Adaptive Behavior
1–15
© The Author(s) 2016
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1059712316664570
adb.sagepub.com
SAGE

Transfer learning by prototype generation in continuous spaces

Enrique Munoz de Cote, Esteban O Garcia and Eduardo F Morales

Abstract
In machine learning, learning a task is expensive (many training samples are needed) and it is therefore of general interest to be able to reuse knowledge across tasks. This is the case in aerial robotics applications, where an autonomous aerial robot cannot interact with the environment hazard free. Prototype generation is a well known technique commonly used in supervised learning to help reduce the number of samples needed to learn a task. However, little is known about how such techniques can be used in a reinforcement learning task. In this work we propose an algorithm that, in order to learn a new (target) task, first generates new samples—prototypes—based on samples acquired previously in a known (source) task. The proposed approach uses Gaussian processes to learn a continuous multidimensional transition function, rendering the method capable of reasoning directly in continuous (states and actions) domains. We base the prototype generation on a careful selection of a subset of samples from the source task (based on known filtering techniques) and transforming such samples using the (little) knowledge acquired in the target task. Our experimental evidence gathered in known reinforcement learning benchmark tasks, as well as a challenging quadcopter to helicopter transfer task, suggests that prototype generation is feasible and, furthermore, that the filtering technique used is not as important as a correct transformation model.

Keywords
Transfer learning, reinforcement learning, Gaussian processes, prototype generation

1 Introduction

Autonomy in robots—and machines in a broader sense—is reaching levels where operation knowledge will no longer be required. Even when this makes final users' life easier, it means that robots must have higher adaptability to changing environments and must learn new tasks with a small number of interactions. Reinforcement learning (RL) is a common learning technique used in robotics (among many other fields), where the robot can discover the solution to a task while it is directly interacting with its environment and receiving rewards as feedback about the desirability of the states and actions. However, one of the most limiting requirements of most RL algorithms (e.g. Q-learning, SARSA or R-max) is their need to collect a large quantity of data (experience) in order to learn an optimal policy. This requirement is often what prevents RL techniques from being deployed in real systems, like in real robotic applications.[1] Several techniques tackle this problem from different angles: using abstract representations, like functional or relational learning (Bertsekas & Tsitsiklis, 1996; van Otterlo, 2009), updating several state-action values at a time, like eligibility traces (Jaakola, Jordan, & Singh, 1995), or incorporating additional guidance or information, like reward shaping (Ng, Harada, & Russell, 1999). In this paper, we propose to approximate a Markov decision process (MDP)'s transition function directly in the continuous space domain while learning and present a transfer learning (TL) technique that reduces the number of training instances needed to learn a correct control policy.

In more detail, TL is a machine learning technique whose objective is to reduce the number of interactions required to learn an optimal policy by sharing experience (knowledge) between related tasks. Needless to say, it is of utmost importance to identify common features between related tasks so that sharing knowledge is valuable in the target task. As an illustrative example,

Computer Science Department, Instituto Nacional de Astrófisica, Óptica y Electrónica, Mexico

Corresponding author:
Enrique Munoz de Cote, Instituto Nacional de Astrófisica, Óptica y Electrónica, Computer Science Department, Luis Enrique Erro 1, Tonantzintla, Puebla, México C.P. 72840.
Email: jemc@inaoep.mx

imagine two rivers with the same dimensions and water density but different constant flows. Now imagine a boat that learns to cross the first river from point "a" to point "b" and that needs to cross (still from point "a" to point "b") the second river. We can intuitively say both *tasks* are related,[2] and that some knowledge can be reused (with probably only needing to apply some scalar transformation along the way) to learn to cross the second river with fewer instances.

There are a number of shortcomings that have not been fully addressed in the literature when solving this kind of task. One is the lack of studies that work directly on continuous domains (without needing a discretization process), which is of special interest in control problems to avoid bang-bang controllers that reduce the life of machines, or in financial, earth science and medical applications where precision cannot be rounded to the nearest integer. Another shortcoming is with respect to the type of knowledge being transferred. In fact, most approaches focus on transferring experience instances[3,4] (Lazaric, Restelli, & Bonarini, 2008; Taylor, Jong, & Stone, 2008). However, this could be detrimental if optimal policies change dramatically between tasks or if instances in a source task are not relevant in the target task. In this work we propose learning a model of the target task by first transferring relevant information (instances) from a source task to the target task. This knowledge transfer is fulfilled by first *selecting* the instances that are considered most relevant and then *synthesizing* new instances. Such artificially generated instances are known in the literature as prototypes and, for the RL algorithm we will describe later on, instances and prototypes are indistinguishable (as is also the case in the supervised learning setting).

It is worth noting that the generated prototypes should be as much informative as possible so that fewer instances and prototypes are needed to correctly learn a task. Also, this technique will allow the agent to have better prior information when starting than that obtained by just transferring raw information across tasks, which the agent will utilize as a jump-start in the learning process. With this in mind, the probability of negative transfer will be lessened. As the agent progresses in learning the dynamics of the target task, the transferred information is progressively reduced to avoid negative transfer and promote learning an optimal policy.

The experiments presented on well known RL control benchmarks (cart-pole and mountain car) show significant improvements over state-of-the-art algorithms in two objective measures: the total accumulated reward while learning and the number of episodes required to learn. We also present experiments in the more realistic task of transferring knowledge from a quadcopter to a helicopter and show that it is possible to transfer knowledge between high-dimensional continuous challenging tasks. The structure of this paper is as follows. In Section 2, we give an overview of the most closely related work in TL. Section 3 briefly introduces RL, Gaussian processes and how Gaussian processes can be used to represent state transition functions. In Section 4, the proposed transfer method is described in detail. Section 5 presents experimental results in several relevant problems for RL. Section 6 summarizes the paper and proposes future research directions.

2 Related work

TL refers to the problem of retaining and applying the knowledge learned in one or more tasks to efficiently develop an effective hypothesis for a new task (Silver et al., 2005). Most of the early research was performed for the supervised learning paradigm and it is only recently that there has been increasing attention on the RL paradigm. One of the first approaches is described by Atkeson and Santamaria (1997) while comparing model-based and model-free RL algorithms, where they use a locally weighted regression model of the pendulum dynamics to infer reward function structure across tasks that only differ in their reward function. Our approach is partly built on this work, where, as they do, we learn a regression model. However, here we propose using Gaussian processes to learn a model of the probability transition function and furthermore train another Gaussian process to synthesize prototypes based on source task instances.

Other more recent methods commence from the idea that the source task can be decomposed into a set of subtasks that might be also found in the target task (Drummond, 2002; Mehta, Natarajan, Tadepalli, & Fern, 2008). This is the case of the MAX-Q hierarchical decompositions (Dietterich, 2000) that even if interesting, all these approaches only work in highly (hierarchically) structured domains. Other methods focus on problems where the source and target tasks have different state representations, but can still be related through an *inter-task mapping* (Ammar, Tuyls, & Taylor, 2012; Taylor et al., 2008; Taylor, Stone, & Liu, 2007). Inter-task mapping is related to this work but tackles a relatively different problem. Whereas inter-task mapping focuses on finding mappings across domain representations and tasks so that source instances can be directly mapped as target instances, here we focus on transferring a subset of instances and generating prototypes. Lastly, most of the mentioned related approaches have been developed to face discrete problems or, in the best case, to deal with continuous states and discrete actions. In this work, we focus on a challenging scenario related to real world tasks, where the variables that describe the state and actions are continuous and we assume that the problem can not be discretized. In this scenario, all other TL approaches are infeasible, at least without significant adaptations.

To our knowledge, the only exceptions that learn tasks in a continuous domain RL framework and do so without discretization methods are those in (Deisenroth & Rasmussen, 2011; Hasselt, 2011; Lazaric, Restelli, & Bonarini, 2007; Martín H, de Lope, & Maravall, 2011; Ng, Kim, Jordan, & Sastry, 2004; van Hasselt, 2011). However, none of these study the problem of transferring knowledge across domains.

A different vein of research pursues the problem of TL in RL by transferring policies across domains. In Torrey, Walker, Shavlik, and Maclin (2005), the source task Q-function[5] is used to generate recommended actions in the target task. In the target task, such recommended actions have more probability of being executed. This heuristic is useful for tasks where the actions are discrete. However in tasks like the ones considered in this work, this sort of method is infeasible, because discretizing the actions for fine control can lead to unmanageable amounts of information. Soni and Singh (2006) look for relationships between state-action pairs of both the source and target tasks. Using these relationships they search for skills that the agent performs (called *options*) by grouping sequential transitions so they can be used as preferred traces within specific states on the target task. These options can be discovered only if action sequences (i.e. traces) can be identified. However, in continuous tasks it is almost impossible to execute the same action twice in the same learning process, so sequences of actions would all be different and would be hardly useful.

The closest line of work to this paper is the works of Lazaric (2008), Lazaric et al. (2008) and Taylor et al. (2008). Like us, they propose transferring instances composed by $\langle$*state, action, reward, next state*$\rangle$ from the source task to the target task. Lazaric (2008) does so by following similarity measures between tasks that are based on some distance metric. The actions in these works are discrete and are used as indexes to cluster instances together. This type of TL can be thought of like a filtering method that allows only instances that are thought to be relevant to the target task to be transferred. In this work we use a filtering function to select subsets of relevant features, but then we transform such subsets to prototypes in the target task. We use several different filtering functions (see Section 4.3), one such function is Lazaric's technique presented here. As such, we compare this technique with a simpler filtering technique, showing that the filtering technique used is actually not very relevant to the problem of generating relevant prototypes.

When working directly with continuous spaces, most of the published work is related to the use of function approximation techniques. Relevant to our approach, Gaussian processes have been used before to represent value functions (Deisenroth, Peters, & Rasmussen, 2008; Engel, Mannor, & Meir, 2003, 2005; Rasmussen & Kuss, 2004) and, more recently, to represent transition function models with very promising results (Deisenroth & Rasmussen, 2011; Deisenroth, Rasmussen, & Fox, 2011; Deisenroth, Rasmussen, & Peters, 2008; Murray-Smith & Sbarbaro, 2002; Rasmussen & Deisenroth, 2008).

In Bou Ammar, Eaton, Ruvolo, and Taylor (2015), the authors describe an unsupervised manifold alignment to learn an inter-state mapping to transfer samples between task domains. The mapping learned is between states, while our approach learns a model of differences between probability transition functions. We are directly learning a new transition function with our synthetic samples, while they map from states to trajectories and back to states to learn a new policy. Both approaches are not really comparable; our approach learns with about an order of magnitude of fewer episodes (they reported 300 episodes for the cart-pole and over 3000 for the quadcopter) as will be seen in Section 5. On the other hand, their approach is able to map between different tasks while ours is restricted to transfer between domains with the same state and action spaces.

Garcia, Munoz de Cote, and Morales (2013) use Gaussian processes to learn a task's transition function and study how hyper-parameters—which give information about the form of the function—can be used as a means to transfer qualitative information across tasks. In Garcia, Munoz de Cote, and Morales (2014), the authors extended the previous idea by using a new Bayesian rule for combining hyper-parameters across the source and target tasks that take into account the uncertainty in the new task data. These works are similar to this paper in that they all use Gaussian processes to work directly on continuous state and action domains, but differ in that here we focus on generating new instances based on subgroups of instances in related tasks, whereas Garcia et al. (2013) and Gracia et al. (2014) focus on transferring qualitative information in the form of hyper-parameters.

Against this background, this work is inspired by recent advances in knowledge representation techniques, specifically in the work by Triguero, Derrac, Garcia, and Herrera (2012), that generate new informative prototypes for classification tasks. We present a new method called "Trapper Keeper" (TRAnsfer PrototyPE geneRation) to generate, contrary to previous approaches, new prototypes to learn (with fewer instances) the target task.

3 Background

RL refers to the problem of learning how to interact with an environment in order to maximize the expected sum of rewards. RL can be described by an MDP (Puterman, 1994), defined by $\langle S, A, P, R \rangle$, where S is the set of states, A is the set of possible actions that the agent may execute, $P : S \times A \times S \rightarrow [0, 1]$ is the probabilistic state transition function, $R : S \times A \rightarrow \mathbb{R}$ is the reward function that defines the goal and measures the desirability of each state and a policy $\pi : S \rightarrow A$ is a function that maps states to actions (see Sutton & Barto, 1998, for more details). In this work we define "task" as an MDP, so "task"and "MDP" will be used interchangeably. Furthermore, in the case of continuous domains, $S = \mathbb{R}^D$ and $A = \mathbb{R}^F$, where D and F are the dimensions of the state and action features vector respectively. Function approximators can be used to represent the state transition function P and the policy function π. In this work, we use Gaussian processes to represent the state transition function.

A Gaussian process is a generalization of the multivariate Gaussian probability distribution. It is often denoted by $\mathcal{GP}(m, k)$ and specified by a mean function $m(\cdot)$ and a covariance function $k(\cdot, \cdot)$, also called a *kernel*. Given a set of input vectors $\mathbf{x_i}$ arranged as a matrix $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x_n}]$ and a vector of samples or training observations $\mathbf{y} = [y_1, \ldots, y_n]^{\mathrm{T}}$, Gaussian process methods, for regression problems, assume that the observations are generated as $y_i = h(\mathbf{x_i}) + \epsilon$, where $h(\mathbf{x_i})$ follows a multivariate Gaussian distribution and ϵ is additive noise that follows an independent and identically distributed Gaussian distribution with zero mean and variance σ_ϵ^2 $(\epsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2))$.

Given a Gaussian process model of the *latent function* $h \sim \mathcal{GP}(m, k)$, it is possible to predict function values for an arbitrary input $\mathbf{x}_*$. The predictive distribution of the function value $h_* = h(\mathbf{x}_*)$ for a test input $\mathbf{x}_*$ is Gaussian distributed with mean and variance given by

$$E_h[h_*] = k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_\epsilon^2 \mathbf{I})^{-1}\mathbf{y} \tag{1}$$

$$\mathrm{var}_h[h_*] = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_\epsilon^2 \mathbf{I})^{-1} k(\mathbf{X}, \mathbf{x}_*) \tag{2}$$

where $\mathbf{K} \in \mathbb{R}^{n \times n}$ is the kernel matrix with $K_{ij} = k(\mathbf{x_i}, \mathbf{x_j})$.

The covariance function k commonly used is the squared exponential kernel

$$k(\mathbf{x}, \mathbf{x}') = \alpha^2 \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^{\mathrm{T}} \mathbf{\Lambda}^{-1} (\mathbf{x} - \mathbf{x}')\right) \tag{3}$$

with $\mathbf{\Lambda} = \mathrm{diag}([\ell_1^2, \ldots, \ell_n^2])$ and ℓ_k for $k = 1, \ldots, n$, being the characteristic length scales and σ_ϵ^2 the noise term. The parameter α^2 describes the variability of the latent function h. The parameters of the covariance function or hyper-parameters of the Gaussian process $(\alpha^2, \ell, \sigma_\epsilon^2)$ are collected within the vector θ. The hyper-parameters define the shape of the functions in the prior distribution. The kernel hyper-parameters are often optimized to adjust the prior Gaussian distribution to the data, using evidence maximization (see Rasmussen & Williams, 2006, for more details on Gaussian processes and evidence maximization).

In what follows we will describe a new way to transfer knowledge between tasks that involves learning both transition and policy functions and a model of differences between probability transition functions (in the source and target tasks)—all of these through Gaussian processes.

4 Transfer learning by synthesis of samples

Defining models for dynamic systems is not an easy task and requires expert knowledge. The RL paradigm has been used to learn such models and is becoming increasingly popular. This is due to the fact that it does not require a predefined model of the dynamic system. In the traditional RL paradigm, the agent learns the system dynamics and a control policy (given a goal) while exploring the environment, and the policy converges to the (near) optimal policy in the limit. However, this learning process is slow, usually requires a discrete representation and once a policy is learned it cannot be used for a new task, even if it is similar to the original task.

TL has been used as an option to reduce the learning time (i.e. the number of instances needed to learn a task) by reusing knowledge acquired in related tasks. There are several valid definitions of TL for RL tasks.[6] In this work we focus on a *dynamics transfer problem* as defined in Lazaric (2008, def. 3.3).

Definition 1. Dynamics transfer problem. *A dynamics transfer problem is one in which the source and target tasks share the same context (i.e. state and action space) and the same reward function. The objective is to transfer relevant knowledge to improve an RL algorithm in terms of its learning speed, accumulated reward and performance.*

The three objectives in the dynamics transfer problem definition can be defined as follows.

Performance (P). This is the value at convergence. Let r_i denote the reward of episode i, and then

$$P = \frac{r_T + r_{T-1} + r_{T-2}}{3} \tag{4}$$

Learning speed (T). The number of sample instances needed for an RL algorithm to learn its task. In practice, this is calculated as the elapsed number of episodes before reaching 95% of its final performance

$$\min_t \{t \in \{1, \ldots, T\} | r_i \geq P0.95\} \tag{5}$$

Accumulated reward (R). Sum of rewards over all episodes

$$R = \sum_{i=1}^{T} r_i \tag{6}$$

Within the dynamics transfer problem we assume the source and target tasks are represented by continuous state and action spaces, and such spaces are shared among the tasks. Within this problem, there are different kinds of knowledge that can be transferred across tasks: policies, value functions, instances or structural knowledge (e.g. hyper-parameters). In this work, we transfer two types of knowledge from source to target tasks: policies and the use of state transition instances to learn transformations that generate new prototypes for the target task. We do policy transfer in a quite trivial way and it is not central to this paper. We do so to jump-start the learning process domain (this is important in robotics tasks as in the quadcopter–helicopter) as learning from scratch will crash the vehicle before learning.

In detail, given state transition instances and a state transition function from the source task, the steps followed by Trapper Keeper are:

- apply the current policy (initially the final policy from the source task) in the target task to generate state transition instances of the first episode;
- use these instances to learn a probabilistic state transition function for the target task (see Section 4.1);
- learn a function that represents the difference between the target and the source state transition functions (see Section 4.2);
- select a set of state transition instances from the source task (see Section 4.3);
- generate new state transition prototypes for the target task using the selected instances and the difference function learned in the previous step;
- use the generated prototypes and the original instances from the target task to learn a new state transition function and use it to learn a new policy.

These steps are repeated until convergence. In the following subsections these steps are described in more detail.

4.1 Model estimation

For RL, we use model-based learning, which unlike model-free learning, learns a model of the task (i.e. its transition and reward function) before putting it into action in the real task. Therefore, the effectiveness of the learned policy highly depends on the precision of the task model. In our work, the agent models the task by estimating the transition function, which is then used to learn the policy.

When the agent is in the initial learning episodes, there is not enough information for a good assessment of the task, and therefore the executed policy is suboptimal. In our proposed approach (Algorithm 1), we first use the learned policy to initialize the policy for the target task. The source policy will (in most circumstances) not solve the target task (Garcia et al., 2013; Taylor et al., 2008); however, it usually helps jump-start the initial performance, and is usually enough to collect initial samples of the target task.[7]

The state transition function is learned as a non-parametric Gaussian process using available data, going from a prior distribution of transition functions to a posterior one. The learned Gaussian process model internally represents the dynamics in the system (i.e. the transition function). The learned transition model is then used to simulate the system and reason for the long-term behavior without the need of interaction (batch learning). The policy can be represented with any approximator, but in this work we decided to use the approach proposed by Deisenroth and Rasmussen (2011) where a policy is computed by using estimates of the gradient of the value function according to the simulations and after optimizing this policy. This optimized policy is then used to interact with the environment to obtain more instances (state, action, successor state) and continue this process in a loop until some stopping criterion (e.g. change in value function between iterations is small enough) is met.

More precisely, the state transition function is modeled as a Gaussian process, where the next state is defined as $\mathbf{x_t} = f(\mathbf{x}_{t-1}, \mathbf{a}_{t-1}), f \sim \mathcal{GP}(m, k)$, where $\mathbf{x}_t \in S$ is the state of the agent at time t, and is approximated by function f. The transition model f is distributed as a Gaussian process with mean function m and covariance function k. The samples of the task $(\mathbf{x}_{t-1}, \mathbf{a}_{t-1}) \in \mathbb{R}^{D+F}$ and the corresponding $\Delta_t = \mathbf{x}_t - \mathbf{x}_{t-1} + \epsilon \in \mathbb{R}^D$, $\epsilon \sim \mathcal{N}(0, \Sigma_\epsilon)$, are the training targets of the latent function f. Δ_t is used instead of $\mathbf{x}_t$ as the differences vary less than the original function and learning the differences is better than learning the function values directly.

The objective in RL is to find a policy π: $S \mapsto A$ that maximizes the expected accumulated reward

$$V^{\pi}(\mathbf{x}_0) = \sum_{t=0}^{T} \mathbb{E}[R(\mathbf{x}_t)], \mathbf{x}_0 \sim \mathcal{N}(\mu_0, \Sigma_0) \tag{7}$$

which is the sum of the expected rewards $R(\mathbf{x}_t)$ obtained from a trace $(\mathbf{x}_0, \ldots, \mathbf{x}_T)$, T steps ahead. Here π is a continuous function approximated by $\tilde{\pi}$, using a set of parameters ψ. For most continuous tasks, it is useful to use a saturating reward function $R(\mathbf{x}_t) = \exp(-d^2/\sigma_r^2)$ that rewards when the Euclidean distance d of the current state $\mathbf{x}_t$ to the target state $\mathbf{x}_{\text{target}}$ is small enough, while σ_r^2 controls the width of R.

We use policy search, as in Deisenroth and Rasmussen (2011), to find the policy $\tilde{\pi}$, which is approximated by a radial basis function network with Gaussian basis functions given by

$$\tilde{\pi}(\mathbf{x}_*) = \sum_{s=1}^{N} \beta_s k_\pi(\mathbf{x}_s, \mathbf{x}_*) = \beta_\pi^{\mathrm{T}} k_\pi(\mathbf{X}_\pi, \mathbf{x}_*) \qquad (8)$$

where $\mathbf{x}_*$ is a test input, k_π is the squared exponential kernel and $\beta_\pi = (\mathbf{K}_\pi + \sigma_\pi^2 \mathbf{I})^{-1}\mathbf{y}_\pi$ is a weight vector. $\mathbf{K}_\pi$ is formed as $(K_\pi)_{ij} = k_\pi(\mathbf{x}_i, \mathbf{x}_j)$, where $\mathbf{y}_\pi = \tilde{\pi}(\mathbf{X}_\pi) + \epsilon_\pi$, $(\epsilon_\pi \sim \mathcal{N}(\mathbf{0}, \sigma_\pi^2 \mathbf{I}))$ represent the training targets for the policy, with ε_π measuring noise. $\mathbf{X}_\pi = [\mathbf{x}_1, \ldots, \mathbf{x}_N]$, $\mathbf{x}_s \in \mathbb{R}^D$, $s = 1, \ldots, N$, are the training inputs. The support points $\mathbf{X}_\pi$ and the corresponding training targets $\mathbf{y}_\pi$ are a *pseudo-training set* for the preliminary policy, this pseudo-training set will be later replaced by actual target training data as the agent collects such information.

Once the task model has been learned, we use PILCO's algorithm to search for the policy on the learned model. Our proposed approach is not constrained to work with PILCO, as other policy search algorithms could be used. However, PILCO is an algorithm that has shown a good performance in domains with continuous states and actions. Besides, the main emphasis of the paper is not on the way to learn a probabilistic transition function or the function policy, but on the proposed technique for TL in continuous domains.

4.2 A model of differences

We use two Gaussian processes to estimate the source and the target task transition models (see Algorithm 2). Let $\tilde{\mathbf{x}}_{\text{source}} = [\mathbf{x}^{\mathrm{T}}\mathbf{a}^{\mathrm{T}}]^{\mathrm{T}}, \tilde{\mathbf{x}}_{\text{source}} \in \mathbb{R}^{D+F}$ denote a sample from the source task, where D is the dimension of the state vector $\mathbf{x}$ and F is the size of the action vector $\mathbf{a}$. Let $\tilde{\mathbf{y}}_{\text{source}} \in \mathbb{R}^D$ denote an output of the latent transition function f_{source}. In the same way, the samples from the target task, are denoted by $\tilde{\mathbf{x}}_{\text{target}} \in \mathbb{R}^{D+F}$ and their corresponding output $\tilde{\mathbf{y}}_{\text{target}} \in \mathbb{R}^D$, to specify the transition function f_{target}.

Trapper Keeper starts with the policy from the source task to get samples for the first episode. In the first interaction with the environment, instances of the form $\langle \mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}' \rangle$ are collected, where $\tilde{\mathbf{x}}_{\text{source}} = [\mathbf{s}^{\mathrm{T}}\mathbf{a}^{\mathrm{T}}]^{\mathrm{T}}$, $\tilde{\mathbf{y}}_{\text{source}} = \mathbf{s}'$ are taken, as shown in lines 1 and 2 of Algorithm 1. For the target task, new prototypes are generated by taking a subset of instances of the source task and the target task instances.

The generation of new prototypes is performed by *genPrototypes*() (Algorithm 2). The crux of this function is to learn a model of differences between the source and target tasks' state transition functions. The learned model is an estimator of how different the task's functions are. This function is then used to transform source instances into target prototypes. In more detail, two Gaussian processes, one for the source task transition function f_{source} and one for the target task transition function f_{target}, are learned using only the available samples in each task (lines 2 and 3). A third Gaussian process model, f_τ is learned as an estimator of the difference between the first two models (lines 5–7), using instances selected by a *filter function* (line 4). As Taylor et al. (2008), Lazaric et al. (2008), Garcia et al. (2013), Torrey et al. (2005), Ferguson and Mahadevan (2006) and other authors have already stated, not all source task instances are good candidate instances to be transferred to a related task. Against this background we use a function *filter*() to select instances that we believe are most relevant to be transferred. Once a subset of instances from the source task is selected, it is transformed into a set of synthetic prototypes for the target task using f_τ. There are different approaches in the literature to select the most relevant instances to be transferred. Some of the most relevant approaches are presented next.

Algorithm 1: Trapper Keeper

```
Require: $\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}, \psi_{\text{source}}$
 1: $\tilde{\pi} \leftarrow \pi(\psi_{\text{source}})$
 2: Interact with target environment, apply $\tilde{\pi}$ to obtain samples
    of the form $\tilde{x}_{\text{target}}, \tilde{y}_{\text{target}}$
 3: repeat
 4:    $\tilde{\mathbf{X}}_\tau, \tilde{\mathbf{Y}}_\tau \leftarrow$ genPrototypes($\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}, \tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_{\text{target}}$)
 5:    // learn a Gaussian process $f_v$ for each variable $v$ from the
       state vector
 6:    Use $\tilde{\mathbf{X}}_\tau \cup \tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_\tau \cup \tilde{\mathbf{Y}}_{\text{target}}$ to learn task model $f$
 7:    repeat
 8:       Evaluate policy $\tilde{\pi}$ on $f$ to get $V^{\tilde{\pi}}$
 9:       Improve $\tilde{\pi}$ //updating parameters $\psi$ using PILCO
10:    until convergence
11:    $\tilde{\pi} \leftarrow \pi(\psi)$
12:    Interact with environment applying $\tilde{\pi}$ to obtain more
       samples
13: until task learned
```

Algorithm 2: Prototype Generation

```
 1: function GENPROTOTYPES ($\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}, \tilde{\mathbf{X}}_{\text{target}},$
    $\tilde{\mathbf{Y}}_{\text{target}}$)
 2:    Learn Gaussian process $f_{\text{source}}$, using $\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}$
 3:    Learn Gaussian process $f_{\text{target}}$, using $\tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_{\text{target}}$
 4:    $\tilde{\mathbf{X}}_\tau \leftarrow$ filter($\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}, \tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_{\text{target}}$)
 5:    for all $\tilde{\mathbf{x}} \in \tilde{\mathbf{X}}_\tau$ do
 6:       $y_d = f_{\text{source}}(\tilde{\mathbf{x}}) - f_{\text{target}}(\tilde{\mathbf{x}})$
 7:       $\mathbf{Y}_d = \mathbf{Y}_d \cup \{y_d\}$
 8:    end for
 9:    Learn Gaussian process $f_\tau$, using $\tilde{\mathbf{X}}_\tau, \tilde{\mathbf{Y}}_d$
10:    for all $\tilde{\mathbf{x}} \in \tilde{\mathbf{X}}_\tau$ do
11:       $y_\tau = f_{\text{source}}(\tilde{\mathbf{x}}) - f_\tau(\tilde{\mathbf{x}})$
12:       $\mathbf{Y}_\tau = \mathbf{Y}_\tau \cup \{y_\tau\}$
13:    end for
14: return $\tilde{\mathbf{X}}_\tau, \mathbf{Y}_\tau$
15: end function
```

Algorithm 3 Naïve filter

```
1: function NaïveFilter($\tilde{\mathbf{X}}_{\text{source}}$, $\tilde{\mathbf{Y}}_{\text{source}}$, $\tilde{\mathbf{X}}_{\text{target}}$, $\tilde{\mathbf{Y}}_{\text{target}}$))
2:     Initialize $d$ as an $m \times n$ matrix, where $m = |\tilde{\mathbf{X}}_{\text{source}}|$ and
       $n = |\tilde{\mathbf{X}}_{\text{target}}|$
3:     $d_{ij} \leftarrow$ distance($\tilde{x}_i, \tilde{x}_j$), $\forall i \in \tilde{\mathbf{X}}_{\text{source}}$, $\forall j \in \tilde{\mathbf{X}}_{\text{target}}$
4:     $d_i \leftarrow \min_j d_{ij}$, $\forall i \in \tilde{\mathbf{X}}_{\text{source}}$, $\forall j \in \tilde{\mathbf{X}}_{\text{target}}$
5:     Sort $d_i$ rows in ascending order
6:     $\tilde{\mathbf{X}}_\tau \leftarrow$ last $k$ elements in $d$ (where $k = m - n$)
7: return $\tilde{\mathbf{X}}_\tau$
8: end function
```

Algorithm 4 Lazaric's filter

```
1: procedure LAZARIC'S FILTER($\tilde{\mathbf{X}}_{\text{source}}$, $\tilde{\mathbf{Y}}_{\text{source}}$, $\tilde{\mathbf{X}}_{\text{target}}$,
   $\tilde{\mathbf{Y}}_{\text{target}}$)
2:     Let $m = |\tilde{\mathbf{X}}_{\text{source}}|$, $n = |\tilde{\mathbf{X}}_{\text{target}}|$ and $k = m - n$
3:     $\Delta_k \leftarrow$ compliance($\{\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}\}, \{\tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_{\text{target}}\}$)
4:     for all $\{\tilde{\mathbf{x}}_j, \tilde{\mathbf{y}}_j\} \in \{\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}\}$ do
5:         $\rho_j$ = relevance($\{\tilde{\mathbf{x}}_j, \tilde{\mathbf{y}}_j\}, \{\tilde{\mathbf{X}}_{\text{target}}, \tilde{\mathbf{Y}}_{\text{target}}\}$)
6:     end for
7:     $\tilde{\mathbf{X}}_\tau \leftarrow k\Delta_k$ Samples taken from $\{\tilde{\mathbf{X}}_{\text{source}}, \tilde{\mathbf{Y}}_{\text{source}}\}$
       proportionally to $\rho_j$
8: return $\tilde{\mathbf{X}}_\tau$
end procedure
```

4.3 Filtering

Not all sampled instances from the source task are useful for learning the target task. Many approaches in the literature focus on the problem of selecting which instances are most relevant to be transferred between tasks. The function *filter*() used in Algorithm 2 is a call to any algorithm that filters out irrelevant instances. We propose a simple filtering technique based on the idea that the most relevant instances from the source task are those far (in terms of distance) from the experienced instances in the target task. The rationale behind this simple idea is that including instances from the source task that are close to instances from the target task will not provide new information, as we already have instances in that region from the target, and may introduce noise to the model. We call such filter the *naïve filter*. Although we propose this naive filter, more sophisticated filters might be used as well, but as will be shown in the experimental section, we claim the sophistication of the filtering technique to be of minor importance.

In more detail, let us assume that we have m and n instances from a source and target task respectively, where $m \gg n$ and that m instances are enough to solve (learn) the source task. We make the initial assumption that the target task will require close to m instances to be learned. If such assumption is true the agent would require $m - n$ more instances for the target task to be learned. We propose to initially generate $m - n$ prototypes so that together with the n instances already collected, the agent can approximate a solution to the target task. Furthermore, as the agent gathers new samples, prototypes are gradually replaced by true instances obtained directly in the target task, maintaining always m instances for the target task. This assumption was designed as an upper limit to the number of generated synthetic samples, but we are aware that it is also placing a limit to the number of samples needed for the target task. Without any additional knowledge about the target task, it is not possible to know in advance the number of samples needed to learn any model.[8] An obvious and simple extension is to remove this constraint over the target task and allow its number of samples to be increased when needed. In our experiments we did not need to increase the number of samples for the target task and it is left as future work.

We compare two filtering algorithms. The first one (the naïve filter in Algorithm 3) is a simple filter that considers relevant the most distant instances, where a Euclidean distance is used between state-action pairs of the source and target tasks. Note that line 4 of the algorithm finds the closest neighbor $j \in \tilde{\mathbf{X}}_{\text{target}}$ to each source task $i \in \tilde{\mathbf{X}}_{\text{source}}$.

The second approach is based on Lazaric's TL algorithm. The algorithm was designed for tasks described by continuous states and discrete actions. We modified the algorithm to work with continuous actions (see Algorithm 4). In their approach, the tasks are restricted to having discrete actions and instances are clustered according to the selected actions. Instead of clustering, the action variables are considered as variables of state-action pairs and taken into account for the *compliance* and *relevance* metrics (see Lazaric et al., 2008, for more details).

Task compliance measures the probability of $\hat{f}$ being the model that generated the instances $\tilde{\mathbf{X}}_{\text{target}}$, as described in equation (9) (for more detail on *task compliance* please refer to Lazaric et al., 2008)

$$\Lambda_{\text{compl}} = \frac{1}{|\tilde{\mathbf{X}}_{\text{target}}|} \sum_i P(\hat{f} | \tilde{\mathbf{x}}_{i,\,\text{target}}) \tag{9}$$

The metric *task compliance* uses the available instances in the source task as well as the ones in the target task and returns the probability that the source task has generated the target task's instances. The *sample relevance* metric is calculated for each of the samples, using the *compliance* previously calculated. ρ_j indicates the percentage of samples taken from the source task, ordered according to *relevance*.

5 Experiments

The experiments we present are designed to show how our proposed technique, embedded in the Trapper

Keeper algorithm fares. We perform tests in three tasks and measure performance, convergence time[9] and accumulated reward as defined in equations (4)–(6) respectively.

We compare Trapper Keeper against a benchmark learning algorithm that does not transfer knowledge, two state-of-the-art TL algorithms and a naïve transfer technique. In detail, the comparison algorithms are as follows.

- **PILCO.** A state-of-the-art RL batch learning algorithm proposed in Deisenroth and Rasmussen (2011) specially designed for continuous domains. We used PILCO as the benchmark algorithm for comparison, as it is a *tabula rasa* learning algorithm that needs to learn from scratch each time a task changes. The kernel hyper-parameters were initialized as proposed in Deisenroth and Rasmussen (2011).
- **Lazaric's filter.** A state-of-the-art TL algorithm proposed in Lazaric et al. (2008) that was adapted to run with continuous actions.
- **QTL.** A TL technique proposed by Garcia et al. (2013) that transfers qualitative information using hyper-parameters from Gaussian processes.
- **Transfer all.** A technique that transfers all the samples from the source task.

Furthermore, we study different combinations of Trapper Keeper and filters. These combinations are:

- $\mathbf{TK_{all}}$– transform all samples from the source task;
- $\mathbf{TK_{simple}}$– transform a subset of source task samples using a simple filter;
- $\mathbf{TK_{laza}}$– transform a subset of source task samples using Lazaric's filter.

The experiments were performed in three benchmark tasks in the RL and control literature: the inverted pendulum on a cart and the mountain car task, as described in Sutton and Barto (1998), and a more realistic quadcopter control task with transfer to a helicopter. In all benchmark tasks, the tests were repeated five times with the initial state randomly selected. Figures 2, 3 and 4 show the accumulated reward of each algorithm for each learning episode (top part) and the corresponding standard deviation (bottom part).

This experimental setting was designed to test how the proposed algorithm (Trapper Keeper) performs on incrementally more challenging tasks along a different axis and compared to the most suitable algorithms found in the literature. Also, we show that generating prototypes with Trapper Keeper outperforms—across all measures—other transfer techniques like filters. We present such results next by tasks.

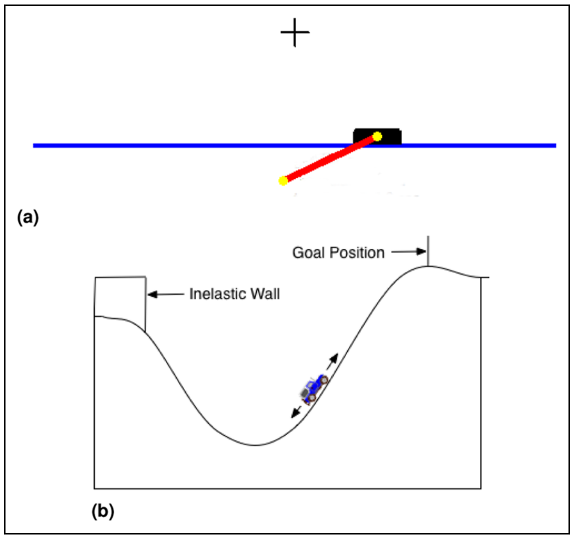

Figure 1. Two benchmark domains commonly used in RL experimentation: (a) inverted pendulum on a cart; (b) mountain car.

5.1 Inverted pendulum on a cart

This task consists of a pendulum attached to a cart that moves along a horizontal axis when a force is applied (as seen in Figure 1(a)). The objective is to raise and balance the pendulum by swinging it. The agent must learn to apply actions (in this case forces) that temporarily get away from the target state in order to finally reach the desired state. The agent learns to swing the pendulum up and balance it with the same policy, which in other approaches often requires separate controls, and thus it is not trivial to solve.

Here we consider the inverted pendulum problem with continuous action and state spaces, as defined in Deisenroth and Rasmussen (2011). In this scenario, not only has the pendulum to be swung up and balanced, but it has to be maintained at some particular point as well, which makes it a more difficult task than only balancing the pendulum. In the continuous scenario, a state $\mathbf{x}$ is formed by the position x of the cart, its velocity $\dot{x}$, the angle θ of the pendulum, and its angular velocity $\dot{\theta}$. The reward function is expressed as

$$r(\mathbf{x}) = \exp\left(-\frac{1}{2}ad^2\right) \tag{10}$$

where a is a scale constant of the reward function (set to 0.25 in the experiments) and d is the Euclidean distance between the current and desired states, expressed as $d\left(\mathbf{x}, \mathbf{x}_{target}\right)^2 = x^2 + 2xl\sin\theta + 2l^2 + 2l^2\cos\theta$, where l is the length of the pendulum. The reward remains close to zero if the distance of the pendulum tip to the target is far away. The source task consists of swinging a pendulum of mass 0.5 kg while in the target tasks the pendulums weights are changed to 0.25 kg ($0.5\times$), 1 kg ($2\times$), 1.5 kg ($3\times$) and 2 kg ($4\times$), respectively. In the

Table 1. Tabular results in the inverted pendulum task. Columns labeled 0.5 ×, 2 ×, 3 ×, 4 × represent 0.5, 2, 3, 4 times (respectively) the mass of the pendulum in original task (0.5 kg). *R*: accumulated reward; *P*: performance; *T*: convergence time. Numbers in bold represent the best result in its category.

Algorithm	0.5 ×			2 ×			3 ×			4 ×		
	R	*P*	*T*	*R*	*P*	*T*	*R*	*P*	*T*	*R*	*P*	*T*
PILCO	216.00	33.1	8	206.10	35.54	10	218.63	**33.33**	22	204.42	29.72	25
QTL	267.56	**33.62**	8	236.43	**35.66**	8	602.55	32.68	**10**	539.79	**29.97**	**12**
Transfer all	42.10	5.3	NC	48.16	4.61	NC	100.25	5.87	NC	88.46	3.75	NC
Simple filter	296.41	33.23	7	342.51	34.06	8	541.38	31.95	15	564.21	28.66	15
Lazaric's filter	336.70	33.00	6	359.73	33.61	7	560.69	32.07	15	605.71	29.36	13
TK_{all}	200.58	32.01	10	217.43	34.61	10	256.90	31.58	14	237.00	29.81	25
TK_{simple}	344.99	33.15	6	380.94	33.45	**5**	**641.18**	31.79	21	613.44	28.85	14
TK_{laza}	**346.35**	33.06	**5**	**383.37**	34.00	6	613.58	31.19	13	**616.58**	29.19	14

experiments, $l = 0.6$ m, $x \in (-\infty, \infty)$, $\theta \in [-180, 180]$ and the action or force $f \in [-1, 1]$. The number of samples used by PILCO to learn the task with mass = 1 kg (i.e. potential number of samples to transfer and total number of samples used to learn) was $285 = m$ on average. From these, the agent collects $17 = n$ sample instances from the target task on the first episode (line 2 in 1) to then generate 268 (i.e. $m - n$) prototypes initially. On each episode, new knowledge (instances) is acquired and fewer prototypes are generated, so that on episode t there will be $m - tn$ prototypes generated. If $m \leqslant tn$, no more prototypes are generated.

The results in terms of R, P and T across algorithms and specific tasks are shown in Table 1. In terms of the accumulated reward obtained by the set of algorithms, as seen in the column labeled "R" in Table 1 and displayed in Figure 2, we can draw the following conclusions.

- Transferring all instances from one task to another, without processing or filtering the samples causes negative transfer.[10] In other words, the source and target tasks are not similar enough so as to directly use all the instances from the source without hampering the model of the target task.
- The second to worst model is PILCO, which clearly reflects the benefits of TL, at least in this domain.
- Using some filters to transfer selected instances can help to improve the convergence of the algorithms. This is especially true when the target task is very similar to the source task, with superior performance to QTL, and to a lesser extent with fewer similar domains, where QTL performs better.
- Using filters and generating prototypes with Trapper Keeper outperforms the benchmark (learning from scratch using PILCO) and outperforms all the other approaches in terms of accumulated reward, where TK_{simple} and TK_{laza} clearly improve over the other approaches in the learning curves.
- Performance (column labeled "P" in Table 1) measures the reward obtained when the algorithm converges (calculated as the average over the last three episodes). As expected, the convergence performance of all algorithms is similar (except when transferring all samples). This is a sign that the algorithms converge to near optimal policies. In this case, QTL, which transfers qualitative knowledge, results in the best performance on convergence three out of four times.
- In terms of learning time (column "T" in Table 1), results are mixed, where for similar enough tasks Trapper Keeper with any filtering technique converges faster than any other transfer technique. However, as tasks become less similar, transferring qualitative information between tasks boosts learning times the most.

5.2 Mountain car

This task, as presented in Sutton and Barto (1998), consists of a car in a valley between two hills where the agent must learn a strategy to take the car up to the right side hill and stay on top. The car cannot drive up at full throttle, so a strategy to gain kinetic energy must be learned. The strategy to reach the top consists of driving up the opposite hill to gain speed, which implies moving away from the objective before reaching it.

The problem is normally described using discrete variables for actions (left, zero, right), as in Sutton and Barto (1998). Here we describe the problem using continuous spaces for states and actions. With this in mind, an agent's state is a vector $\mathbf{x} = (x, \dot{x})$, where x is the position on the horizontal axis and $\dot{x}$ is the velocity of the car on the horizontal plane. The action or force f is a single variable corresponding to the force applied to the car. The initial state is $\mathbf{x_0} = (-5, 0)$, and the goal state is whenever $x > 0.5$; however, here we consider a

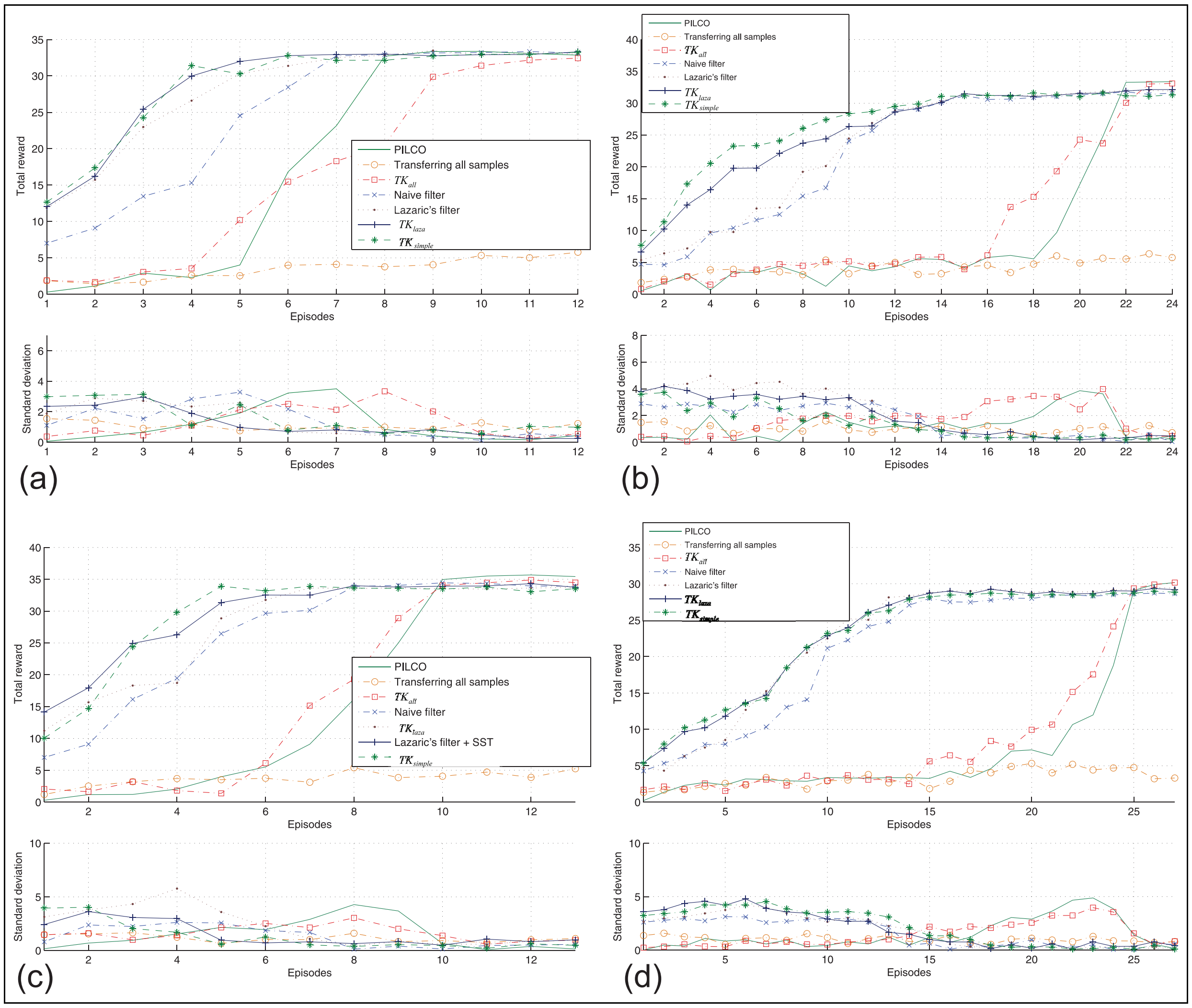

Figure 2. Learning curves in the inverted pendulum when the target task is (a) 0.5 ×, (b) 2 ×, (c) 3 × and (d) 4 × (respectively) the mass of the pendulum in the original task (0.5 kg). For each test algorithm, the averages and standard deviations are shown in the upper and lower subplots (respectively).

more challenging task of stopping the car as soon as it reaches the right hill, so the goal state is $\mathbf{x}_{\mathbf{target}} = (1, 0)$. The reward function is expressed as equation (10) where a is a scale constant of the reward function (set to 0.25 in the experiments) and $d = x - x_{\text{target}}$. The agent receives a zero reward at every time step when the goal is not reached. In the experiments, $x \in (-\infty, \infty)$ and $f \in [-1, 1]$.

We consider as the source task the one specified in Sutton and Barto (1998). For target tasks, we tested the same problem with a modified engine power of 0.5 ×, 1.5 × and 3 × the power of the source task. In this case, it is important to notice that with 3 × the power of the source, the car can drive up at full throttle without swinging. The number of samples used by PILCO to learn the task with engine power = 1 (i.e. potential number of samples to transfer and total number of samples used to learn) was $400 = m$. From these, the agent collects $40 = n$ sample instances from the target task on the first episode to then generate (i.e. $360 = m - n$) prototypes.

As can be seen in Table 2, the results lead to the following conclusions.

- Again, transferring all the instances without filters produce the worst results in terms of accumulated reward (column "R") followed by learning from scratch (PILCO), except for 3 × where the car is able to climb directly the hill. In this case, PILCO outperforms all the other transfer strategies (i.e. it produces more damage to transfer), except for *TK* which is superior to PILCO even in this particular case.
- The performance (column "P") again is similar in all the approaches, except for the case of transferring all the instances and for PILCO which shows a poor performance in the 0.5 × and 2 × cases and the best performance for the 3 × case.

Table 2. Tabular results in the mountain car task. Columns labeled 0.5 ×, 2 ×, 3 × represent 0.5, 2, 3 times (respectively) the power of the car in the original task. *R*: accumulated reward; *P*: performance; *T*: convergence time. Numbers in bold represent the best result in its category.

	0.5 ×			1.5 ×			3 ×		
Algorithm	*R*	*P*	*T*	*R*	*P*	*T*	*R*	*P*	*T*
PILCO	26.39	3.01	**4**	173.72	21.94	7	265.48	**32.60**	7
QTL	51.21	5.55	6	222.60	23.64	**2**	206.60	25.31	**5**
Transfer all	17.56	2.54	NC	52.90	5.22	NC	64.05	6.87	NC
Simple filter	45.81	4.32	6	186.32	25.32	6	202.70	30.23	7
Lazaric's filter	46.78	5.23	6	191.28	20.47	4	223.68	27.64	9
TK_{all}	50.57	5.31	6	232.47	24.74	5	252.73	30.41	6
TK_{simple}	51.34	5.30	**4**	249.75	**25.13**	4	268.78	32.19	6
TK_{laza}	**54.70**	**5.84**	**4**	**289.75**	24.23	3	**279.78**	32.34	**5**

- In terms of number of episodes there are mixed results but, in general, Trapper Keeper shows the best overall performance.

The results show that TL techniques can outperform learning from scratch (the benchmark algorithm), especially when tasks are closely related. Also, note that combining tuple transformations (Trapper Keeper) with any filtering technique can outperform any other TL techniques, as was also the case in the previous domain.

5.3 Quadcopter to helicopter

This task is the most complex and interesting experiment in this paper. Here we seek to transfer knowledge from a quadcopter to a helicopter. The task consists of finding a policy to move the aircraft from an initial (on land) position to a desired position, specified by the $(x_{\text{target}}, y_{\text{target}}, z_{\text{target}})$ coordinates. The agent must learn to take off, deal momentarily with the *ground effect*,[11] reach a specific three-dimensional position and keep the vehicle stabilized at that position. This task is learned in a quadcopter and then transferred to a helicopter which is a related vehicle, but with different dynamics.

Although both aircraft have the same state and action variables, they behave differently due to different aerodynamics. The quadcopter has four propellers which generate lift, the change in the speed of the propellers induces a change in the altitude of the quadcopter and a change in position. In the quadcopter, the difference between the torque generated by the motors is used to change the yaw angle. On the other hand, the helicopter has a main rotor which generates lift and changes position by moving the blades' angle as they rotate around the main axis. The helicopter also has a tail rotor to compensate the torque generated by the main rotor. So, in order to control the yaw angle the helicopter changes the pitch in the tail rotor's blades.

Both quadcopter and helicopter have a state vector with 12 variables, comprising its position (x, y, z), orientation (roll ϕ, pitch θ, yaw ω), velocity $(\dot{x}, \dot{y}, \dot{z})$ and angular velocity $(\dot{\phi}, \dot{\theta}, \dot{\omega})$. All of the state and action variables are continuous. We define the goal position as $[x, y, z] = [-1, -1, 1.5]$, starting from $[x, y, z] = [0, 0, 0]$. The reward function obeys equation (10), with a set to 0.25 and d evaluated as $d(\mathbf{x}, \mathbf{x}_{target})^2 = x^2 + y^2 + z^2$.

This task requires the flight to be stabilized with high precision with a large number of continuous state variables. As in the previous sections, we performed experiments with and without TL, with and without prototype generation and with and without filtering instances. For simulation purposes, and as a reference, we use V-REP (Robotics, 2013), which is a simulator, where the dynamic models for the quadcopter and helicopter have been implemented. In the experiments we used the values specified in V-REP with, $x, y, z \in (-\infty, \infty)$, row, pitch and yaw angles $\in [-180, 180]$, roll, pitch and yaw actions $\in [-1, 1]$ and for throttle $\in [0, 1]$. The number of samples used by PILCO to learn the task with the quadcopter (i.e. potential number of samples to transfer and total number of samples used to learn) was $1500 = m$. From these, the agent collects $50 = n$ sample instances from the target task on the first episode to then generate (i.e. $1450 = m - n$) prototypes.

From Figure 4 and Table 3 we can observed the following.

- The proposed approach with any filter is clearly superior to the other approaches in terms of accumulated reward and convergence time. All the transfer approaches, except when directly transferring all the instances, are superior to PILCO.
- The final performance of the all the algorithms, except when transferring all the instances without transforming, are similar and superior to the V-REP simulator, as they learn to compensate the inertia of the helicopter by tilting pitch and roll angles before the helicopter reaches the target

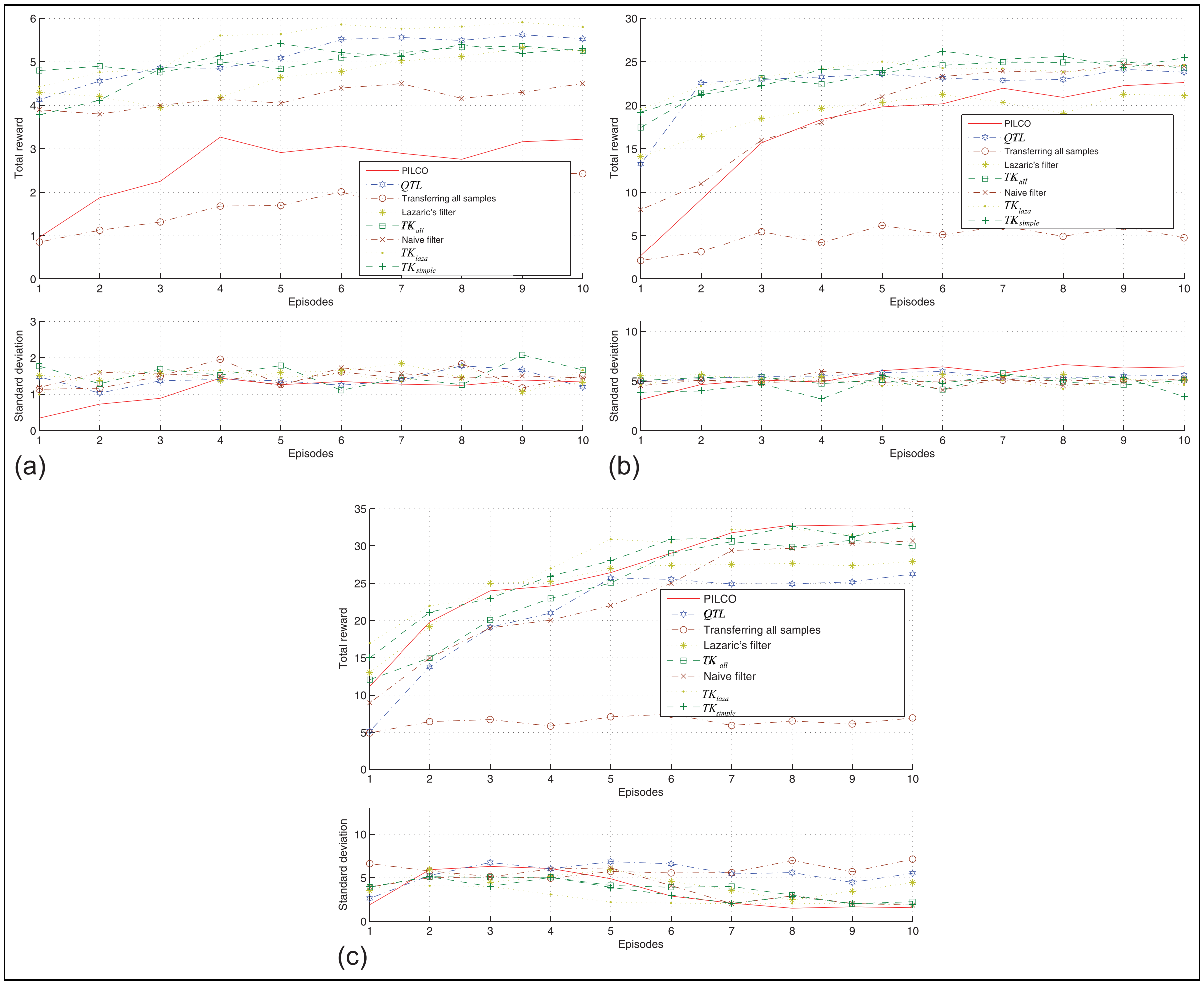

Figure 3. Learning curves in the mountain car task when the target task is (a) 0.5 ×, (b) 2 × and (c) 3 × (respectively) the car power with respect to the original task. For each test algorithm, the averages and standard deviations are shown in the upper and lower subplots (respectively).

Table 3. Total reward, final performance and convergence time for the helicopter task.

Algorithm	Total reward	Final performance	Convergence time
V-REP Autopilot (as reference)	-	126.79	-
PILCO	1809.44	132.50	24
QTL	2225.46	**131.99**	22
Transfer all	467.89	22.5	NC
Simple filter	2388.77	128.69	22
Lazaric's filter	2578.79	129.77	20
TK_{all}	2811.15	131.33	18
TK_{simple}	3169.64	131.39	17
TK_{laza}	**3212.75**	130.29	**15**

position in contrast to V-REP's autopilot which tends to overshoot.

5.4 Overview of the experiments

The experiments performed in Sections 5.1, 5.2 and 5.3 were used to evaluate different TL strategies and without TL (PILCO). From the different experiments we can conclude the following.

- As expected, TL can significantly reduce the number of episodes required for convergence and increase the total accumulated rewards with a similar final performance as without using any TL.
- Transferring source instances without any filtering severely harms the learning process. However, selecting some of them can produce clear benefits.

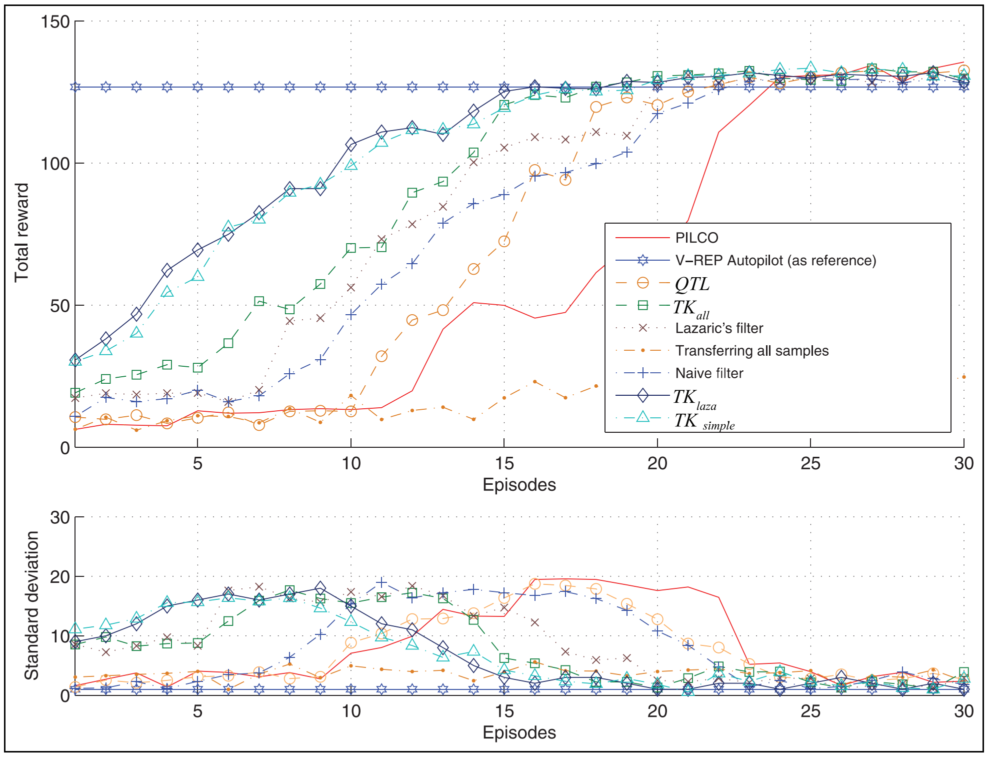

Figure 4. Learning curves for the helicopter task using transfer learning from a quadcopter. The figure show the accumulated reward of the V-REP simulator as reference (top figure) and the standard deviations (bottom figure).

- Generating prototypes is the best option when combined with a filtering mechanism, although the particular filter used to select instances seems not to be very relevant.

6 Conclusions and future work

We have presented a novel TL approach for continuous state and action spaces. The algorithm learns a difference function between the source and the target task and uses it to generate prototypes using selected instances from the source task to learn faster. We performed several experiments under different TL conditions and show that the proposed approach can help to significantly improve the learning process.

As future work we would like to transfer from several tasks, where a more sophisticated filter mechanism needs to be defined. Also we would like to consider domains with different state and action vector variables and we will like first to consider domains where the state and action variables of one of the tasks is a subset of the other task.

Notes

1. We cannot expect a robot to interact with the environment thousands of times just to train it for a simple task.
2. As in Taylor and Stone (2009), we refer to the term *task* as the specification of the domain and its objective (i.e. the MDP specification).
3. We use the terms tuples and instances indifferently throughout the document.
4. An example instance could be of the form (state, action, next state). See Section "3" for a formal definition of this concept.
5. The Q-function is a matrix of states and actions that captures the expected sum of discounted rewards for each state and action pair.
6. We refer the reader to (Taylor & Stone, 2009) and Lazaric (2008) for a thorough examination of TL definitions and objectives.
7. Note that in the general case the source policy will not help as a starting point, but for robotics applications like the ones presented in this work starting with an initial (not random) policy is needed to prevent the robot from performing actions that could harm the hardware.
8. The authors are not aware of any PAC-learning results for Gaussian processes, but normally the number of samples needed to guarantee some degree of performance is too large, especially for Gaussian processes where it is desirable to keep this number as small as possible.
9. Convergence time is defined in terms of episodes, where an episode is a single interaction with the environment.
10. Negative transfer is an effect suffered when the knowledge transferred between tasks results in a worst convergence performance and/or accumulated reward compared to learning the objective task from scratch.
11. Ground effect is an aerodynamic effect derived from air hitting the ground when an aircraft is close to a surface that makes harder to lift and control the aircraft.

References

Ammar, H., Tuyls, K., & Taylor, M. (2012). Reinforcement learning transfer via sparse coding. In *Proceedings of the 11th international conference on autonomous agents and multiagent systems* (*Vol. 1*, pp. 383–390).

Atkeson, C., & Santamaria, J. (1997). A comparison of direct and model-based reinforcement learning. In *IEEE international conference on robotics and automation* (*Vol. 4*, pp. 3557–3564). Piscataway, NJ: IEEE.

Bertsekas, D., & Tsitsiklis, J. (1996). *Neuro-dynamic programming*. Belmont, MA: Athena Scientific.

Bou Ammar, H., Eaton, E., Ruvolo, P., & Taylor, M. E. (2015). Unsupervised cross-domain transfer in policy gradient reinforcement learning via manifold alignment. In *Proceedings of the twenty-ninth AAAI conference on artificial intelligence* (pp. 2504–2510). AAAI Press. Austin, Texas, USA. AI Access Foundation.

Deisenroth, M., Peters, J., & Rasmussen, C. (2008). Approximate dynamic programming with Gaussian processes. *In American control conference (pp.* 4480–4485).

Deisenroth, M., & Rasmussen, C. (2011). PILCO: A model-based and data-efficient approach to policy search. In L. Getoor, & T. Scheffer (Eds.), *ICML* (pp. 465–472).

Deisenroth, M., Rasmussen, C., & Fox, D. (2011). Learning to control a low-cost manipulator using dataefficient reinforcement learning. In *Proceedings of robotics: Science and systems*. Los Angeles, CA.

Deisenroth, M., Rasmussen, C., & Peters, J. (2008). Model-based reinforcement learning with continuous states and actions. In *16th european symposium on artificial neural networks* (pp. 19–24), Bruges, Belgium.

Dietterich, T. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Drummond, C. (2002). Accelerating Reinforcement Learning by Composing Solutions of Automatically Identified Subtasks. *Journal of Artificial Intelligence Research*, *16*, 59–104.

Engel, Y., Mannor, S., & Meir, R. (2003). Bayes meets bellman: The gaussian process approach to temporal difference learning. In T. Fawcett, & N. Mishra (Eds.), *ICML* (pp. 154–161). AAAI Press, Washington D.C.

Engel, Y., Mannor, S., & Meir, R. (2005). Reinforcement learning with Gaussian processes. In *Proceedings of the 22nd international conference on Machine learning - ICML '05* (pp. 201–208), ACM International conference proceedings series , Bonn, Germany.

Ferguson, K., & Mahadevan, S. (2006). Proto-transfer learning in markov decision processes using spectral methods. In *Proceedings of the ICML-06 workshop on structural knowledge transfer for machine learning*. Citeseer.

Garcia, E., Munoz de Cote, E., & Morales, E. (2013). Qualitative Transfer for Reinforcement Learning with Continuous State and Action Spaces. In J. Ruiz-Schucloper, & G. Sanniti di Baja (Eds.), *18th. iberoamerican congress on pattern recognition* (pp. 198–205). Springer-Verlag Santiago, Chile.

Garcia, E., Munoz de Cote, E. & Morales, E. (2014). Transfer learning for continuous state and action spaces. *International Journal of Pattern Recognition and Artificial Intelligence*, *28* 1460007 (20 pages).

Hasselt, H. (2011). Reinforcement Learning in Continuous State and Action Spaces. In *Reinforcement learning: State of the art*. Springer.

Jaakola, T., Jordan, M., & Singh, S. (1995). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, *6*, 1185–1201.

Lazaric, A. (2008). *Knowledge transfer in reinforcement learning* (PhD Thesis). Politecnico di Milano, Italy.

Lazaric, A., Restelli, M., & Bonarini, A. (2007). Reinforcement learning in continuous action spaces through sequential monte carlo methods. In *Advances in neural information processing systems*. Citeseer.

Lazaric, A., Restelli, M., & Bonarini, A. (2008). Transfer of samples in batch reinforcement learning. *In Proceedings of the 25th international conference on Machine learning - ICML '08 (pp.* 544–551).

Martín, H. J., de Lope, J., & Maravall, D. (2011, March). Robust high performance reinforcement learning through weighted k-nearest neighbors. *Neurocomputing*, *74*, 1251–1259.

Mehta, N., Natarajan, S., Tadepalli, P., & Fern, A. (2008). Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, *73*, 289–312.

Murray-Smith, R., & Sbarbaro, D. (2002). Nonlinear adaptive control using non-parametric Gaussian process prior models. In *In 15th IFAC world congress on automatic control* (pp. 21–26).

Ng, A., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the sixteenth international conference on machine learning* (pp. 278–287). Morgan Kaufmann.

Ng, A., Kim, H., Jordan, M., & Sastry, S. (2004). Autonomous helicopter flight via reinforcement learning. In S. Thrun, S. Lawrence & B. S. (Eds.), *Advances in neural information processing systems 16*. Cambridge, MA: MIT Press.

Puterman, M. (1994). *Markov decision processes—discrete stochastic dynamic programming*. New York, NY: John Wiley & Sons, Inc.

Rasmussen, C., & Deisenroth, M. (2008). Probabilistic inference for fast learning in control. *Recent Advances in Reinforcement Learning*, *5323*(November), 229–242.

Rasmussen, C., & Kuss, M. (2004). Gaussian Processes in Reinforcement Learning. *Advances in Neural Information Processing Systems*, *16*, pp. 751–759.

Rasmussen, C., & Williams, C. (2006). Gaussian Processes for Machine Learning. *International Journal of Neural Systems*, *14*(2), 69–106.

Robotics, C. (2013). V-rep pro edu, version 3.0.1 [Computer software manual]. Retrieved from http://www.coppeliarobotics.com/

Silver, D., et al. (Eds.). (2005). *Inductive transfer*: 10 years later workshop.

Soni, V., & Singh, S. (2006). Using homomorphisms to transfer options across continuous reinforcement learning domains. In *AAAI* (pp. 494–499).

Sutton, R., & Barto, A. (1998). *Introduction to Reinforcement Learning*. MIT Press.

Taylor, M., Jong, N., & Stone, P. (2008). Transferring instances for model-based reinforcement learning. *Joint*

European Conference on Machine Learning and Knowledge Discovery in Databases. Springer Berlin Heidelberg.

Taylor, M., & Stone, P. (2009). Transfer learning for reinforcement learning domains : A survey. *Journal of Machine Learning Research*, *10*, 1633–1685.

Taylor, M., Stone, P., & Liu, Y. (2007). Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, *8*, 2125–2167.

Torrey, L., Walker, T., Shavlik, J., & Maclin, R. (2005). Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg (pp. 412–424).

Triguero, I., Derrac, J., Garcia, S., & Herrera, F. (2012). A taxonomy and experimental study on prototype generation for nearest neighbor classification. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, *42*(1), 86–100.

van Hasselt, H. (2011). *Insights in reinforcement learning: formal analysis and empirical evaluation of temporal-difference learning algorithms* (Unpublished doctoral dissertation). Universiteit Utrecht, The Netherlands.

van Otterlo, M. (2009). *The logic of adaptive behavior: Knowledge representation and algorithms for adaptive sequential decision making under uncertainty in first-order and relational domains*. IOS Press.

About the Authors

Enrique Munoz de Cote graduated in 2008 from Politecnico di Milano with a PhD in computer science. He spent two years at the University of Southampton as a postdoc, researching in the boundary between game theoretic interactions, machine learning and implicit negotiations. He was an invited scholar at the RL3 lab at Rutgers university and has been awarded for his research, including a UAI best student paper and two times winner of the Lemonade Stand Game tournament organised by Yahoo! research. He has been responsible of 4 research projects and has published over 50 papers in top-tier conferences and scientific journals. He is now associate professor at INAOE (Mexico) where he leads the COLD intelligent systems lab, which focus on intelligent systems, machine learning and human-computer interactions. He is also a member of the board of directors of the Association for Trading Agent Research (ATAR) and a member of the International Foundation for Autonomous Agents and Multiagent Systems.

Esteban O García received his BSc degree in Computer Sciences from "Benemerita Universidad Autonoma de Puebla" and his M.Sc. and Ph.D degrees from "Instituto Nacional de Astrofisica, Optica y Electronica", in Puebla, Mexico. He is currently a collaborator in the project "High precision agriculture through collaborative UAVs" in the COLD group at INAOE and the CEO in Vertical AP, an aerial robotics startup company in Mexico. His recent research focuses on transfer learning and close range drone navigation in construction and mining sites.

Eduardo F Morales received his PhD degree from the Turing Institute - University of Strathclyde, in Scotland. He has been responsible of more than 25 research projects and has more than 150 peer-review papers. He was an Invited Researcher at the Electric Power Research Institute (1986), a Technical Consultant (1989–1990) at the Turing Institute, a Researcher at the "Instituto de Investigaciones Electricas" (1986–1988 and 1992–1994) and at ITESM - Campus Cuernavaca (1994–2005). He is currently a senior researcher at the "Instituto Nacional de Astrofísica, Óptica y Electrónica" (INAOE) in Mexico where he conducts research in Machine Learning and Robotics.