to a scalar, $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$.

Thus, having a covariance function k and two sets of vectors $\{\mathbf{x}_i\}, \{\mathbf{x}'_i\}$ we can build the matrix K, in which the element $K_{ij} = k(x_i, x'_j)$. If k is applied within one set of vectors, K is the Gram matrix of the set in the vector space with inner product $k(\cdot, \cdot)$. In this sense, k plays the role of a *kernel* [7] in the construction of the GP. The fundamental role of a kernel is to provide a notion of *distance* within the input-space (see introduction to section 4). For a discussion on kernels for Gaussian Processes and their performance see [8] and references therein.

So far we have described the GP as a mathematical tool, a generalization of the multinormal distribution, but we have not described how it can be used for supervised learning. At this point we recall the fundamental definition of the supervised learning problem: Given a set of labelled examples $\{X, \mathbf{y}\}$ and a new input $\mathbf{x}^*$, our goal is to estimate the probability distribution of its associated output y^*, i.e. $p(y^*|\mathbf{x}^*, X, \mathbf{y})$.

The good news is this prediction comes naturally from the structure of the GP. First, we must set a GP *prior*, for which we need a mean and a covariance function. For simplicity, we take $m_{prior} = 0$, and leave $k(\mathbf{x}, \mathbf{x}')$ unspecified. Then our prior is simply

$$f(\mathbf{x}) \sim \mathcal{GP}(0, k(\mathbf{x}, \mathbf{x}')) \; . \tag{15}$$

To incorporate our knowledge about the function (i.e. the data) we take every data point as a parameter in the GP, and assume each measured point f to be drawn from a normal distribution $\mathcal{N}(f, \sigma^2_{noise})$. With this information, we can calculate the GP *posterior* as

$$\begin{aligned} f(\mathbf{x})|X, \mathbf{y} \sim \mathcal{GP}\Big(& m_{post}(\mathbf{x}) = k(\mathbf{x}, X)[K + \sigma^2_{noise} I]^{-1}\mathbf{y}, \\ & k_{post}(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}, \mathbf{x}') - k(\mathbf{x}, X)[K + \sigma^2_{noise} I]^{-1} k(X, \mathbf{x}')\Big) \; , \end{aligned} \tag{16}$$

where $k(\mathbf{x}, X)$ is a vector whose i-th component is $k(\mathbf{x}, X_{i\cdot})$ (and similarly for $k(X, \mathbf{x}')$) and K is the Gram matrix of the set of input vectors X. This posterior encapsulates all the information we can obtain from the data. Being a GP, now we could sample a function $f(\mathbf{x})$ from the distribution inferred from the data, that we could then evaluate at any desired point $\mathbf{x}^*$.

However, this naïve procedure is unreliable, since we are just taking one random sample of the predicted distribution, and we would lose information of the variance of this estimate. Instead, given a new input $\mathbf{x}^*$ whose function y^* we want to predict, the optimal approach to estimate the probability distribution over y^* is to calculate the Gaussian predictive distribution

$$\begin{aligned} p(y^*|\mathbf{x}^*, X, \mathbf{y}) = \mathcal{N}\Big(& k(\mathbf{x}^*, X)[K + \sigma_{noise} I]^{-1}\mathbf{y} \;, \\ & k(\mathbf{x}^*, \mathbf{x}^*) + \sigma^2_{noise} - k(\mathbf{x}^*, X)^\top [K + \sigma_{noise} I]^{-1} k(X, \mathbf{x}^*)\Big) \; . \end{aligned} \tag{17}$$

Thus, by manipulating the GP posterior and the new input $\mathbf{x}^*$ we can estimate the mean and variance of the predicted distribution for y^*. Recall that to compute this posterior one needs to fully specify $k(x, x')$. The problem of selecting k or adjusting its parameters is what we refer to as *training* the GP.

To train the GP we need to find the kernel k that best describes the data. We consider a family of kernels $k(\mathbf{x}, \mathbf{x}'; \boldsymbol{\omega})$, with $\boldsymbol{\omega}$ being the *hyperparameters* of the covariance function. To find the optimal hyperparameter vector $\boldsymbol{\omega}$ we use a standard maximum marginal likelihood method

— given a dataset, we find the kernel $k(\mathbf{x}, \mathbf{x}'; \boldsymbol{\omega})$ that has the maximum probability of having generated the data, assuming the data actually follows a GP.

More specifically, we maximize the *marginal likelihood*, since the distribution over the function is marginalized over the latent function to obtain an expression for the marginal likelihood that depends only on the measured data points. As usual, for convenience we maximize the log-marginal-likelihood,

$$\log p(\mathbf{y}|X, \boldsymbol{\omega}) = -\frac{1}{2}\mathbf{y}^\top K^{-1}\mathbf{y} - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi \ , \tag{18}$$

that conveniently counts with an analytical expression for its gradient,

$$\frac{d}{d\omega_j}\log p(\mathbf{y}|X, \boldsymbol{\omega}) = \frac{1}{2}\mathbf{y}^\top K^{-1}\frac{dK}{d\omega_j}K^{-1}\mathbf{y} - \frac{1}{2}\mathrm{tr}\left(K^{-1}\frac{dK}{d\omega_j}\right) \ . \tag{19}$$

One of the advantages of the GP is that given the simple form of its probability density, both the log-likelihood and its gradient are analytically computable, what makes gradient-based methods a convenient and effective approach.

3.3 Policy learning

Once the GP regressor has been trained, it can be used to learn a policy to minimize some *cost function* $c(\mathbf{s})$. The cost function maps every state $\mathbf{s}$ to a scalar that represents its immediate loss or non-desirability[6].

Nonetheless, as described in section 2, the goal of the agent is not to minimize the immediate loss, but to minimize the total loss in a whole episode of the task. To take this consideration into account, the policy is trained to optimize the *expected total cost* (or *loss*)[7]

$$J^\pi(\boldsymbol{\theta}) = \sum_{t=0}^{T}\mathbb{E}_{\mathbf{s}_t}[c(\mathbf{s}_t)] \ , \tag{20}$$

where T is the length or *time horizon* of the episodic task we are considering, and $\mathbb{E}_{\mathbf{s}_t}$ represents an expected value over the distribution of states encountered at time t after following the policy π from the start of the episode. Note the analogy between the total cost J^π defined this way and the return R defined in equation (3) setting $\gamma = 1$. Since the goal of the problem is to control the helicopter for 6000 time steps, the task is effectively episodic, so given that the cost function is bounded then J^π is also bounded.

To perform the indirect policy search we restrict to a class of policy functions Π, that contains policies parametrized by a parameter vector $\boldsymbol{\theta}$, i.e. $\pi(\mathbf{s}; \boldsymbol{\theta})$. By using the policy, given a state distribution at a certain time $p(\mathbf{s}_t)$ we can calculate the mean and covariance of the joint distribution $p(\mathbf{s}_t, \mathbf{a}_t)$. Then, by approximating this joint distribution by a multivariate Gaussian with the correct mean and covariance we can use the trained GP to predict the distribution of the next state, i.e.

$$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = \mathcal{N}(\boldsymbol{\mu}_{t+1}, \Sigma_{t+1}) \ . \tag{21}$$

That is, the trained GP is in charge of estimating $\boldsymbol{\mu}_{t+1}, \Sigma_{t+1}$, the mean and covariance of the next state given the current state and action.

[6]For practical purposes, a negative reward.

[7]J^π is also named the cost-to-go of policy π.

We cascade T steps of this one-step prediction procedure, such that we can obtain estimates of $p(\mathbf{s}_t)$ for $t = 1, 2, ..., T$ provided we can also estimate the initial state distribution $\mathcal{N}(\mathbf{s}_0, \Sigma_0)$. Then, having an estimator of $p(\mathbf{s}_t)$ we can estimate the expected value of the cost at each time step $\mathbb{E}_{\mathbf{s}_t}[c(\mathbf{s}_t)]$ and thus calculate the expected total cost of a given policy J^{π}.

Note that to generate one single estimate of J^{π} we have required T applications of the GP regressor, what makes this an expensive operation. Without further information minimizing this cost would be a very expensive task.

The key is that, provided that the cost function has an analytical expression for its gradient, *it is possible to compute analytic gradients of the total expected cost.* The way these are calculated is by repeated application of the chain rule on the expression of J^{π}. Let $\mathcal{E}_t = \mathbb{E}_{\mathbf{s}_t}[c(\mathbf{s}_t)]$, then we can write

$$\frac{dJ^{\pi}}{d\theta_j} = \sum_{t=0}^{T} \frac{d\mathcal{E}_t}{d\theta_j} = \sum_{t=0}^{T} \left[\frac{\partial \mathcal{E}_t}{\partial \mu_t} \frac{d\mu_t}{d\theta_j} + \frac{\partial \mathcal{E}_t}{\partial \Sigma_t} \frac{d\Sigma_t}{d\theta_j} \right] , \tag{22}$$

and by successive application of the chain-rule and basic properties of Gaussian distributions we can calculate $\frac{d\mu_t}{d\theta_j}$, $\frac{d\Sigma_t}{d\theta_j}$ and obtain an analytic expression for $\vec{\nabla}_{\boldsymbol{\theta}} J^{\pi}$. Having this gradient it is convenient to use a gradient-based optimizer, as we did with the training of the GP. Nonetheless, this is a non-convex function, and as such we must use a non-convex optimization method, like CG or BFGS [35].

The details of the gradient derivations and the optimization algorithm are complex and out of the scope of this report. For a rigorous derivation we refer to [2], [5] and [4].

3.4 Visualization

When diagnosing problems in the learning algorithm it is of crucial importance to observe the behaviour of the system. However, for a relatively high-dimensional system like the helicopter raw data can be difficult to interpret. We recall that the helicopter state is described by 12 variables,

1. Linear velocity $\{v_x, v_y, v_z\}$,
2. Position relative to origin $\{x, y, z\}$,
3. Angular velocity $\{\omega_x, \omega_y, \omega_z\}$, and
4. Orientation, expressed as a quaternion $\{q_x, q_y, q_z\}$;

and has 4 continuous action variables that represent the controls of the pilot,

1. Longitudinal cyclic pitch a_1,
2. Latitudinal cyclic pitch a_2,
3. Main rotor collective pitch a_3, and
4. Tail rotor collective pitch a_4.

To make the interpretation of the data more intuitive we developed a visualization tool to represent the state of the helicopter, the control variables and the immediate cost. Figure 9 shows a snapshot of the visualization tool.

The helicopter representation is built with a superposition of a few simple geometric figures. This simple representation, beyond its delightful artistic content, allows a quick visual recognition of the orientation state of the helicopter. Additionally, a set of {X, Y, Z} coloured axes is drawn inside the helicopter to identify the helicopter body-fixed frame, in which all variables in the problem are expressed.

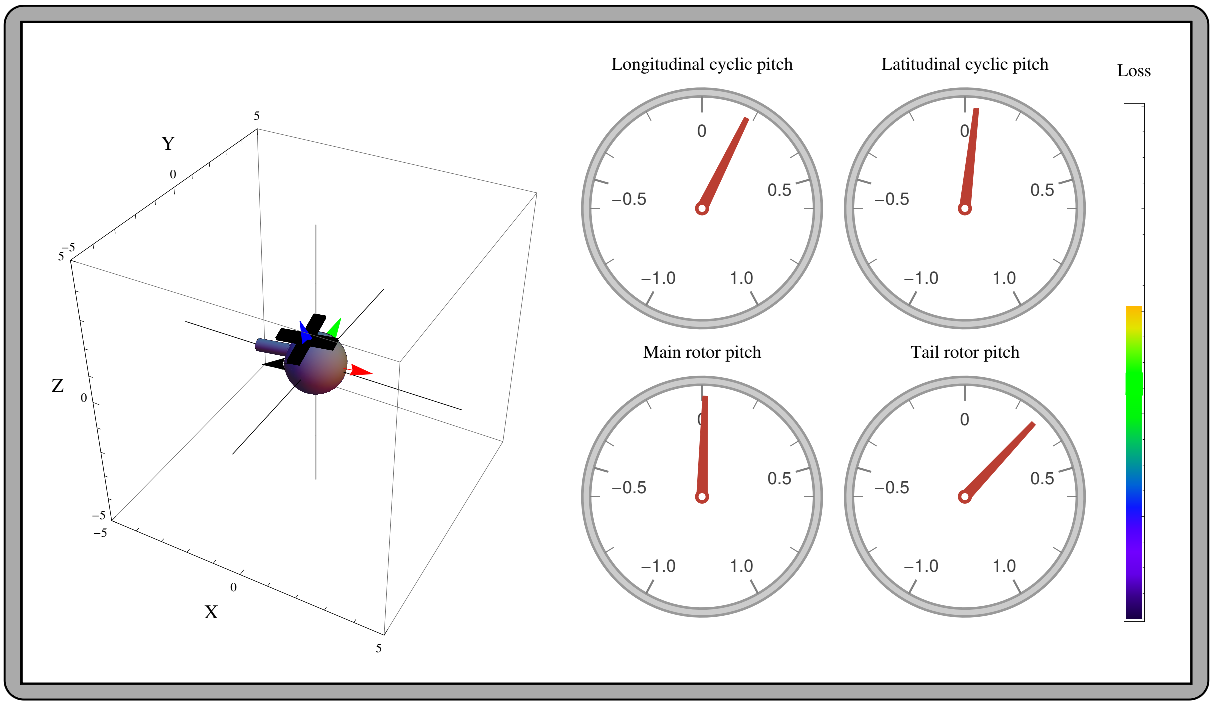


Figure 9: Visualization tool used to display the state of the helicopter, the control variables and the immediate loss. See text for details.

The three black lines shown intersect at the origin, and the black arrow indicates the direction and magnitude of the velocity of the helicopter. The four angular gauges depicted represent the state of the control variables. The rightmost element of the panel shows the immediate cost, calculated following the considerations in section 4.1.

4 Experimental results

As described above, GPs can be used to learn scalar functions $f : \mathbb{R}^n \to \mathbb{R}$. Otherwise, if we wanted to learn a vector field $f : \mathbb{R}^n \to \mathbb{R}^m$ we would need a vector mean and covariance functions, which would be more difficult to handle. Instead of extending the proposed GP to estimate the m-dimensional output distribution, we take the simpler approach of using one different GP to predict each of the 12 state variables in the helicopter domain.

As mentioned in section 3.2, to train a Gaussian Process regressor we must specify a hyperparameter-dependent covariance function. In this study we use the squared exponential kernel,

$$k(\mathbf{x}, \mathbf{x}') = \alpha^2 \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^\top M(\mathbf{x} - \mathbf{x}')\right) + \sigma_n^2 I \ , \tag{23}$$

where $M = \operatorname{diag}(\boldsymbol{\ell})^{-2}$ is a matrix whose diagonal contains the inverse square of the 16 *characteristic length-scales* $\boldsymbol{\ell}$ (12 state variables, 4 actions) of the Gaussian Process, α^2 is the *signal variance* and σ_n^2 is the *noise variance*.

The length-scale ℓ of a certain input is a measure of the distance we have to move along a certain dimension in the input space to see an important effect of the variable on the output. In this sense, the kernel introduces a notion of distance in the state-space through the length-scales. For instance, if the length-scale is much larger than the standard deviation of the input, we can infer that that variable does not play a significant role in the prediction. This provides a fast, simple way to estimate which variables are more informative in the prediction.

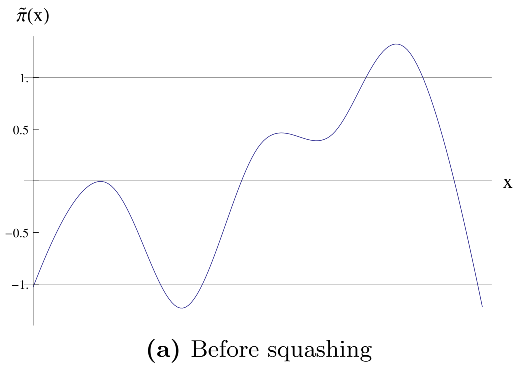
To specify the dynamics GP prior we also need a mean function $m(\mathbf{x})$. While $m(\mathbf{x}) = 0$ is a simple and often effective prior, we can obtain higher performance by considering a more suitable function. Specifically, being a physical system we expect the state of the helicopter to undergo small changes at each step. Therefore, we use the prior $m(\mathbf{x}) = \mathbf{x}$ that is simple and makes the GP more effective.

Additionally, we have to provide a differentiable cost function that represents the reward returned by the RL-Glue environment. To do this, we define a function that transforms rewards into costs and train a GP to predict costs based on the state of the helicopter.

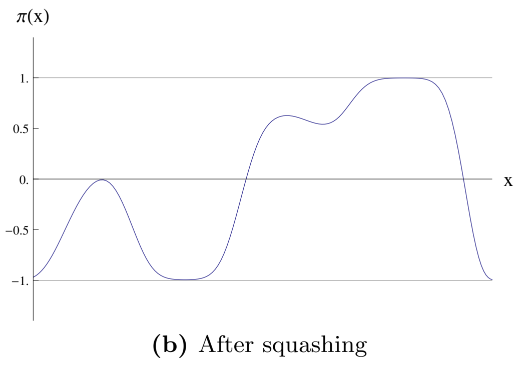
For the policy search step we consider the class of linear policies, $\tilde{\pi}(\mathbf{x}; A, \mathbf{b}) = A\mathbf{x} + \mathbf{b}$. However, these policies are not bounded. To make sure that the action variables are bounded in the range $[-1, 1]$, we introduce a *squashing function* $\sigma(x)$ to map the policy outputs to the desired range. In this case, the squashing function is the third order Fourier approximation of a trapezoidal wave, i.e.

$$\sigma(x) = \frac{9\sin(x) + \sin(3x)}{8} , \tag{24}$$

and its effect is illustrated in figure 10. Then, the final policy class we optimize in the policy search is $\pi(\mathbf{x}; A, \mathbf{b}) = \sigma(A\mathbf{x} + \mathbf{b})$.



(a) Before squashing



(b) After squashing

Figure 10: Action of the squashing function $\sigma(x)$ on the policy. Actions outside the $[-1, 1]$ range are capped, while actions inside the range are left undisturbed.

Adding up, the dynamical model GP has 18 free parameters for each of the 12 predicted dimensions, plus the 14 free parameters of the reward GP. At the same time, the class of linear policies has 52 free parameters. In short, we are facing a difficult RL problem with complex dynamics in a $\mathbb{R}^{12}$ state space and a $\mathbb{R}^4$ action space, and a proposed model with 282 free parameters.

Throughout this section we measure the performance of PILCO and the proposed modifications on the helicopter simulator provided by the Competition. The simulator implements 10 different MDPs, which correspond to 10 different learning tasks. The content of the tasks is unknown, and they are identified by a (0-9) integer.

4.1 Learning the reward function

As mentioned in section 3.3, PILCO's policy learning algorithm is based on the minimization of the cost function $c(\mathbf{s})$. However, the Competition's simulator provides reward instead of cost. Therefore, we must introduce a function $\tilde{C}(r)$ that transforms the reward returned by the simulator to a cost we can feed into PILCO's policy learning.

There are certain requirements that $\tilde{C}(r)$ must meet:

- Its image should be bounded.
- It should not create new maxima or minima.
- Its first derivative should be negative, such that by minimizing the cost PILCO will effectively maximize the reward.

Remember that we assume *no prior knowledge* of any parts of the problem, so we must use $\tilde{C}(r)$ to ensure that the cost has the desired bounds independently of the bounds of the reward function. The only information we use is that we know the reward is upper-bounded by 0 and it depends solely on the state of the helicopter, and not on the actions. This information can be found in the Competition's helicopter problem specifications [12].

After the previous considerations, the proposed transformation is

$$\tilde{C}(r) = 1 - \exp(-r/r_0) \ , \tag{25}$$

where r_0 is a reward scaling parameter. If the value of r_0 is small, the algorithm will be able to better discriminate between two good states with similar rewards, and opt for the best one. However, if r_0 is small and the algorithm is faced with two bad states it will not be able to tell which is worse. Similarly, the optimizer will face the opposite problems if r_0 is too large.

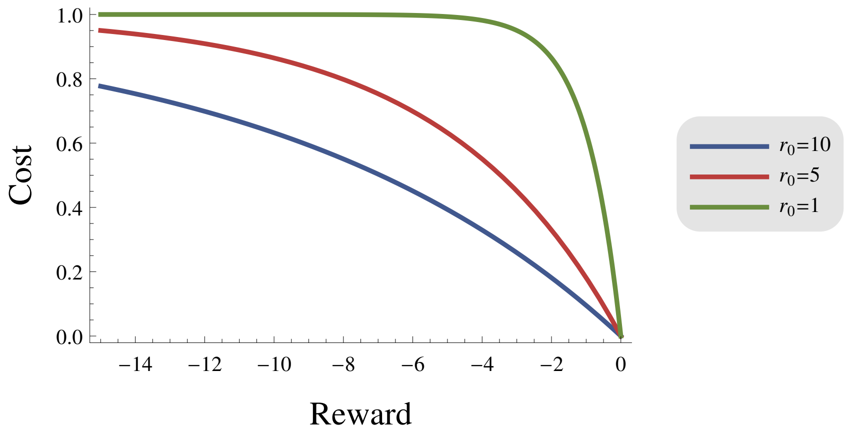
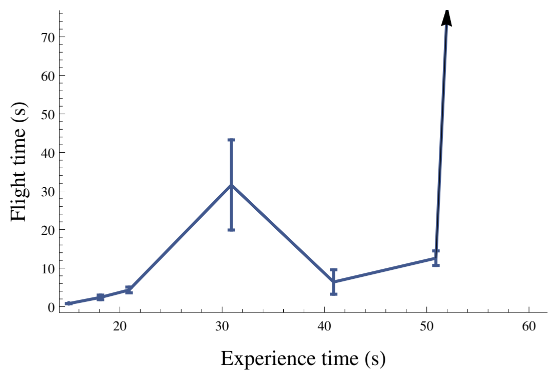


Figure 11: Reward-cost transformation function with different reward scaling parameters r_0. A larger scaling parameter allows the function to handle very bad states, but has less resolution in the $r \to 0^-$ limit than a smaller scaling parameter.

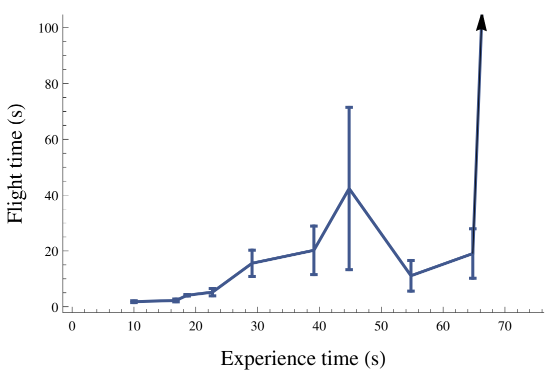
Given any dataset of trajectories, $\tilde{C}(r)$ is applied to the rewards along the trajectory and the resulting cost is used as input for a GP. With this step we ensure that we can interpolate the cost between unobserved states and that we can compute the gradients of the cost required by PILCO (see section 3.3). Then this GP is used as the cost function in the policy search step.

After trying more sophisticated options like a variable reward scaling or the use of several scales simultaneously, the simplest option proved to be the most effective. PILCO is relatively robust against the specific value of r_0, so a fixed, moderate value of the scaling parameter is able to lead to a successful policy. Other forms of $\tilde{C}(r)$ were tried, but the amount of hand-tuning was such that the algorithm was not easily generalizable could not be applied to other tasks.

The proposed algorithm was tested with this reward-cost transform on the Competition's helicopter simulator. Figure 12 shows the results of two sample runs of the algorithm using a fixed reward scaling parameter $r_0 = 10$ on two randomly selected tasks. The performance of a policy is judged by the mean and standard deviation of the flight time it achieves.



(a) Task 1



(b) Task 3

Figure 12: Performance of the policy trained by PILCO in each iteration after certain experience time, using a reward scaling parameter $r_0 = 10$. At the black arrow performance of the policy jumps to 600s and the problem is solved.

The first thing we note is that the algorithm has successfully learnt a controller policy that can survive the 10-minute limit set by the Competition, and it does this in a very small number of trials. Using around 1 minute of experience and 5–10 trials the algorithm is able to learn a controller that can perform several aerobatic trajectories. Note that the number of trials is a very important measure of the performance of the algorithm — it represents the number of spare helicopters we had to “crash” before learning the task, and thus poses a crucial restriction for practical applications.

However, the method proposed above is not completely successful, and does not learn a successful controller more than 50% of the times. There is a high probability that the optimizer finds a local minimum with very poor performance, with catastrophic consequences. This kind of events is illustrated in figure 13.

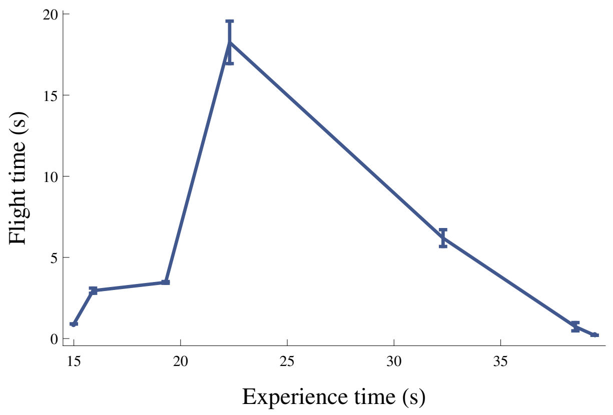


Figure 13: Sample unsuccessful run of the algorithm. Despite achieving a non-trivial 17s flight time, the algorithm fails and its performance drops.

In other words, this method is highly unstable. The fundamental question we have to address is: Why does the algorithm fall to such a poor minimum, even though it has started to move in the right direction?

The dynamics model GP is good enough to make predictions that support a 20s-long flight, and the policy search predicts low cost estimates. Thus, the only possibility is that the instability is in the reward model GP.

We can test this hypothesis by plotting the estimated cost predicted by the reward model GP. This is a multidimensional $c : \mathbb{R}^{12} \to \mathbb{R}$ function, so for simplicity we plot the results along one axis, namely the linear forward velocity v_x. The estimated cost, the true cost and the measured data density are displayed in figure 14.

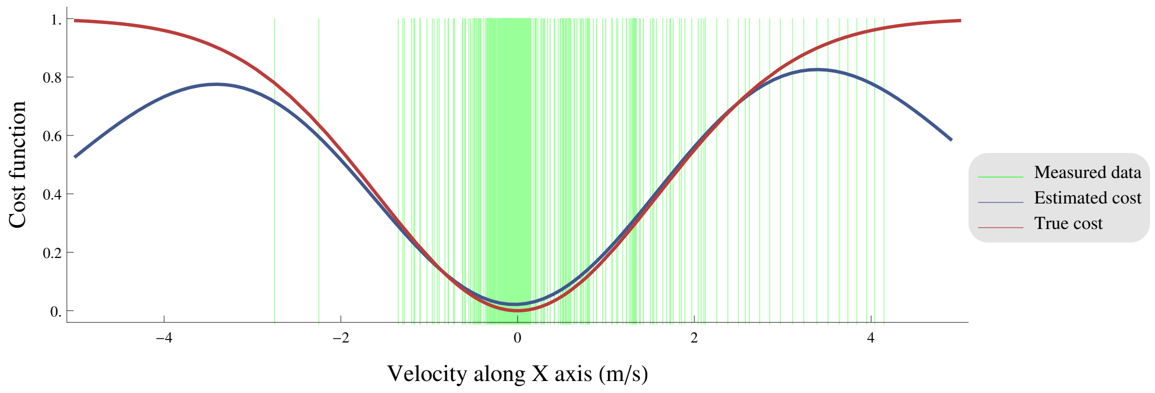


Figure 14: Learned and true cost function along the v_x axis. Measured data points are represented with a shaded green line. Predictions far from the region with high data density approach the GP prior mean, in this case $m_{prior}(\mathbf{s}) = 0$.

With this information, we can infer the origin of the problem. In unvisited regions of the state-space, where the GP has no information, the predictions tend to approach the GP prior mean, which in this case is 0. If the time horizon T, the number of steps predicted by the GP in the policy search (introduced in equation (20)) is large enough, the optimizer might be misled to think that it can achieve a lower cost in other regions of the state-space, resulting in a tragic end for the helicopter.

To address this problem we robustify our method by using a pessimistic and, therefore, more conservative agent. In that direction, we change our reward-cost transformation function and use

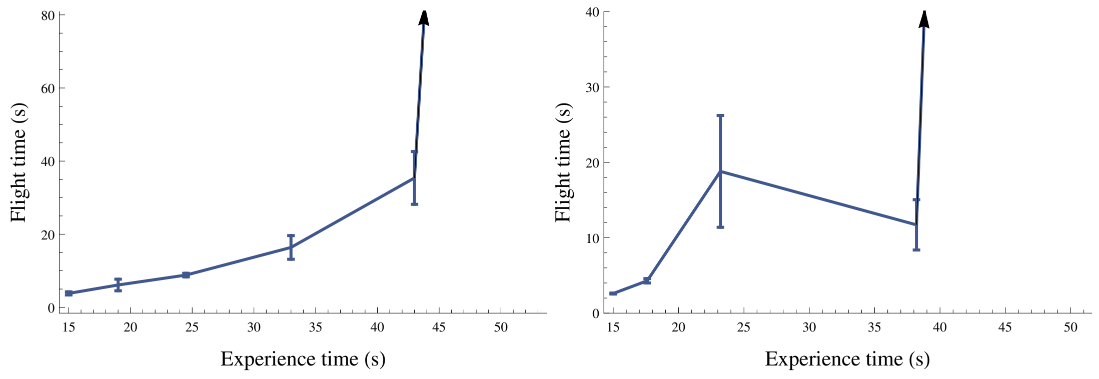
$$\tilde{C}(r) = -\exp(r/r_0) \tag{26}$$

instead. Note that this function is bounded in the range $[-1, 0)$, instead of the conventional $[0, 1]$. This choice has a specific goal, which is to address the exploration/exploitation trade-off, which was previously mentioned in section 2.1.7.

By choosing this reward-cost transformation in combination with a zero-mean GP prior we are effectively building a pessimistic agent, that will assign the maximum loss to unvisited regions of the state-space. In this way we discourage the agent from exploring new regions of the space, so reducing the risk of uncontrolled exploration and unstable policy searches. Note that we can safely adopt this pessimistic cost function because we are guaranteed that the helicopter always starts at the origin, which is the state with maximum reward, therefore ensuring that the region close to the optimal reward is explored.

In figure 15 we show two sample runs of the algorithm with the new cost function $\tilde{C}(r) = -\exp(r/r_0)$. Again, the performance of a policy is judged by the mean and standard deviation of the flight time it achieves.

With this modification the algorithm is more stable, and it even reduces the amount of training data required for the optimizer to find the global minimum. However, the algorithm is not completely reliable yet — while it succeeds around 70% of the time, it might fail in practice due to the very long computation time it takes to learn.



(a) Task 4 **(b)** Task 6

Figure 15: Performance of the policy trained by PILCO in each iteration, using a reward scaling parameter $r_0 = 10$ and a pessimistic cost function. At the black arrow performance of the policy jumps to 600s and the problem is solved.

Given the high demand of computational resources by the GP, if PILCO does not succeed before a large amount of data is collected, the computations will become impractical[8]. We address this and other problems in section 4.3 to add robustness to our modification of the original PILCO algorithm.

4.2 Incorporating prior knowledge

To contrast our results we can make use of the information about the helicopter simulator released by the Competition in previous years. Using some knowledge about the simulator can help us understand and evaluate our learning algorithm.

The simulator computes an approximation of the helicopter dynamics differential equation system described in [20] using the Euler numerical method for differential equations [34]. The experiment is terminated (i.e. the helicopter "crashes") if any of the state variables goes out of a pre-defined range, and a very large negative reward is returned.

The different tasks correspond to different aerobatic manoeuvres, implemented as a time-dependent velocity bias added to the helicopter's angular and linear velocities. In this way, each task effectively represents a different MDP.

At the same time, we know that the reward function implemented in the simulator is

$$r(\mathbf{s}) = \sum_{i=1}^{12} -s_i^2 = -\|\mathbf{s}\|^2 \ . \tag{27}$$

In combination, this velocity bias and this reward function give us a picture of how the different tasks are achieved by the helicopter. By adding controlled velocity offsets, the helicopter is forced to move along a pre-determined trajectory, while the origin moves with it. That is, by the Galilean relativity principle, the velocity offsets added to the helicopter can be seen as a displacement of the reference frame, such that the helicopter is effectively performing an aerobatic manoeuvre following a moving origin.

Since the goal of the helicopter is to remain close to the origin, we can merge the known reward function with the proposed reward-cost transformation function to substitute our reward GP by

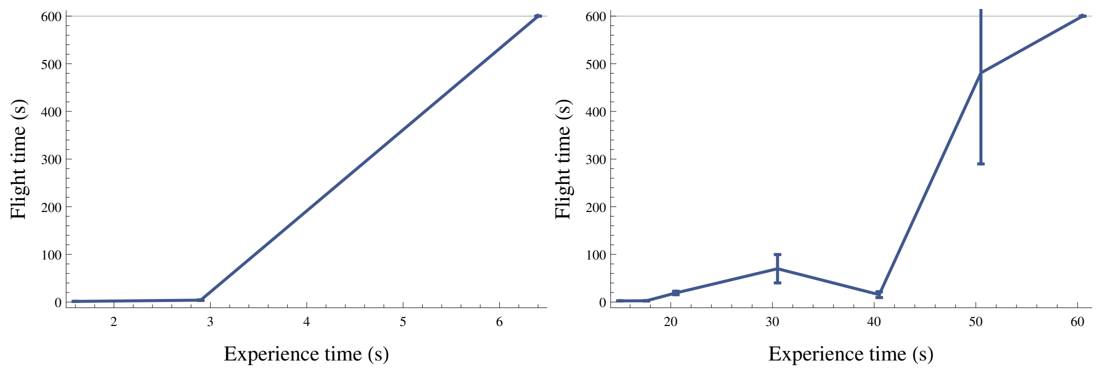
[8]At least impractical on a personal laptop, where these experiments were implemented.

the true cost function

$$c(\mathbf{s}) = 1 - \exp\left(-\frac{1}{2\sigma_c^2}\|\mathbf{s}\|^2\right) , \tag{28}$$

where σ_c represents the width of the global cost minimum centred around the origin. Note that in this case $c(\mathbf{s})$ is bounded in the standard range $[0, 1)$. In fact, this is the original cost function suggested by the authors of PILCO [2].

Knowing what function we were trying to learn, we can now run the original version of PILCO with the cost function (28). Two sample runs are shown in figure 16. Again, the performance of a policy is judged by the mean and standard deviation of the flight time it achieves.



(a) Task 2 **(b)** Task 6

Figure 16: Performance of the policy trained by PILCO in each iteration, using a cost width $\sigma_c = 1$ and the true cost function.

Using the true cost function provides more stability to the algorithm and increases its chances of success. However, it does not give a substantial improvement in neither training trials required nor total reward obtained by the policy. For more details about these problems and how to address them see sections 4.3 and 4.4.

As an aside, we mention a feature of the helicopter control problem as formulated in the Competition. Since in all tasks the goal is to keep the helicopter close to the origin, we can expect that policies could be generalizable — that is, a policy trained in one task could be used to control the helicopter in another task and succeed.

Experiments show that 80% of the trained policies perform well (i.e. do not crash) in more than 80% of the tasks. This is a good result, given the low complexity of the considered policy class (linear policies), and in comparison with other participants in the Competition (see section 4.5).

4.3 Improving performance

In this subsection we introduce further modifications to the original PILCO algorithm that overcome some of the problems observed in the previous sections. Our goal is to make the algorithm more reliable and reduce its chances of failure.

The main limitation of PILCO is the high computational complexity of the algorithms involved. Training the GP scales as $\mathcal{O}(N^3)$, which is dominated by the inverse matrix calculations in equations (18) and (19). Prediction scales as $\mathcal{O}(N^2)$, which is dominated by the matrix-vector product in equation (17). Furthermore, policy learning relies on the GP predicting function so it also scales as $\mathcal{O}(N^2)$. Therefore, we must be selective when manipulating the dataset, since small additions can quickly boost the computation time.

With the goal of speeding up the algorithm and making it more reliable, in this section we describe and justify three modifications to the original algorithm that

- Speed up the GP training by training each component separately;
- Avoid spending unnecessary time in policy search if the model is inaccurate; and
- Make the algorithm more robust against fluctuations in performance by controlling the data aggregation.

4.3.1 Speeding up GP training

In this problem the initial state of the task is always fixed – the origin. Furthermore, as discussed in section 4.2, the goal of the helicopter is to remain close to the origin during the whole task. The result is that this region of the state-space is heavily explored and the density of data points is much higher than in the rest of the space, sometimes resulting in an excessive, unnecessary amount of information that has to be processed by the dynamics GP.

This is specially relevant for the position and orientation variables (i.e. $\{x, y, z, q_x, q_y, q_z\}$). Since the helicopter is a physical system, the action of the controls or the wind imparts a force[9] to the helicopter that modifies its linear and angular velocities. However, since the simulator is built as a discrete-time one-step approximation we can expect that position only depends on the position and velocity in the previous time step. In other words, since position and orientation variables are related to the controls and the wind by a second derivative, it will take two time steps to see the effect. Since both the previous position and velocity are known with complete certainty, this means we can make essentially zero-error predictions.

While this would not be a problem with other supervised learning methods, it is something we should avoid when using a GP. The GP training algorithm implemented in PILCO can run into numerical problems if the signal-to-noise ratio (SNR) is too high, which is usually the case with the variables mentioned above. The SNR can be calculated using the hyperparameters in the definition of the kernel in equation (23), as

$$\mathrm{SNR} = \frac{\alpha}{\sigma_n} \ . \tag{29}$$

Note that since both α and σ_n are hyperparameters of the model we must train the GP to know whether numerical problems might be playing a role.

The simple approach to reduce the SNR is to artificially introduce zero-mean white Gaussian noise to the data, in order to increase σ_n. After adding noise, the GP is trained again, and if the SNR is still high this procedure is repeated until the SNR is below a certain threshold. However, this process can be highly demanding, since in each iteration the whole batch of GPs (in this case, 12) for all the predicted dimensions is re-trained.

To speed up this process, we modify the GP training algorithm used by PILCO. Instead of training the GP for all the predicted dimensions in the target at once, we sequentially train a separate GP for each of the dimensions and finally merge the 12 of them into the full GP.

This procedure reduces the overhead of having to re-train parts of the GP that were already valid, and in this way we achieve a faster training that conserves the numerical guarantees of our previous approach. The results of this improvement are illustrated in figure 17.

However, the specific training times depend heavily on the dataset. We can attain a better intuition by computing the ratio between the training times, shown in figure 18.

[9]More specifically, an acceleration, given that we are working within a classical-mechanical context.

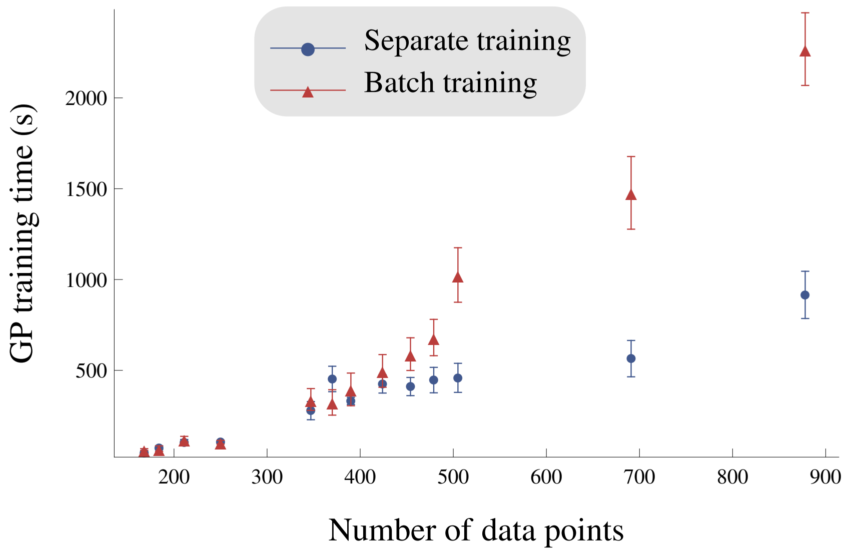


Figure 17: Computation time of GP training with (blue) and without (red) separate training

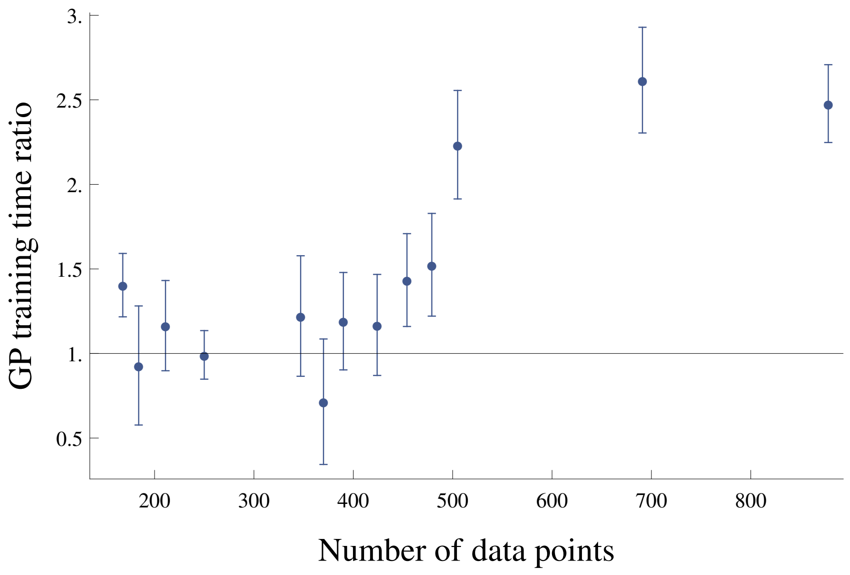


Figure 18: Ratio of the computation time of GP training with and without separate training. A ratio greater than 1 indicates that separate training is faster.

Separate GP training effectively reduces the total training time of the GP. While the speed-up is relatively small (∼25%) for small amounts of data, it proves to be an important improvement for larger datasets, in which the improvement can be of up to 250%. This modification has a large effect on the computation time of PILCO, specially in the advanced stages of the algorithm.

4.3.2 Limitations to learning

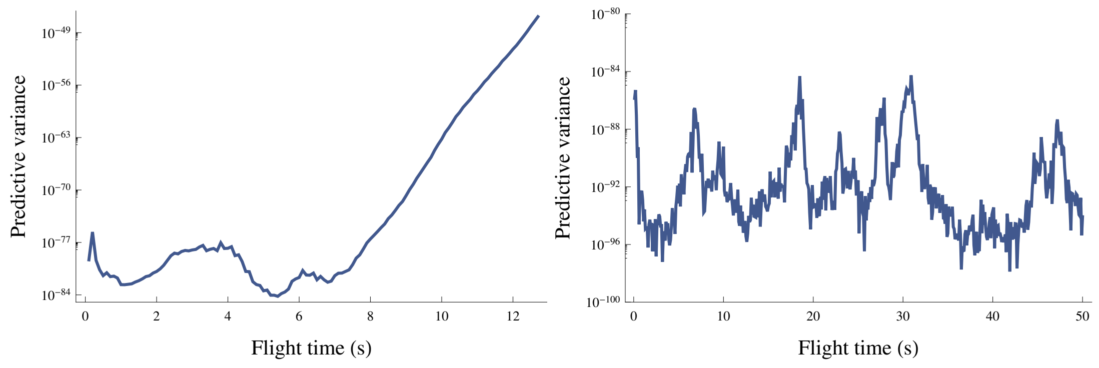
Having a multi-step algorithm like PILCO, in which there are parts relatively well separated from each other, we can ask which part is the most constraining in the algorithm. The natural approach is to explore and test which part of the algorithm is the limiting factor that slows down the learning process, and act in consequence.

In this case, we found that the major limitation to learning in the early iterations of the algorithm is the limitation in the predictive range of the dynamics GP.

The GP is very effective at estimating the dynamics of a certain region of the state-space with few data points, since it is a model-free algorithm that can learn the non-linear, asymmetric dynamics of the helicopter. But as all other supervised learning algorithms, it does not extrapolate well to unseen regions of the state-space. More specifically, if the distance along dimension i between the boundary of the dataset and the new input is large compared to ℓ_i the GP will always predict the prior mean, $m(\mathbf{x}) = \mathbf{x}$ in this case.

To test the hypothesis that the GP is the limiting factor in the early stages of the learning process we measure the width of the multivariate normal distribution predicted by a GP along the trajectory of the helicopter. That is, given a certain trajectory $\{\mathbf{s}_t, \mathbf{a}_t\}$ for $t = 1, ..., T$, we use equation (17) to calculate the predictive variance estimated by the dynamics GP. At each time step t we quantify the width of the distribution by calculating the quantity $|\Sigma_t|$, for the Σ_t in expression $p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) = \mathcal{N}(\boldsymbol{\mu}_t, \Sigma_t)$.

In figure 19 we show these plots computed before and after the algorithm has succeeded in the task.



(a) Early in the learning process **(b)** After learning a successful controller

Figure 19: Width of the predictive Gaussian distribution (measured as the determinant of the covariance matrix, $|\Sigma|$) at each point along the helicopter's trajectory, before and after learning.

The most salient difference between these plots is that when the algorithm is in the early stages of learning the width of the predicted distributions increases significantly, whereas once it has found a successful policy the width remains relatively constant.

The observed increase in uncertainty means that the helicopter is leaving the zone where the previous datasets were taken, and the GP responds by having more uncertainty in its predictions. Given this observation we see that to build a good dynamical model of the helicopter with a GP we do not need just more data, we need *different* data. We need to explore different regions of the state-space, and that is done by using different policies.

We can reinforce this hypothesis by ruling out the policy search as the limiting step. To do this, we compute the expected immediate cost along the trajectory predicted by the GP. Results are shown in figure 20.

In both situations the policy learning algorithm estimates a low cost along the predicted trajectory. However, since one of the policies fails, the predictions must necessarily be wrong. This shows that the policy search algorithm effectively finds a good policy with the available dynamical model, so it is the dynamical model that limits the policy search, and not the opposite.

At this point we recall that to train the policy we must specify a value for T, the number of

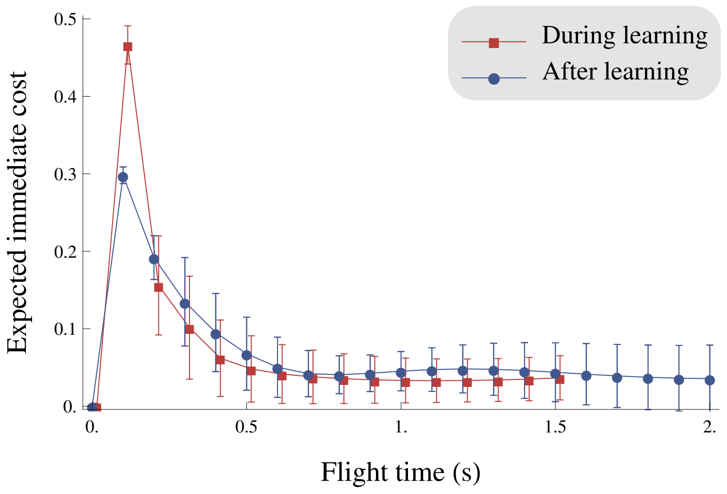


Figure 20: Expected immediate cost predicted by PILCO's policy search algorithm, computed during the learning process and after learning a successful controller.

time steps the GP predicts to compute J^π. Intuitively, a larger T leads to a better policy, since the predictions look further ahead in time and result in more solid policies. However, since we know the model is not very good in the early stages it is not convenient to set a high T, because it will increase the computation time without leading to a meaningful improvement.

Instead, we start learning with a small value of T and increase it in each iteration. This way we make sure that little effort is invested in training when the model is still bad. As more data is collected, the GP becomes more reliable and T increases, so that the trained policy makes more anticipating cost predictions.

The results of incorporating this modification and the one proposed in 4.3.3 are shown in figure 15 and are detailed below.

4.3.3 Adding robustness

Given the high computational complexity of the algorithms involved, we must be very careful when adding new points to the dataset and training the GP. We want our new data to be informative, and we want to avoid unnecessary iterations of the algorithm.

In this direction, we can face two problems during the execution of PILCO:

Adding too many data points: The problem arises when a learned policy is good enough to survive for a long time, but not good enough to survive the 6000-step interval set by the Competition. If this happens, plain dataset aggregation will result in a very large dataset, that will take very long (and unnecessary) time to train. This is the most common cause of the unsuccessful attempts mentioned in section 4.1.

To make our algorithm robust against this problem, we modify slightly the dataset aggregation step. Instead of merging the whole dataset, we set an upper limit ΔN_u to the maximum number of data points added to the dataset. If the generated trajectory is longer than ΔN_u time steps, we take the first ΔN_u data points and ignore the rest of the trajectory.

Adding too few data points: Similarly, it could be the case that due to statistical fluctuations the noisy environment we are dealing with could knock the helicopter down very soon, resulting in a very small new dataset to aggregate. If this is the case, in the next iteration the algorithm would train the GP and the policy again, with only a small amount information more than the last iteration, and will probably produce a similar policy after wasting valuable computation time.

To make our algorithm robust against this problem, we modify the policy application step. Instead of running the policy once and proceeding to the GP training again, we set a lower limit ΔN_l to the length of the recorded trajectory, such that if the trajectory is shorter than ΔN_l the policy is run again until the limit is surpassed. If this does not happen in several trials, we ignore the ΔN_l limit, aggregate the small new dataset anyway and continue with the algorithm.

With this modification, our algorithm becomes more robust against unusually poor policies (that would make it train an unnecessary iteration) and against unusually good policies (that would make it train with an unnecessarily large amount of data).

The result of applying these changes, the ones above and those in section 4.3.2 are the sample learning curves depicted in figure 21 computed for four randomly selected tasks.

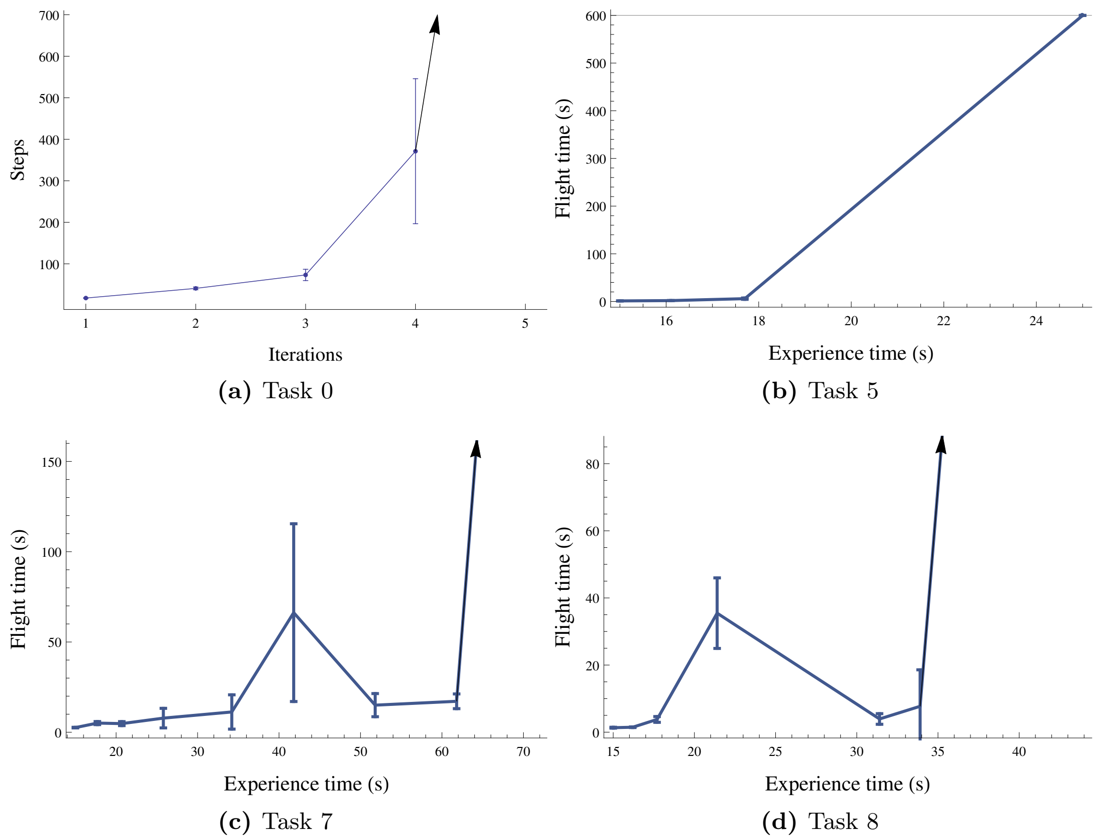


Figure 21: Performance of the policy trained by PILCO in each iteration, using a reward scaling parameter $r_0 = 10$, modified data aggregation and increasing prediction horizon. At the black arrow performance of the policy jumps to 600s and the problem is solved.

The result is that learning now requires more iterations, but the procedure is faster and more reliable. Now the algorithm succeeds close to 100% of the trials. In other words, with the previous modifications we have traded a small worsening in the number of trials needed to obtain a more reliable and fast algorithm.

4.4 Scoring higher rewards

The only performance criterion in the Competition is guided by the reward achieved by the agent. Thus, it makes sense to measure and evaluate the techniques we have used based on the rewards they obtain. We measure the performance of a policy by the total reward (or return) it obtains using the true reward function as shown in section 4.2, i.e.

$$R = \sum_{i=1}^{T} r_i = -\sum_{i=1}^{T} \|\mathbf{s}_i\| \ . \tag{30}$$

We recall that in the algorithms used above the learning process is finished when the controller can successfully execute a 10min flight. Thus, in each run learning is stopped at a different point. One could expect that the more data points collected by that time, the more accurate the models will be, and thus the better the policy will perform as a result.

To understand this relation we plot the average and standard deviation of the reward obtained by each trained policy as a function of the experience time required to train it. The result is depicted in figure 22.

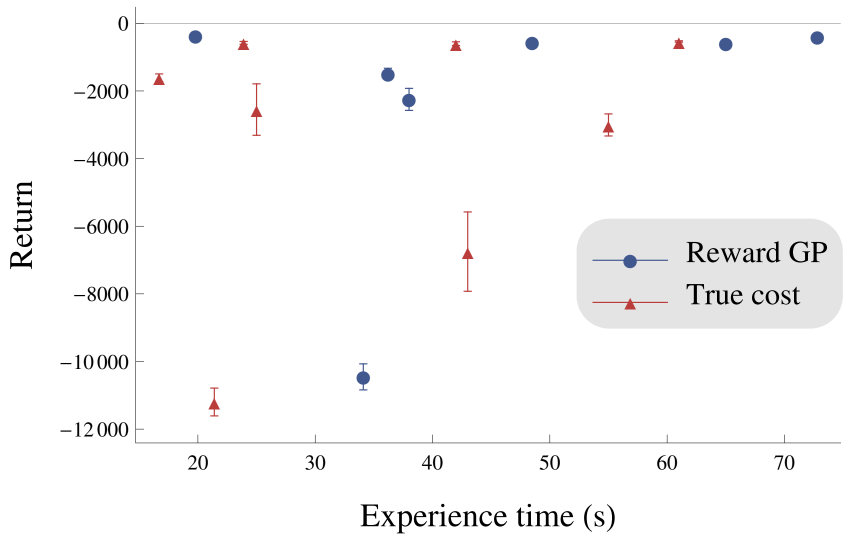


Figure 22: Return obtained by policies trained in each task versus the experience time they required to train. Learning is interrupted as soon as the policy completes a 10min flight. Points in blue correspond to policies trained with a reward model GP and points in red to policies trained with the true cost function.

We note the absence of correlation between the experience time (i.e. number of data points) inverted in the training of a policy and its final performance. This reinforces the idea that, when training a GP, it is crucially important to maintain data-efficiency by intelligently sampling the state-space.

This represents a new obstacle in the work of improving the algorithm — given that more data (and more computing time) will not necessarily improve the results, we must devise some more intelligent way to improve performance.

To look for potential improvements we must inspect the three elements of the algorithm separately: the reward model GP, the dynamics GP and the policy search.

The main function of the **reward model GP** is to guide the policy search algorithm to the global reward maximum at the origin. While PILCO is robust against the details of the cost function (such as the reward scaling parameter mentioned in section 4.1), one would expect it to be of crucial importance that the location of the maximum is correctly estimated.

In fact, the learned reward model GPs often fail to place the maximum reward at exactly the origin, missing by a distance that ranges from 0.1 to 10 units in the 12-dimensional state space. However, this might still not be the limiting problem. To test this hypothesis we evaluate the average performance of the policies trained using a reward model GP or the true cost function (28), as shown in figure 22. Results are shown in table 1.

Table 1: Average return of policies trained with the true cost function and with the reward model GP. See figure 22 for more information.

	Mean	Std. deviation
Reward model GP	-2299	3363
True cost function	-3353	3781

This shows that although the reward model GP is not perfect, it is not the most restrictive element, since it can achieve the same performance as the policies trained used the true cost function from the beginning.

The next part of the algorithm we can test is the **dynamics model GP**. To test its performance, we can use the GP to predict the cost using equation (28) along the helicopter's trajectory, and compare it with the real cost obtained. This result is shown in figure 23.

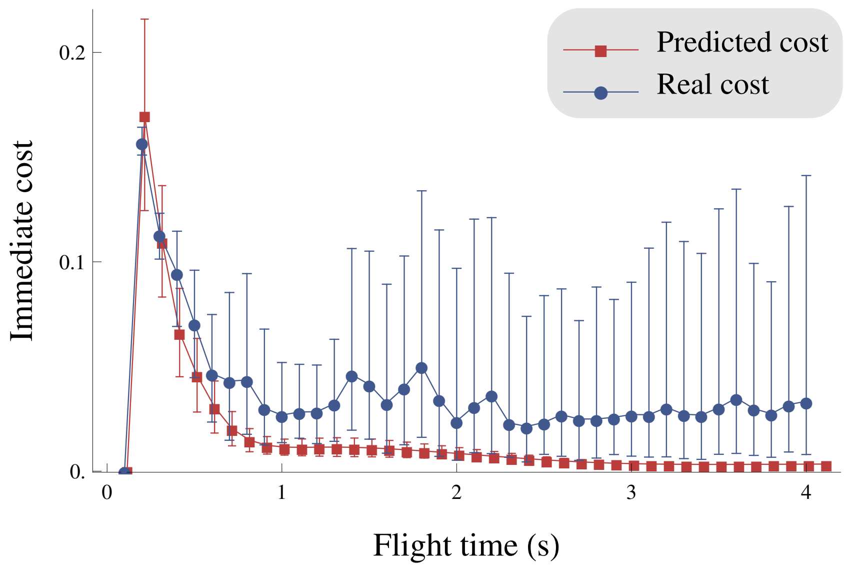


Figure 23: Predicted and real cost along the helicopter's trajectory, after learning a successful controller. Real cost at each step is calculated as the median of 200 runs and error bars represent the 95% confidence interval.

As observed, the dynamics GP is not completely reliable yet and the real cost is significantly (with 95% confidence) higher than the predicted cost. This indicates that we might be able to obtain a better performance by improving the GP.

As seen in section 4.3.1, GPs are computationally expensive to train, and their training time scales as $\mathcal{O}(N^3)$, with N the number of data points used. Since the flights are 6000-time-step long, it is impractical to add the whole trajectory to the dataset. Then we face the problem of selecting the most useful data points in a certain trajectory to increase the predictive power of the GP. By useful we mean that the new data point should provide information about new regions of the state-space, so selecting points from regions that have already been heavily explored should be strongly avoided.

To estimate the quality of the GP at each point we compute the *Negative-Log-Predictive-Density* (NLPD), which as its name implies is the negative log of the probability density of an n-dimensional Gaussian distribution, i.e.

$$\text{NLPD}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{2} \log |\Sigma| + \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) + \frac{n}{2} \log 2\pi \ . \tag{31}$$

The NLPD combines information about the distance between $\mathbf{x}$ and $\boldsymbol{\mu}$ with the total width of the distribution, which makes it a suitable quantity to evaluate probabilistic models. However, it is difficult to interpret and we need a baseline we can use for comparison.

If the GP model is correct, the state $\mathbf{s}_t$ follows a $\mathcal{N}(\boldsymbol{\mu}_t, \Sigma_t)$ Gaussian distribution, where the role of the GP is to estimate $\boldsymbol{\mu}_t, \Sigma_t$ based on $\mathbf{s}_{t-1}, \mathbf{a}_{t-1}$. Therefore, we can interpret the observed state $\mathbf{x}_t$ in the trajectory as a sample from the $\mathcal{N}(\boldsymbol{\mu}_t, \Sigma_t)$ distribution. This interpretation provides a simple way to understand NLPD values.

At any time t, given a $(\mathbf{s}_t, \mathbf{a}_t)$ pair, we use the GP process to estimate $\boldsymbol{\mu}_{t+1}, \Sigma_{t+1}$. Then we draw random samples from this multivariate Gaussian distribution and calculate their NLPD values. The average of these sample NLPDs is what we call the optimal NLPD — the expected value of the NLPD at any point in the trajectory if the dynamics model were perfect. We can compare the optimal and measured NLPDs to understand how far from the real distribution our predictions are.

Figure 24 shows an excerpt of the real and the optimal NLPD computed along the trajectory of the helicopter. The shaded area is the 2σ interval of the optimal NLPD. Following the previous argument, if the GP model were perfect, NLPD would be within the shaded area 95% of the trajectory. We see that even though the model is good enough to train a successful controller, there are still points with very high NLPD, indicating that the model is far from perfect, and we might be able to improve the controller by having a better GP.

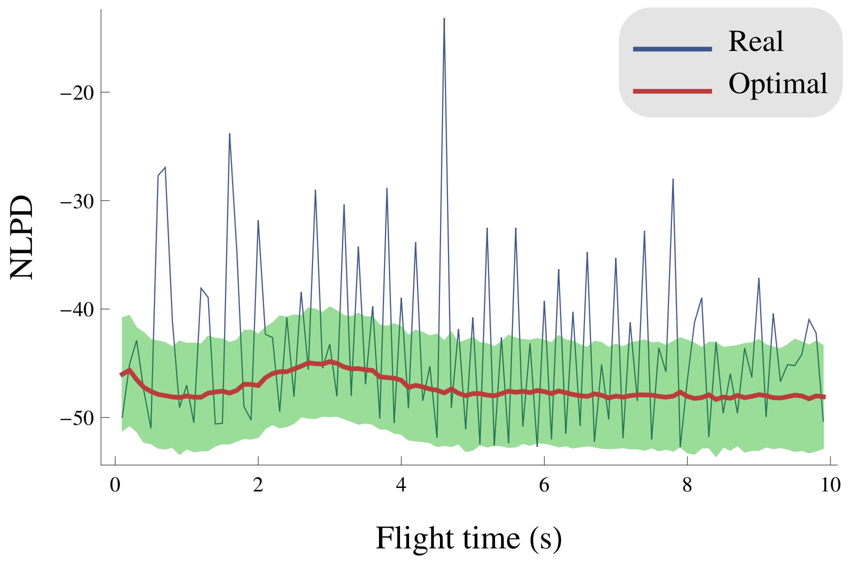


Figure 24: Negative-Log-Predictive-Density (NLPD) along the helicopter's trajectory. Red line represents the optimal NLPD and shaded area is its 95% confidence interval. If the GP model were perfect, NLPD would be within the shaded area 95% of the time.

Using NLPD we can devise a simple method for data selection. After performing a successful flight, we calculate the NLPD along the trajectory and add to the dataset the ΔN points with highest NLPD. This guarantees that we are adding the points where the model is either too uncertain (large Σ) or too wrong (large $(\mathbf{x} - \boldsymbol{\mu})$).

Last, to increase the return obtained by the agent we can also use a more complicated, non-linear **policy**. In this case we use a Radial Basis Function (RBF) controller with Gaussian basis, which is parametrized as the mean of a GP.

The analytic expression for the RBF controller is

$$\tilde{\pi}(\mathbf{s}) = \sum_{i=1}^{N_c} w_i \exp\left(-\frac{1}{2}(\mathbf{s} - \mathbf{c}_i)^\top \mathbf{W}(\mathbf{s} - \mathbf{c}_i)\right) , \tag{32}$$

where $\mathbf{W}$ is a weight matrix that plays the same role as the length-scales in equation (23) and the $\mathbf{c}_i$ are the N_c centres of the Gaussian basis functions that act as the inputs for the GP. Since this policy also has an analytic gradient, we can use the same methods described in section 3.3 to learn the parameters $\mathbf{c}_i, \mathbf{W}$.

Finally, we proceed to evaluate these extensions (data selection for the dynamics GP and a non-linear policy) with the aim of achieving the highest return possible. According to table 1, linear policies achieve an average return of -2826. RBF policies achieve an average return of -747, outperforming the simpler linear policies.

Furthermore, we can extend the learning period using data selection to achieve even higher return. Figure 25 shows a sample run of the algorithm using a non-linear policy and NLDP-based data selection.

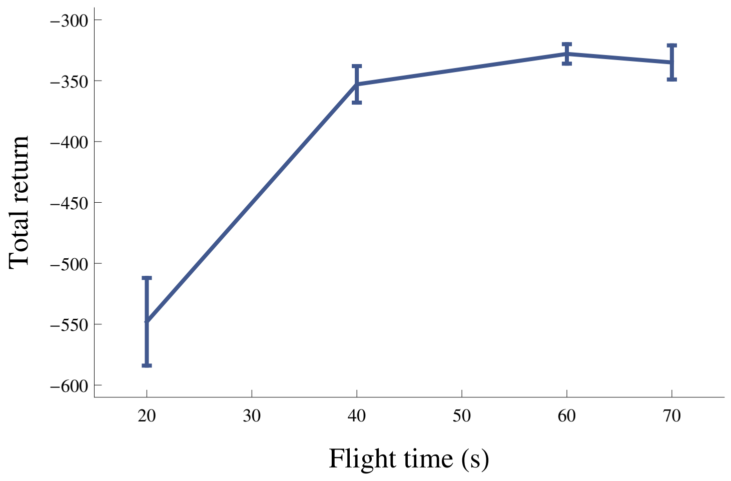


Figure 25: Total return achieved by a non-linear policy after the first successful flight. Error bars represent the standard deviation of the return computed in 20 10min flights.

Preliminary results indicate that these extensions can indeed achieve a higher reward in the helicopter task, although they also seem to be more unstable. Data selection does not always succeed in increasing the return. The thorough exploration of these methods and other alternatives to increase returns is an important part of the future work following this study.

4.5 Comparison with previous Competition winners

To understand how these results compare to the state-of-the-art in Reinforcement Learning we compare our results with those of the participants in previous Competitions. We have been able to track three teams from the top scores of the 2008, 2009 and 2013 Competitions.

J.A. Martín and J. de Lope [28] developed an online evolutionary RL method based on value learning on an artificial neural network, and obtained the second place in the 2008 Competition. In their paper they describe their method, but no results are presented and no code is available. Thus, comparison is not possible.

A. Asbah et al. [27] used a method based on Kernel-Based Stochastic Factorization (KBSF) that gave them the second place in the 2013 Competition. A. Asbah et al. have an approach similar to ours, in the sense that they keep the amount of prior knowledge to the minimum, and use the same algorithm to tackle all the tasks. They use a value learning TD approach, and as such, it takes a high number of interactions to train, of the order of 10^5 (equivalent to 3 hours of experience). Furthermore, after 17 hours of experience their agent was not able to survive the 6000-time-step limit set by the Competition in 2/10 tasks, and in their own words "the algorithm was unable to consistently fly the helicopter for more than 1000 steps". They provide no results in terms of rewards.

R. Koppejan and S. Whiteson [25][26] won the second place in the 2009 Competition and won the 2008 and 2013 competitions, and provide the most in-depth discussion of the generalized helicopter control problem available, to the best of our knowledge. Their discussion is broad and they provide multiple results, so we devote most of this section to the comparison between Koppejan and Whiteson's work in the 2008, 2009 Competitions [26] and after [25].

Koppejan and Whiteson (KW) provide three main results — using direct policy search, using dynamics model learning and using dynamics and wind model learning. In both model learning cases previous knowledge of the differential equations and wind patterns was used.

In all of the cases they use a highly engineered, expert-designed Multi-Layer Perceptron (MLP) topology, that outperformed the state-of-the-art topology-optimizing evolutionary methods [42]. Furthermore, for the Competition KW initialize the network with different specialized policies, that vary according to the task under consideration (that is guessed by the agent in the first few runs).

In short, KW use a high amount of previous knowledge in their approach. For the case where they use the smallest amount of previous knowledge (direct policy search (DPI) with specialized baseline policy initialization) they need several tens of thousands of 6000-step episodes to achieve their best result.

Their result is indeed impressive, achieving a total return of around $R = -130$ during a 6000-step episode. This is equivalent to remaining on average 0.1 units of distance away from the origin in the 12-dimensional state-space.

In table 2 we provide a short comparison between KW's result and our proposed method, in terms of maximum return (i.e. total reward during the 10min flight), number of training trials and prior knowledge. The maximum return is compared with the single best policy found by each method. Average performance is not reported in KW's paper and thus no comparison is possible. Additionally, KW do not provide the total number of data points used in training (i.e. experience time), so direct comparison is also impossible. However, given that they start from a moderately good baseline policy, we could expect the experience time to be close to 10min per trial, in each of the $\sim 10^4$ trials of the experiment.

As shown in the table, PILCO achieves a comparable, but lower performance in terms of maximum return, but drastically outperforms the KW method in the number of training trials (i.e. experience time needed), and additionally it does not use prior knowledge of any kind.

The results of the KW team in table 2 correspond to the performance of their human-designed MLP. They also report the performance of policies trained on a simpler, Single-Layer Perceptron

Table 2: Summary of the comparison between the record-holder Koppejan-Whiteson MLP method [25][26] and the proposed algorithm.

	KW (DPI)	KW (dynamics model)	KW (wind model)	Mod. PILCO
Max. return	-132.6	-142.25	-126.6	-328.0
Training trials	$\sim 4 \times 10^4$	$\sim 6 \times 10^3$	$\sim 6 \times 10^3$	≤ 10
Problem-specific prior knowledge	Baseline policy	Baseline policy and dynamics ODE structure	Baseline policy, dynamics and wind ODE structure	None

(SLP), that obtains a maximum return of -496.2 on the policy it was trained and an average return of -2.508×10^6 on the rest of the tasks. Our maximum-scoring policy achieves a return of -328.0 on the task it was trained and -1.471×10^4 on the rest of the task, giving a better performance than the SLP in both cases.

We can also compare the performance of our modified version of PILCO with the KSBF algorithm of [27], that, unlike KW's method, does not use any prior knowledge. With this comparison, summarized in table 3, we can understand where PILCO stands among the zero-prior-knowledge helicopter controllers.

Table 3: Summary of the comparison between the runner-up Asbah et al. [27] and the proposed algorithm. Success rate is measured on the Competition's 10 training tasks, and the number of training trials is only compared for the tasks that KBSF succeeded.

	KBSF	Modified PILCO
Success rate	8/10	10/10
Training trials	$\sim 10^5$	≤ 10

To the best of our knowledge, no other algorithm has been known to solve more efficiently the generalized helicopter control problem with no prior knowledge[10]. In this sense, this work marks a breakthrough in the learning of generalized helicopter controllers.

5 Conclusions and future work

Contributions. In this work we have successfully solved the generalized helicopter control problem, by training a controller able to perform 10 different aerobatic tasks without using any problem-specific prior knowledge. Our approach brings an improvement of several orders of magnitude in the number of training iterations compared to all the winners of the RL Competition in previous years [25][26][27][28]. In terms of maximum obtained return, our method achieves comparable, but nonetheless lesser performance than other methods that incorporate large amounts of prior knowledge and agent-system interaction. In comparison, zero-prior-knowledge methods so far have not managed to successfully solve the problem — this is the first time in the Competition that an algorithm has been able to succeed in the 10 tasks without using expert demonstration, baseline policies or knowledge of the helicopter dynamics.

The algorithm can find a successful policy, capable of surviving the 10 minute flight required by the Competition in less than 10 trials, which in most of the cases is below 1 minute of agent-environment interaction. To obtain this result we have modified the base PILCO algorithm to

[10]Assuming that, if there were any, the authors would have entered the Competition or would have been cited by the organizers or any of the participants (e.g. A. Asbah et al. claim their results to be the best available by the end of 2013).

relax the requirement of a fixed target state, by incorporating a new Gaussian Process that learns an unknown reward (or cost) function. Additionally, we have modified PILCO to be more robust against fluctuations in the performance of the trained policies and to be slightly faster than the original version.

Future work. One of the major drawbacks of the method is that the algorithm is computationally demanding. A typical run can take several hours[11] to learn a successful policy, and in the worst case scenario the algorithm can take up to 12 hours. A possible simple improvement in this direction would be to trade some predictive power of the dynamics model for a faster performance, for instance ignoring the input dimensions in those cases in which the characteristic length-scale is much larger than the standard deviation of the input variable. Another possibility would be to implement faster methods of sparse GP that can perform well even in conditions of high signal-to-noise ratios. A sparse GP method [19] could also eliminate the problem of data selection by allowing us to efficiently use more data to learn the dynamics model, and therefore build a better policy that can achieve a larger reward.

As pointed out in section 4.3.2, the bottleneck in the learning process is the quality of the dynamical model of the helicopter. In this work we adopted a fairly simplistic position – advance quickly to collect more data until the model is good enough. A possible modification would be to incorporate some knowledge about the system in the form of a GP prior that accounts for the fact that the helicopter is a physical system by incorporating basic information like $m_{prior} = x_{t-1} + v_{t-1}\Delta t$. Note that this is not prior knowledge about the specific dynamics of the helicopter, but is a general statement applicable to any physical system.

[11]The simulations were tested on a HP EliteBook 2540p running Ubuntu 14.04.

References

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

[2] M.P. Deisenroth and C.E. Rasmussen. *PILCO: A Model-Based and Data-Eficient Approach to Policy Search.* Proceedings of the 28th International Conference on Machine Learning, 2011.

[3] C.E. Rasmussen and C.K. Williams. *Gaussian Processes for Machine Learning.* MIT Press, 2006. ISBN-10 0-262-18253-X, ISBN-13 978-0-262-18253-9.

[4] M.P. Deisenroth. *Efficient Reinforcement Learning using Gaussian Processes.* KIT Scientific Publishing, 2010. ISBN 978-3-86644-569-7.

[5] M.P. Deisenroth, D. Fox and C.E. Rasmussen. *Gaussian Processes for Data-Efficient Learning in Robotics and Control.* IEEE Trans. Accepted 2014. doi: 10.1109/TPAMI.2013.218.

[6] M.P. Deisenroth, G. Neumann and J. Peters. *A Survey on Policy Search for Robotics.* Foundations and Trends in Robotics, vol. 2, no. 1-2, pp. 1-142, Aug. 2013. doi: 10.1561/2300000021.

[7] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

[8] A. Wilson and R. Adams. *Gaussian Process Kernels for Pattern Discovery and Extrapolation.* Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

[9] D.P. Bertsekas and J. Tsitsiklis. *Neuro-dynamic Programming.* Athena Scientific, 1996. ISBN: 1-886529-10-8.

[10] B. Tanner and A. White. *RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments.* Journal of Machine Learning Research, 10(Sep):2133–2136, 2009.

[11] A. White. RL-Glue 3.04 Overview Manual. URL: `http://rl-glue.googlecode.com/svn/trunk/docs/html/index.html` . Accessed July 2014.

[12] P. Abbeel, A. Coates and A. Ng. Helicopter domain problem specifications. Reinforcement Learning Competition 2014 website. URL: `https://sites.google.com/site/rlcompetition2014/domains/helicopter`. Accessed July 2014.

[13] H. van Hasselt and M.A. Wiering. *Reinforcement Learning in Continuous Action Spaces.* Proceedings of the 2007 IEEE Symposium on ADPRL.

[14] R. Bellman. *Dynamic Programming.* Courier Dover Publications, 2003. ISBN 0486428095, 9780486428093.

[15] R. Sutton. *Learning to predict by the methods of temporal differences.* Machine learning, 3(1):9–34, 1988.

[16] G.A. Rummery and M. Niranjan. *On-line Q-learning Using Connectionist Systems.* (1994)

[17] W.B. Powell and J. Ma. *A Review of Stochastic Algorithms with Continuous Value Function Approximation and Some New Approximate Policy Iteration Algorithms for Multi-Dimensional Continuous Applications.* (2010)

[18] S. Ross, G.J. Gordon and J.A. Bagnell. *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning.* AISTATS 2011. arXiv:1011.0686v3.

[19] J. Quiñonero-Candela and C.E. Rasmussen. *A Unifying View of Sparse Approximate Gaussian Process Regression.* Journal of Machine Learning Research 6 (2005) 1939-1959.

[20] P. Abbeel, A. Coates, T. Hunter and A.Y. Ng. *Autonomous Autorotation of an RC Helicopter.* In 11th International Symposium on Experimental Robotics (ISER), 2008.

[21] P. Abbeel, A. Coates, M. Quigley and A. Ng. *An Application of Reinforcement Learning to Aerobatic Helicopter Flight.* NIPS 2006.

[22] A.Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. *Autonomous Inverted Helicopter Flight via Reinforcement Learning.* 11th International Symposium on Experimental Robotics (ISER), 2004.

[23] A.Y. Ng, H.J. Kim, M. Jordan and S. Sastry. *Autonomous Helicopter Flight via Reinforcement Learning.* NIPS 2004.

[24] R. Muneepeerakul, J.S. Weitz, S. Levin, A. Rinaldo and I. Rodriguez-Iturbe. *A Neutral Metapopulation Model of Biodiversity in River Networks.* Journal of theoretical biology, 245(2), 351-63. doi: 10.1016/j.jtbi.2006.10.005 (2007)

[25] R. Koppejan and S. Whiteson. *Neuroevolutionary Reinforcement Learning for Generalized Control of Simulated Helicopters.* Evol. Intel. (2011) 4:19-241. DOI: 10.1007/s12065-011-0066-z.

[26] R. Koppejan and S. Whiteson. *Neuroevolutionary Reinforcement Learning for Generalized Helicopter Control.* In GECCO 2009.

[27] A. Asbah, A. M. S. Barreto, C. Gehring, J. Pineau and D. Precup. *Reinforcement Learning Competition: Helicopter Hovering with Controllability and Kernel-Based Stochastic Factorization.* Proceedings of International Conference on Machine Learning (ICML), Reinforcement Learning Competition Workshop, 2013.

[28] J.A. Martín H. and J. de Lope. *Learning Autonomous Helicopter Flight with Evolutionary Reinforcement Learning.* EUROCAST 2009, LNCS 5717, pp. 75-82, 2009.

[29] J.A. Bagnell and J.G. Schneider. *Autonomous Helicopter Control using Reinforcement Learning Policy Search Methods.* Proceedings of the 2001 IEEE International Conference on Robotics & Automation. Seoul, Korea. May 21-26, 2001.

[30] H. Murao, H. Tamaki and S. Kitamura. *Application of Reinforcement Learning to RC Helicopter Control.* SICE Annual Conference in Fukui, August 4-6 2003.

[31] Y. Gao and F. Toni. *Compact State Representation for Tree-Structured RL.* Proceedings of the 30 th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

[32] J.G. Schneider. *Exploiting Model Uncertainty Estimates for Safe Dynamic Control Learning.* In NIPS, 1997.

[33] M.L. Eaton. *Multivariate Statistics: a Vector Space Approach.* John Wiley and Sons. pp. 116–117. ISBN 0-471-02776-6. (1983)

[34] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing.* Thomson Higher Education, 2008. ISBN-13: 978-0-495-11475-8.

[35] J. Nocedal and S.J. Wright. *Numerical Optimization* (2nd ed.). Berlin, New York: Springer-Verlag, ISBN 978-0-387-30303-1. (2006)

[36] O. Amidi, T. Kanade and J.R. Miller. *Autonomous Helicopter Research at Carnegie Mellon Robotics Institute.* Proceedings of Heli Japan '98, April, 1998.

[37] V. Gabillon, M. Ghavamzadeh and B. Scherrer. *Approximate Dynamic Programming Finally Performs Well in the Game of Tetris.* NIPS 2013.

[38] G. Tesauro. *TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play.* AAAI Technical Report FS-93-02. (1993)

[39] F. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion.* Princeton University Press. ISBN 0-691-09065-3. (2002)

[40] V. Mnih et al. Playing Atari with Deep Reinforcement Learning. arXiv: 1312:5602v1. (2013)

[41] B. Tastan and G. Sukthankar. *Learning Policies for First Person Shooter Games Using Inverse Reinforcement Learning.* Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[42] K.O. Stanley and R. Miikkulainen. *Evolving Neural Networks Through Augmenting Topologies.* Evolutionary Computation, 10(2):99-127, 2002.

A Code

The full code for the project is released under a GPL licence and hosted at the author's `helicopterRL` repository at Github,

`https://github.com/pmediano/helicopterRL` .