Contents

1 Literature Review: A Taxonomy of Reinforcement Learning

Machine learning can be thought of as subset, of artificial intelligence that makes use of statistical methods to allow machines to show features of learning, such as the inference of functional relationships between data [1], and improvement with experience [2]. There is often considered to be three main categories of machine learning: supervised learning, unsupervised learning and reinforcement learning [3]. In supervised learning, structure is often given to data before the learning exercise, for example in the form of an input and an output; in imitation learning for example, the agent is provided with examples of strategies and policies that have already been deemed to be good by its 'teacher' [4]. In unsupervised learning, there is no structure given to the data, and the learning exercise attempts to infer or impose structure through its algorithms. Reinforcement learning can be thought to sit in the middle, where unstructured data is given structure through an iterative interaction with an 'environment' such that a certain goal is reached through the maximisation of a reward [3, 5, 6].

Due to its relevance for this project, I will be presenting the latest research that has been done with regard to reinforcement learning in the rest of this review. First, the main flow diagram of reinforcement learning will the expressed, followed by a taxonomy of different methods developed for completing each of the components of the flow diagram. Finally, I will be presenting interesting research that has been carried out with regards to the use of Gaussian Process Latent Variable Models (GP-LVMs) in the context of reinforcement learning, and why it will form the main focus of this project.

1.1 Reinforcement Learning: What is it?

Reinforcement learning (RL) consists of an *agent* interacting with its *environment* by carrying out *actions*. The agent then receives signals (which include the environment's new *state* and *reward*) back from the environment which it then uses to select the action it will make next, such that the reward is maximised over time. This is illustrated in Figure 1-1.

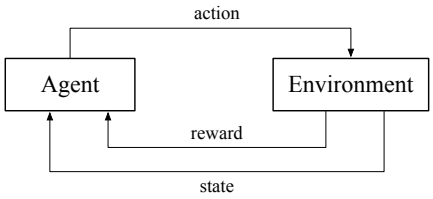

Figure 1-1: The interaction between the agent and the environment that defines the basic structure of reinforcement learning methods.

More formally, the agent receives a representation of the environment's states at time t, $s_t \in S^+$ where S^+ contains all the possible states that the environment can take. Based on this current state, the agent selects an action $a_t \in A(s_t)$, where $A(s_t)$ contains all the possible actions that the agent can take while being in state s_t. The mapping between the state s_t and the action a_t is called the agent's policy. Conventionally this is denoted by π_t. π_t can be deterministic, where $a_t = \pi_t(s_t)$, or probabilistic, the action is selected from a probability distribution defined by $\pi(a_t|s_t)$. Once a_t is enacted, the environment will provide the agent with the next state s_{t+1}, along with a reward r_{t+1}. The main goal of RL, or the Rl problem then is to find the policy that will maximise the long term sum of rewards R_t (Equation 1.1) [5, 4]. This is called the optimal policy π^*; RL algorithms attempt to find π^* using the reward signals by changing its policy as it gains experience [5].

$$R_t = r_{t+1} + r_{t+2} + ... + r_T \tag{1.1}$$

In order to do this, the agent must be able to evaluate how *good* it is to be in a certain state with the current policy in terms of the expected long term rewards. In other words, the *value* of being in state s is the long term reward that is expected by following the current policy after starting from s [5, 7]. This is given by Equation 1.2. This is called the *state-value function*.

$$V^\pi(s) = E(R_t|s_t = s) = E(\sum_{k=0}^{K} \gamma^k r_{t+k+1}|s_t = s) \tag{1.2}$$

where γ is the discount rate, and $\gamma \in [0, 1]$. It measures how much into the future the agent looks at; if $\gamma = 0$, the agent is myopic, and if $\gamma = 1$, the agent looks at all future rewards equally.

Similarly, the *action-value function* provides the expected long term reward for being in state s and following action a given the current policy. It is given by Equation 1.3.

$$Q^{\pi}(s,a) = E(R_t|s_t = s, a_t = a) = E(\sum_{k=0}^{K} \gamma^k r_{t+k+1}|s_t = s, a_t = a) \tag{1.3}$$

The framework of RL provides great flexibility in terms of where it can be applied to. The time step does not necessarily need to relate to fixed intervals of time, but can often be individual steps in a decision making process [5]. Furthermore it is often the case that no prior task-specific knowledge is needed; through the process of trials, the algorithm gains the experience it needs to have to find the best path to achieve its goal [6], and it does this without the presence of a teacher [7]. It is also important to note that RL is capable of being applied to problems to which classical machine learning and other solutions to simpler problems have been applied to. It is capable of solving problems that require both interactive sequential prediction as well as the assimilation of complex reward structures; most other methods are only able to reconcile a spectrum of one [4].

1.1.1 Key assumptions in Reinforcement Learning

Before moving on, there are some key concepts that need to be addressed in the context of RL.

- The first of these is the agent-environment boundary. The boundary between the two does not necessarily need to be physical [5]. More often than not, the agent is simply an abstract construct that is the decision maker, and the physical systems that it interacts with then form the environment. For example, in the case of a robotic arm, the environment would consist of all the motors and the structural components of the arm, while the agent would simply be the 'brain' that decides what torque to provide the motor with.

  A general rule that can be followed is that the agent-environment boundary is the limit of the agent's absolute control [5]; if the agent cannot arbitrarily change something, it is to be a part of the environment. It should be noted however that this does not form the limit of the agent's knowledge. That is, the agent can, potentially know everything

about its environment, and yet not be able to control everything [5].

- A common assumption in RL algorithms is that the states of system have the Markov Property. This property states that knowledge of the current state is sufficient to make decisions that also need to take into account all the historical states that were necessary to be at the current state. Mathematically speaking, a state is said to be Markov if and only if the one step prediction given by Equation 1.4 can be made by simply using Equation 1.5.

$$P(s_{t+1} = s', r_{t+1} = r' | (s_t, a_t, r_t), (s_{t-1}, a_{t-1}, r_{t-1}), ..., r_1), (s_0, a_0)) \tag{1.4}$$

$$P(s_{t+1} = s', r_{t+1} = r' | (s_t, a_t)) \tag{1.5}$$

  This assumption is powerful as it allows for a RL algorithm to be made simpler, as it only needs to take into account the current state to predict the next state and reward when searching for the optimal policy. A reinforcement learning task that consists of Markov states is called a Markov Decision Process (MDP) [5].

1.1.2 The many faces of Reinforcement Learning

Due to the generality of RL and its applications, there are many ways by which the different algorithms that have been developed in light of it can be categorised. These categorisations are shown in Figure 1-2.

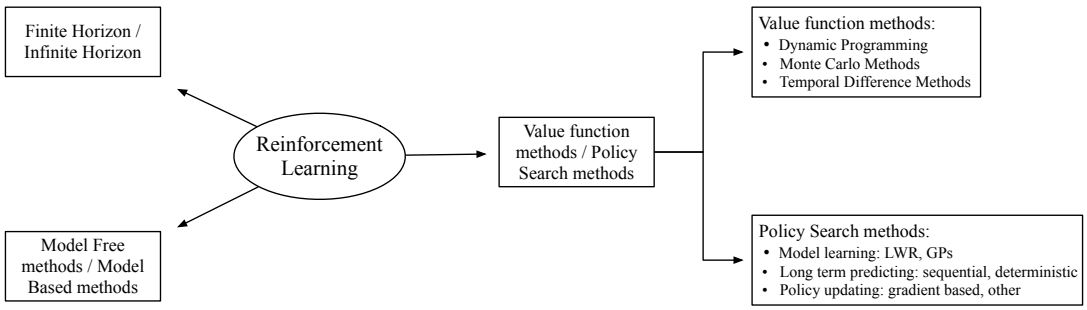

Figure 1-2: The major classifications of reinforcement learning algorithms. These are by no means the only classifications that exist, e.g. on policy / off policy methods.

Finite Horizon vs Infinite Horizon problems

Simply put this distinguishes between the case where T in Equation 1.1 and K in Equation 1.2 and 1.3 are finite and the case where they are infinite. In the former case, the *episode* is terminated after reaching a subset of S^+ known as the *terminal states*. These states are denoted by T, where $(T \cup S) \cap S^+$ and $T \cap S = \emptyset$. Such problems are called to be *episodic*, and as can be imagined, a sequence from a starting state to a terminal state is called an episode; in the case of the latter, where an episode lasts till infinity, the problems is said to be *continuous* [5, 7].

The definitions, or rather limitations of a finite horizon problem are sufficient for the task that was tackled by this project, and as such, no more will be said about this matter.

Model Free vs Model Based Methods

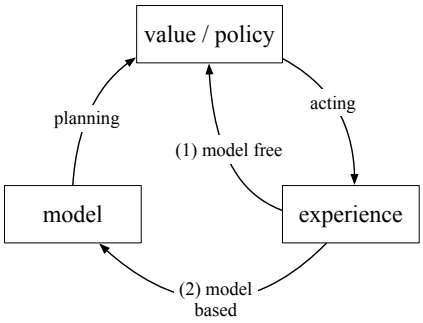

Figure 1-3: The relationships between key features of reinforcement learning and how they relate to (1) model free learning and (2) model based learning [5].

Figure 1-3 succinctly illustrates the differences between the paths taken by: (1) model free learning and (2) model based learning. In model free learning, the agent does not attempt to build knowledge of the environment, but rather uses the experience it gains through iterative interactions with the environment to directly affect its policy and value functions. In contrast, in model based methods, the agent use these experiences to create a model of what it believes to be the behaviour of its environment. It then uses the model to *simulate* the behaviour of its environment to *plan* its policy and value function [5, 7]. As stated by [5], *planning* here is defined as any computational process that produces or improves

a policy when given a model of the environment. This model can be thought of as being the *transition function* [7, 6], the *transition probabilities* [5] or the *transition dynamics* [6]. This can be notationally expressed as Equation 1.6.

$$M^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a) \tag{1.6}$$

As can be imagined, in the case of applications of RL to mobile robots, model free methods will involve a lot of real interactions with mechanical systems. This can lead to long experimentation times, as well as wear and tear of the physical components. For example, non-optimal policies of a bipedal robot could cause severe damage as a result of falling; robotic cars can could cause crashes, which could be fatal to both it and third-party observers. Model based methods however, will be able to carry out offline simulations to find currently best optimal policy, and only use these on the mechanical system; this has the advantages of being able to find the optimal policy with minimal interactions with the real mechanical system. Model based methods do however pose the problem of uncertainties caused by assumptions in the model created; this will be dealt with in Section 1.3, and will form a major part of this project. As such, for the most part of this project, model free methods will not be used. For completion however, some examples of model free methods will be mentioned in this chapter.

In [5], there is a clear distinction between *learning* and *planning*; the former is where real experience is used via direct interaction with the environment, while the latter is where simulated experience is used via interaction with the model of the environment, to then interact with the environment to further update the model. However, in the present work, due to a model based method being used for the most part, the term *learning* will be used to mean *planning* as defined by [5].

Specific literature that has been used to develop efficient methods of creating this model of the environment will discussed in a later section.

Value Function methods vs Policy Search methods

This classification describes the main strategies used by RL algorithms to carry out its main function of finding the optimal policy. Due to its importance, the next two sections provide the main strategy used by each, and list and describe the most widely used, and some new algorithms for each. Proceeding this, each will be compared with respect to their application in robotic motion problems.

1.2 Reinforcement Learning: Value Function Methods

Most classical reinforcement algorithms fall under this category. As the name states, these methods make use of the state-value function, or the action-value functions given by Equations 1.2 and 1.3 respectively, to find the optimal policy π^*. As mentioned before, π^* is the policy that will maximise the long term expected reward; notationally, by following π^*, we will produce a long term reward that satisfies the following for the state-value function and the action-value function respectively.

$$V^*(s) = \max(V^\pi(s)) \tag{1.7}$$

$$Q^*(s,a) = \max(Q^\pi(s,a)) \tag{1.8}$$

This logic holds because the value functions define a partial ordering of the policies; if π_1 can produce V_1 or Q_1, and π_2 can produce V_2 or Q_2, where $\pi_1 \neq \pi_2$ and $V_1 > V_2$ and / or $Q_1 > Q_2$, then there must exists a policy π^* that produces $max(V^\pi(s))$ or $max(Q^\pi(s,a))$.

Due to the recursive nature of the value functions (which is a result of Markov property), it is possible to find an expression for $V^*(s)$ in terms of the transistion function and the reward function. The reward function computes the expected reward from going from state s_t to state s_{t+1} after following an action a_t, as shown in Equation 1.9 [5].

$$L^a_{ss'} = E(r_{t+1}|s_t = s, a_t = a, s_{t+1} = s') \tag{1.9}$$

This expression for $V^*(s)$ is called the *Bellman Optimality Equation*, and is stated in Equa-

tion 1.10 for the state-value equation [5].

$$V^*(s) = \max_a \sum_{s'} M^a_{ss'}[L^a_{ss'} + \gamma V^*(s')] \tag{1.10}$$

Similarly, it can be expressed for the action-value function as Equation 1.11

$$Q^*(s,a) = \sum_{s'} M^a_{ss'}[L^a_{ss'} + \gamma \max_{a'} Q^*(s',a')] \tag{1.11}$$

As such, value function methods make use of these equations to obtain the optimal policy. The main algorithms that have been used in this regard include dynamic programming, Monte Carlo methods and Temporal Difference (TD) methods. To remain concise, only dynamic programming will be discussed due to a piece of work that uses this algorithm to introduce a key concept that will be important to the present work. More about the other methods can be found in [5] and [4].

1.2.1 Dynamic Programming

Equations 1.2 and 1.3 can be computed as a recursive sum as expressed by Equations 1.12 and 1.13 [5, 4, 8]. This is called the *Bellman equation*.

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} M^a_{ss'}[L^a_{ss'} + \gamma V^\pi(s')] \tag{1.12}$$

$$Q^\pi(s,a) = \sum_{s'} M^a_{ss'}[L^a_{ss'} + \gamma V^\pi(s')] \tag{1.13}$$

Typically, dynamic programming iteratively computes the $V^\pi(s)$ by recursively moving through all the states that would result from following the policy π. It would then compute a $Q^{\pi'}(s,a)$ where a is sampled from a different policy π'. If $Q^{\pi'}(s,a) > V^\pi(s)$, then it follows that $V^{\pi'} > V^\pi$. Thus there must exist a policy that produces a maximum Q given all the actions that the agent can take at that particular time step. It follows then that, for a finite problem MDP, with a finite number of policies, an iterative loop of each of these processes will converge to the optimal policy in a finite number of steps [5].

These two steps have been called *policy evaluation* and *policy improvement* respectively, and the process of finding the optimal policy through these is called *policy iteration* [5, 8]. A key drawback of is that it requires several iterative computations in each of its processes. A way around this is to use *value iteration*, whereby policy evaluation and policy improvement are combined by directly updating the value function based on the Bellman Optimality Equation, given by Equation 1.10 [5, 4]. This process can dramatically reduce the number of iterations required for convergence.

Typically, if this algorithm is applied to a MDP with fully known transition dynamics, it is indistinguishable from optimal control [8, 9]. As such dynamic programming has many applications in varying fields [9].

In [8] and [10], the authors suggest the approximation of the optimality equations through the use of gaussian process (GP) regression. Through their Gaussian process dynamic programming algorithm (GPDP), they create two value function GP models using training data, which are then used to predict the values for the required states in the dynamic programming loop. In this way, they are able to be more data efficient. Furthermore, GPDP is independent of the time-sampling frequency, while classical dynamic programming would require more memory and computations as the total number of states in the entire space grows. In both papers, the authors suggest the use of GPs to model the transition dynamics, where they are *a priori* unknown. In [10], the authors used an offline method, where training data for the dynamics model was first gathered and the model was trained without interacting with the physical system; in the next year, they extended this algorithm to an online method, where the dynamics model and the value models are developed within the dynamic programming loop. Due to the use of probabilistic approximations of these models, both algorithms were capable of achieving higher data efficiency than classical dynamics programming. This will be a recurring theme caused by the use of GPs; the theory on GPs and their advantages will be discussed in a later chapter.

1.3 Reinforcement Learning: Policy Search Methods

In the previous section, the algorithms discussed were designed around the notion of a value function. An alternative method of carrying out reinforcement learning is to place an emphasis on the policy itself. Thusly, policy search methods parameterise the policy in terms of a parameter space Θ, where the policy is then a function of $\theta \in \Theta$ [4, 11]. As such, in policy search methods, the optimal policy is found by exploring the parameter space with the goal of (once again) maximising the expected long term reward. In this way, a value function need not be found or approximated.

A key advantage of this, particularly for applications in the field of robotics, is that it provides better scalability to high dimensional problems, which is often the case in robotics, reducing the dimension of the space that is searched [4, 11]. That is, instead of searching the entire state and action spaces, the algorithm only needs to search the parameter space, which more often than not, can be solved as a lower dimensional problem. Furthermore, appropriate choice of the parameterised policy can guarantee stability and robustness [11].

These methods can be classified as being either model-free or model-based. This classification, as mentioned in Section 1.1.2 relates to the algorithm itself. In addition, it is possible to classify these in terms of the representation of the policies; they can be either time dependent, or time independent [11]. The former case is the policy changes with time, that is it can be represented as $\pi(s, t)$. In the latter, the policy doesn't and is represented simply by $\pi(s)$. In the interest of conciseness, only model-based, time independent methods will be described, as the others are beyond the scope of this project. However, refer to [11] for more discussions about these. Popular time independent policy representations include the linear model [11] and the non-linear Radial Basis Function (RBF) network [11, 12, 13].

When carrying out policy search algorithms, three main considerations, or rather challenges need to be addressed. These, as can be guessed from Figure 1-3, are the problems of the structure of the model the agent needs to learn, how to use these to generate long term predictions of states and rewards, and how to the policy can be updated in light of these; namely, these are model learning, internal simulating and policy updating [11].

The following subsections detail possible solutions found in the literature, followed by some recent algorithms that have been developed to make use of these.

1.3.1 The problem of which model to learn

The problem of which model to learn is an important one. The model needs to be able to accurately and efficiently provide an emulation of the environment; errors from an incorrect, biased or under/over confident model can propagate through the rest of the algorithm leading to perhaps catastrophic conclusions. According to [11], two main methods have been currently used for this problem in model based policy searching are locally weighted (Bayesian) regression (LWR) [14] and GPs. Key features of these include the fact that they are probabilistic and non-parametric. Being probabilistic means that they can express uncertainty about their beliefs of how the environment behaves; this can later be utilised in the other two problems mentioned. The latter allows these models to be as functionally variable as possible. Parametric models such as polynomial and trigonometric often have a standard shape and are limited to problems that behave as such; take for example the problem of trying to fit data that behaves quadratically using a linear model.

LWR methods build upon the ideas of bayesian learning used for in linear regression models (as given by Equation 1.14 for the case of the transition dynamics).

$$s_{t+1} = [s_t, a_t]^T \phi + w, \quad w \sim \mathcal{N}(0, \Sigma_w) \tag{1.14}$$

It does this by equipping weights to every test input that is calculated as some function of the distance between the test input and a training point. These functions could be gaussian shaped [11], exponential, quadratic to name a few, utilising unweighted euclidean distance, diagonally weighted euclidean distance and many other measurements of the distance [14]. These are then used to find a regression based on locally linear approximations of the underlying function [11], such that a cost defined as some sum of weighted differences between points is minimised [14].

GPs will be discussed in a later section.

1.3.2 The problem of making long term predictions

This next problem relates to how, given that the model is known for example through the methods discussed in the previous section, long term predictions of the environment's state progression can be made. In [11] it is stated that there are two main methods that have been utilised: sampling based trajectory prediction nd deterministic approximate inference.

The former of these can be best thought of as recursively evaluating the one step transitions, moving from the starting states to the terminal states of the episode. This is suggested by [15]. An example of this method is that used for this in the PEGASUS framework [16]. PEGASUS (Policy Evaluation-of-Goodness and Search Using Scenarios) is a framework that makes use of the observation that a probabilistic MDP or POMDP (Partially Observed MDP) can be transformed to an equivalent MDP or POMDP where the state transformations are deterministic [11, 16]. More details about this transformation can be found in [16]. After this key transformation, PEGASUS uses externally generated random numbers to simulate noise, and inputs these and a randomly sampled initial state into the new deterministic model to generate sequentially the new states and rewards, the latter of which it averages.

The alternative method to this is to concatenate the one step predictions into a probability distribution over the trajectory (τ) of the agent. That is, find

$$P(\tau), \quad \text{where } \tau = (s_0, s_1, ..., s_T) \tag{1.15}$$

Therefore, in order to find the probability distribution $P(s_{t+1})$, it is necessary to marginalise the joint distribution of $P(s, a)$, which is used in the transition dynamics given by Equation 1.6. This operation can be given by the following integral.

$$P(s_{t+1}) = \iint P(s_{t+1}|s_t, a_t)P(s_t, a_t)\mathrm{d}s_t\mathrm{d}u_t \tag{1.16}$$

If the transition function is nonlinear, and $P(s_{t+1})$ is non-gaussian, this integral is analytically intractable [11]. The solution to this is to then approximate $P(s_{t+1})$ as a gaussian

$\mathcal{N}(s_{t+1}|\mu^s_{t+1}, \Sigma^s_{t+1})$, the mean μ^s_{t+1} and covariance Σ^s_{t+1} of which can be calculated through various methods, the most common of which are linearisation, sigma-point methods and moment-matching [11].

1.3.3 The problem of policy updating

The final step in policy searching is to then make use of the model learned and the long term predictions made to update the policy such that an optimal policy is found (as defined previously). Once again, in this sense there are two main classifications: gradient free and gradient based.

Gradient free methods include but are of course not limited to the Nelder-Mead method (a heuristic simplex method), hill climbing (a local search method) [11] and the particle swarm optimisation policy (a population-based search method) [7]. These methods often require low computational effort compared to other methods as listed later due to its comparative simplicity. However, they are often disadvantaged by the fact that convergence to the optimal policy is slower [11], and the fact that they can only be practically used for policies with parameters in the order of tens [17].

Gradient based policy update methods are similar to gradient descent and ascent methods. They follow the gradient of the expected reward function to suggest new parameters. This can be expressed as Equation 1.17 for a defined step size α.

$$\theta_{i+1} = \theta_i + \alpha \frac{\mathrm{d}R}{\mathrm{d}\theta} \tag{1.17}$$

The gradient of the expected long term reward can either be approximated or found analytically. Processes such as *finite difference methods* [4, 11] and the method used by the PEGASUS framework [16] can be used. If the policy, the reward function and the learned transition model are differentiable, then the gradient can be found analytically. This has the advantages of not suffering from estimation errors, as well as being scalable to policies with a lot of parameters.

1.4 Reinforcement Learning: Key works from Literature

The following subsections will detail some key pieces of recent work that was thought to be relevant to the present work. This list is therefore not exhaustive; a more comprehensive and inclusive list can be found at [4].

1.4.1 The PILCO framework

[6, 17] present an interesting framework, called PILCO (Probabilistic Inference for Learning Control), that will form the main starting point for the present work. PILCO contains three layers that essentially address each of the problems mentioned in Section 1.3; the bottom layer learns a gaussian process model of the transition dynamics, the intermediate layer then uses either linearisation or moment-matching to estimate the probability distribution of Equation 1.16 and uses this to carry out policy evaluation, and the top layer finally carries out policy updating by using analytical gradient evaluation. The framework uses either a simple linear or a non-linear RBF network as its initial policy. The cost function (the negative reward function) in this framework is defined as the euclidean distance between the current state and the goal terminal state mapped on to gaussian shape.

In [17], this framework was tested in a variety of problems, including classical reinforcement learning problems like that of a simulated double pendulum and the cart-pole swing up (simulated and real), as well as higher dimensional problems, like unicycling, where a unicycle had to be balanced, and the control of a low-cost robotic arm. Through these exercises, it was shown that PILCO had exceptional efficiency in terms of using data; for the simulated cart pole and double pendulum problems, it learned sufficiently to succeed 95% of the time after just $15 - 20s$ and $50s$ of interaction time respectively. For the cart pole problem, PILCO performed several orders of magnitude better than existing methods.

The main reason for PILCO's success is its use of probabilistic modelling and inferencing, in particular with GPs. The power of GPs is that they provide a probability distribution over functions [3, 7, 6]. That is, it provides a representation of all the possible functions that can fit the training data, and essentially ranks them according to how likely they are to occur. Thusly, GPs allow for a lot unknown structures of the data set to be represented

within itself, reducing significantly the errors of model bias, and allowing the processes that follow to be more data efficient.

1.4.2 Autonomous helicopter flight

In [18] a model based RL algorithm was employed for the task of controlling a helicopter. In [18], the helicopter learned to however, and by [19], this work was extended to carrying out inverted flight. The algorithm uses LWR method to learn the transition model which consisted of a 12 dimensional state space and a 4 dimensional action space. They used the PEGASUS algorithm then for policy evaluation and updating.

In [18], using 30 Monte Carlo steps, the authors were able to make the helicopter (a remote controlled model of course) hover better than a trained human pilot could in terms of maintaining position and velocity.

1.4.3 Miscellaneous methods

These works aren't as important to the present work as those listed previously; they were however too interesting to not mention, and present some other faces of reinforcement learning that was not covered in this review.

From Pixels to Torques

In [20], the authors describe a model based reinforcement learning method that learns a policy from pixel information of images. The main objective of this work was to present a way to translate information from images directly to torque inputs. Instead of the agent being able to directly gleam the states from the environment; it can only do so using high dimensional information from pixels of images.

In this work, they use deep auto-encoders to learn a low-dimensional embedding of images jointly with a predictive model in this low dimensional space. The term 'deep' implies that there are several layers of structure in the models. They use nonlinear model predictive control (MPC) to learn closed loop policies. Using these, the proposed algorithm reached a 90% success rate after 1500 frames, while the benchmarking PILCO framework reached

a 100% success rater after approximately 200-300 frames.

Natural Actor-Critic

Actor-Critic methods attempt to bring together advantageous features of the value function methods and policy search methods. They are named as thus because policy search methods can be thought of as being *actors*, as they only worry about *how to do something well*, while value function methods wait for the actor do something, and then *critiques* its work by giving its actions value [4]. In simple high level terms, the critic (value function) in actor-critic algorithms observes the performance of the actor (the policy), and decides when it needs to updated.

In [21, 22] a novel algorithm dubbed the Natural actor-critic was proposed. This was put into practice in application of humanoid robots in [21]. In this method, the actor is updated based on stochastic policy gradients, while the critic uses both natural policy gradients and additional parameters of the value function at the same time through linear regression [22]. Natural gradients differ from conventional gradients in that they do not follow the steepest direction in the relevant space, but rather follow the steepest direction as given by the Fisher metric [21, 22]. This supposedly has the advantage of not getting stuck in plateaus of the space. It was found that the natural critic-actor method takes about 200 steps to find the optimal policy.

Bibliography

[1] S. Rogers and M. Girolami, *A First Course in Machine Learning*. Taylor and Francis Group, 2012.

[2] T. M. Mitchell, *Machine Learning*. Mcgraw-Hill, 1997.

[3] N. D. Lawrence, "Probabilistic non-linear principal component analysis with gaussian process latent variable models," *JMLR*, vol. 6, no. 1783-1816, 2005.

[4] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, September 2013.

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

[6] M. P. Deisenroth, "Efficient reinforcement learning using gaussian processes," Ph.D. dissertation, Karlsruhe Institute of Technology, November 2010.

[7] M. Kaiser, "Incorporating uncertainty into reinforcement learning through gaussian processes," Master's thesis, Technische Universität München, June 2016.

[8] M. P. Deisenroth, C. E. Rasmussen, and J. Peters, "Gaussian process dynamic programming," *Neurocomputing*, vol. 72, pp. 1508–1524, 2009.

[9] R. Tedrake, *Underactuated Robotics*, Lecture Notes, MIT, 2009.

[10] M. P. Deisenroth, J. Peters, and C. E. Rasmussen, "Model-based reinforcement learning with continuous states and actions," *Proceedings for the 16th European Symposium on Artificial Neural Networks*, April 2008.

[11] 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, August 2013.

[12] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural computation*, vol. 3.2, pp. 246–257, 1991.

[13] ——, "Approximation and radial-basis-function networks," *Neural computation*, vol. 5.2, pp. 305–316, 1993.

[14] C. Atkeson, A. Moore, and S. Schaal, "Locally weighted learning," *Artif Intell Rev*, vol. 11.1, no. 5, pp. 11–73, 1997.

[15] J. Ko, D. J. Klein, D. Fox, and D. Haehnel, "Gaussian process and reinforcement learning for identification and control of an autonomous blimp," Rome, Italy, 2007, pp. 10–14.

[16] A. Y. Ng and M. Jordan, "Pegasus: A policy search method for large mdps and pomdps," *Proceedings for the 16th conference on Uncertainty in artificial intelligence*, pp. 406–415, 2000.

[17] M. P. Deisenroth, D. Fox, and C. E. Rasmussen, "Gaussian processes for data-efficient learning in robotics and control," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 2, February 2015.

[18] H. Kim, M. I. Jordan, S. Sastry, and A. Y. Ng, "Autonomous helicopter flight via reinforcement learning," in *Advances in neural information processing systems*, 2003, p. None.

[19] 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," in *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[20] M. P. D. Niklas Wahlstrom, Thomas B. Schon, "From pixels to torques: Policy learning with deep dynamical models," *arXiv preprint*, vol. arXiv:1502.02251v3, 2015.

[21] J. Peter, S. Vijayakumar, and S. Schaal, "Reinforcement learning for humanoid robotics," in *Third IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, Germany, September 2003.

[22] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *European Conference on Machine Learning*. Springer, 2005, pp. 280–291.