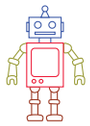
Generalization strategies in reinforcement learning

Snel, M.

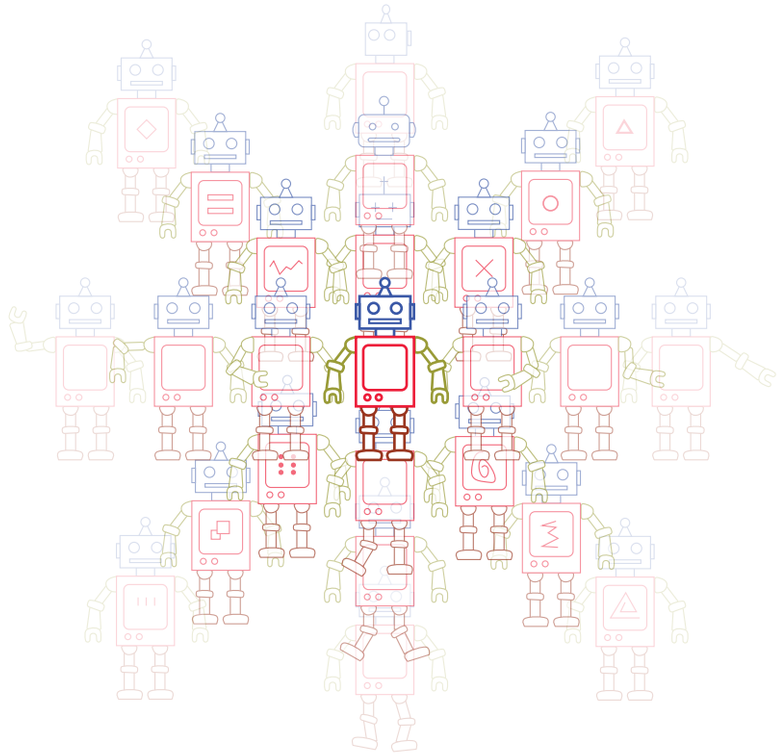
Publication date
2018
Document Version
Final published version
License
Other



Generalization Strategies in Reinforcement Learning

Matthijs Snel

Generalization Strategies in Reinforcement Learning



Matthijs Snel

Generalization Strategies in Reinforcement Learning

Matthijs Snel

Generalization Strategies in Reinforcement Learning

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K. I. J. Maex
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in
de Aula der Universiteit
op vrijdag 20 april 2018, te 11.00 uur

door

Matthijs Snel

geboren te Utrecht

Promotiecommissie

Promotor:		
	Prof. dr. ir. B. J. A. Kröse	Universiteit van Amsterdam
Co-promotores:		
	Prof. dr. ir. F. C. A. Groen	Universiteit van Amsterdam
	Dr. S. A. Whiteson	University of Oxford
Overige leden:		
	Prof. dr. K. P. Tuyls	University of Liverpool
	Prof. dr. P. W. Adriaans	Universiteit van Amsterdam
	Prof. dr. M. Welling	Universiteit van Amsterdam
	Dr. M. A. Wiering	Rijksuniversiteit Groningen
	Dr. T. E. J. Mensink	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

1 Introduction

Hermann, a German Shepherd dog, is being trained for search and rescue: he is tasked with locating a human victim in a bounded area of terrain, and is rewarded with praise and bits of food upon completion of the task. After several such exercises – all on different terrains, with the victim in a new, unknown location each time – Hermann has figured out that the overall goal is to follow the human scent to locate the victim as quickly as possible. Using this knowledge, he is able to easily complete new tasks, possibly on larger and more complex terrains, just by following his nose.

Hermann is an example of a *learning agent*. In this thesis, an *agent* is any entity that perceives its environment through sensors (in Hermann's case, eyes, nose, ears) and acts upon it through effectors (paws, teeth). While Hermann is organic, this thesis concerns itself with artificial agents such as robots and software agents. What does it mean for an (artificial) agent to learn? A commonly cited quantifiable definition is that [92]

> [An agent] is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

The crucial point is thus that the agent's behavior or output, or whatever P applies to, improves with experience.

The above example further illustrates the two central themes that this thesis focuses on:

- **Reinforcement learning**. A reinforcement-learning agent learns through trial and error by interacting with the world and observing the effect of its actions and the reward or punishment that it receives after each action. Generally, the goal of the agent is to identify the sequence(s) of actions that will lead to the maximal sum of rewards. In the example, Hermann the dog is the agent, and the reward comes in the form of praise and food. Note that reward can be significantly *delayed* – Hermann only receives it after taking many actions. In the case of artificially intelligent agents, the dog could be replaced by a robot equipped with appropriate sensors. Among the many examples of tasks that artificial reinforcement learning agents have succeeded at are helicopter control [1], playing the game of Go at and beyond human expert level [120], and elevator system control [18].

- **Multi-task reinforcement learning**. Hermann the dog is engaged in a sequence of

tasks; one task corresponds to the location of a given victim in a given terrain. Unknowingly, Hermann is using multi-task reinforcement learning (MTRL): he uses knowledge gained in previous tasks to learn new tasks more efficiently. Efficiency could be measured, for example, by measuring the time, energy, or amount of experience (or data) required to learn the task. Hermann is doing this by abstracting away sensory input and building a representation of the input that is relevant across tasks, namely (mainly) smell, in this example[1]. This improves efficiency since Hermann can now follow his nose instead of having to explore the terrain semi-randomly. The first part of this work investigates methods for discovering just such cross-task relevant sensory inputs.

Why should we care about the ability to learn or perform on new tasks efficiently (in terms of time, energy / computing power, or data), when computing power and data are so abundant nowadays? Consider the following factors:

- **Data availability**. Data is not ubiquitous in all application areas, and even if it is, significant time may be involved in gathering it. For example, consider a trading agent that trades products on capital markets. As a new financial product is launched, the trading agent would not like to wait weeks or even months until it has gathered sufficient data to learn how to trade this new product, thereby incurring a potentially large opportunity cost. Instead, the agent would like to start trading right away, using its knowledge of similar financial products. Its trading strategy will likely not be optimal for this product from the start, but it may be good enough to start making a profit early on. Indeed, one of the aims of the thesis is to jumpstart performance on new tasks, rather than learning a better final solution.

- **Online performance**. The distinction between "offline" and "online" performance, for the purpose of this thesis, is that the latter is measured while the agent is engaged in the actual task, at a time when performance matters. For example, robot controllers are often pre-trained in simulation ("offline") before being deployed to the real world, on the actual robot ("online"): It would be a risky and time-consuming endeavor to learn from scratch on the real robot, since it might irreparably damage itself while learning. In contrast, during offline learning, performance in terms of accrued reward is not critical, as long as the learning agent reaches an acceptable performance threshold within an acceptable time. However, when having to learn a new task online – for example, a Mars rover faces an unforeseen terrain, or a household robot needs to learn a new domestic task –, performance while learning is critical, and leveraging knowledge from previously encountered tasks can help jumpstart or otherwise boost this performance.

- **Generalization and robustness**. Just as supervised learning agents should be able to generalize from observed samples to new samples, so reinforcement learning agents should be able to generalize from observed tasks to new tasks. In this case, we would like the agent's performance not to deteriorate significantly when it encounters tasks with slight variations on the tasks it has seen during learning. This is

[1] Of course, other sensory modalities such as vision also help in completing tasks efficiently. Those modalities are, indeed, helpful for MTRL on an even larger scale: daily life. Smell, however, is idiosyncratic to this particular example.

important for many real-world applications where, for example, the agent has been trained in a simulator before being deployed to the real world, such as in robotics.

This thesis investigates strategies that reinforcement-learning agents can employ to attempt to maximize a performance measure while engaging in a sequence of tasks. The strategies this thesis discusses can broadly be divided into two approaches. The first of these concerns agents that explicitly examine and leverage structure and knowledge that is shared between tasks. Hermann is an example of this: the structure that is shared between tasks is that following smell leads to reward.

The second approach investigates neural controllers that exhibit a degree of robustness to changes in environment; a robust controller is here defined as a controller of which performance, measured according to the objective function of the task on which the agent was trained, does not significantly deteriorate in the face of dynamics changes. This means that the agent needs to spend no or less time to learn to adjust to the change, which improves efficiency. For example, Hermann's performance would be mostly unaffected by differences in terrain, up to a point.

In order to introduce the questions addressed in this thesis in more detail, some more formal background on relevant concepts is discussed in the next section.

1.1 Reinforcement Learning

A *dynamical system* is a time-dependent system. The *state* of the system at any given moment is modeled by a real-valued vector that contains all variables required to compute the next state of the system. For example, a swinging pendulum can be completely described by two variables: its angle, and its velocity.

It is often desirable to control a dynamical system in such a way as to achieve a predetermined objective. For example, one may wish to balance a pendulum in a vertical position; this requires applying small force pulses at appropriate moments in order to prevent the pendulum from falling. More practical examples are as diverse as maintaining a vat of chemicals in a reactor at the right temperature and concentrations, controlling the growth of a population of an animal or plant species, controlling a mobile robot, and adjusting a financial portfolio in order to maximize profit.

Control of dynamical systems lies at the heart of the field of *optimal control*. In an optimal control problem, each state transition is assigned a *utility* or *reward*, or equivalently a *cost* (negative reward). Transitions not only depend on the current state, but also on *actions* or *controls* exerted by the agent controlling the system. The goal is to find the sequence of actions that maximizes the sum of rewards, or *return*, for a given time horizon; optimal control is thus concerned with solving *sequential decision problems*.

A typical assumption is that the (probability distribution over the) system's next state (and reward) can be fully determined given only its current state (and action) and the transition rules. This implies that knowing the history of transitions and actions is of no additional use in predicting the future. Systems for which this holds are said to possess the *Markov property*, which turns the optimal control problem into a *Markov decision process* (MDP). Since all that is needed is the current state, the Markov property permits a solution that is a mapping from each state to the appropriate action, also called a (memoryless) *policy*. *Dynamic programming* (DP) [8], pioneered by Richard Bellman in

the 1950s, is a set of methods for solving MDPs. Bellman also realized that DP methods suffer from the *curse of dimensionality*: in the worst case, the size of the state space, and therefore the effort required to solve an MDP, increases exponentially with the state dimensionality, the number of variables used to describe the state. Nevertheless, dynamic programming is the method of choice when a full model of transitions and rewards is available.

When no model is available, it becomes essential to *learn* the dynamics of the system to be controlled by interacting with it. *Reinforcement learning* [60, 135] is the unification of ideas from the fields of optimal control and trial-and-error animal learning. In reinforcement-learning terminology, the decision maker is called the *agent* and the controlled system the *environment*. The agent learns about the unknown environment by interacting with it and observing the transition and received reward after each action it takes. The goal remains unchanged: find the policy that maximizes expected return.

A reinforcement-learning agent faces several fundamental challenges. Firstly, it receives a potentially *delayed reward*: reward may only be received after a long sequence of actions has been taken (for example, consider a sequence of moves required for winning a game of chess). Related to this is the *credit assignment problem*: how should the agent distribute credit for the reward among all the actions comprising the sequence leading up to it?

Secondly, the agent can only learn about the environment through trial and error. This leads to an *exploration/exploitation dilemma*: at any given time step, the agent needs to decide whether it chooses the action that is currently estimated as best for the present state (exploitation), or choose to gain more information about the environment by picking a different action that might lead to higher return, but might also have negative consequences (exploration).

Traditionally, the aim has been to converge to the optimal policy, which maximizes expected return. However, due to the curse of dimensionality, delayed reward, and the exploration / exploitation dilemma, on some problems a naive RL agent can take a long time to converge to a solution. Expected *online return* (return incurred while the agent is learning and interacting with the environment), rather than convergence, is often more important. By guiding the agent's exploration strategy, the application of prior knowledge, for example gained through experience of previous tasks, is one tool for improving online return.

1.2 Recurrent Neural Networks

As mentioned earlier, the second generalization strategy this thesis investigates makes use of neural controllers. A feedforward neural network (FNN) may, in the context of reinforcement learning, be used to represent policies, taking states as input and producing actions as output; or as value functions, taking states as input and producing expected return as output. An FNN is a directed acyclic computation graph organized into layers of nodes, where each node computes a (usually non-linear) transformation of its inputs in order to produce a desired output. Layers in between the input and output layer are called *hidden* layers. So-called deep FNNs typically use many hidden layers, while shallow FNNs typically have only one hidden layer. Each edge in the graph has a weight, and

learning occurs by updating the weights based on a given objective function and learning algorithm. Since FNNs produce outputs based solely on the current input state, they implicitly assume inputs have the Markov property.

A *recurrent* neural network (RNN) is a popular tool for non-Markovian problems. It is a directed cyclic graph that introduces recurrent connections, typically on the hidden layer(s). This makes the net much more powerful, though generally harder to train; for example, RNNs are universal Turing machines [118], and nonlinear dynamical systems that can represent complex dynamics such as oscillatory and chaotic functions. In an RNN with a single recurrent connection, the network essentially takes the current state and a transformation of the previous state as input. This thus allows it to maintain an internal state which summarizes the history of inputs seen so far.

Briefly, the following RNN architectures are used in this thesis (each will be described in detail in section 5.2). The most basic, also known as an Elman network [30], is a shallow network with a recurrent connection on the hidden layer, which we here call a discrete-time RNN (DTRNN). Most RNNs assume time is discrete, but one can also define continuous-time networks based on differential equations with respect to time, and the one we use here is the direct continuous-time analog of the DTRNN, a CTRNN. Basic RNNs can have trouble learning long-term dependencies between input and output because of the *vanishing gradients* problem: propagating error gradients many steps backward in time may cause dilution and eventually the disappearance of the (relevant) gradient. Gated architectures, that allow activations and gradients to circulate in theory indefinitely, were designed to address this problem and allow networks to learn long-term dependencies. Two popular ones are the Long Short-Term Memory (LSTM) [40, 54] and Gated Recurrent Units (GRU) [16]. Finally, Echo State Networks (ESN) [57] treat the RNN explicitly as a dynamical system, and aim to create the desired dynamics prior to learning by initializing it with a large number of hidden nodes and scaling the weights in order to achieve rich dynamics, and subsequently only learn the output weights.

1.3 Focus and Research Questions

This thesis develops theory for, and empirically compares, several strategies for improving online return across multiple related reinforcement-learning tasks. These strategies can be roughly divided into two categories. The first entails a form of *transfer learning*. Here, prior knowledge is derived from previous tasks seen by the agent or other agents in order to guide exploration and improve online return. More specifically, transfer learning aims to improve return on a set of *target tasks* by leveraging experience from a set of *source tasks*. Clearly, the target tasks must be related to the source tasks for transfer to have an expected benefit. In *multi-task reinforcement learning* (MTRL), this relationship is formalized through a *domain*, a distribution over tasks from which the source and target tasks are independently drawn [140, 155]. In this thesis, agents capture domain knowledge by learning a *shaping function*, which is a function that aims to guide the agent with additional informative reward on top of the base reward of the MDP. Shaping functions can be functions of state, state-action pairs, or transitions; they can be human-designed, or learned by the agent itself. While they can be equivalent to value functions, this does not have to be the case; for example, a human designer could choose to only

provide additional reward in certain important regions of the state space.

While shaping functions and similar methods have been used in previous work on multi-task learning [68, 122, 138], that work provided no theoretical motivation for the target function the shaping function should approximate. Furthermore, previous work relied on manually designed representations for shaping or other cross-task functions [37, 38, 68, 122], or learned a representation but did not learn a cross-task function [36].

We propose and compare several different forms of shaping function, and in addition develop a theoretical foundation and algorithm for learning the cross-task representation the shaping function should be based on. The research questions addressed in this part of the thesis are:

- What kind of targets could a cross-task shaping function approximate, and under which settings do these targets perform well?
- What state representation should a cross-task shaping function be based on, and what distinguishes this state representation from the representation that should be used for the value function of each individual task?
- Is it possible to develop an algorithm for finding both kinds of representation?

While learning about dynamics changes – within or across tasks – is a natural approach, doing so may be expensive. On a real robot, for example, it may be both time-consuming and risky. Therefore, the second part of the thesis investigates fixed controllers that nonetheless exhibit a degree of robustness to dynamics changes, such as variations in terrain and sensor noise. By virtue of their ability to exhibit rich dynamics and maintain internal state, recurrent neural networks (RNNs) seem like an appropriate class of parameterized, learnable architectures for creating fixed, robust controllers.

In supervised learning, RNNs, in particular the Long Short-Term Memory (LSTM, an RNN designed specifically to learn about long-term dependencies, and explained in more detail in Section 5.2.3) have achieved state-of-the-art results in for example machine translation [133] and speech recognition [43]; in RL, most recent work that learns the full RNN parameterization[2] is also LSTM-based [5, 27, 53, 156].

The second part of the thesis benchmarks several modern RNN architectures, and a deep and shallow FNN used as baseline, in RL for continuous control on a set of simulated robotic locomotion tasks, and tests the resulting policies on robustness in the face of dynamics changes. It addresses the following questions:

- How do the architectures relate to each other in terms of learning performance, and why?
- How do the architectures relate to each other in terms of testing performance when the policies are fixed, and why?
- How robust is each of the architectures to a change in dynamics in the form of sensor noise and a terrain switch?

[2] This does not include the body of work that makes use of central pattern generators or other pre-designed recurrent systems and learns a limited set of parameters, e.g. [31].

1.4 Overview

Chapter 2 provides a more detailed and formal overview of the concepts on which the new material introduced in this thesis builds. Sections 2.1 to 2.3 review dynamical systems, Markov decision processes, and basic reinforcement learning theory and algorithms, with additional detail on partially observable MDPs (POMDPs), state abstraction, and shaping functions. The second half of the chapter, starting in section 2.3.4, reviews material required for understanding Chapter 5, in particular policy gradient algorithms and recurrent neural networks.

Our contributions start in Chapter 3, which proposes and empirically compares several shaping functions for MTRL. We propose and provide arguments for three different targets the shaping function should approximate, based on the optimal value function of the encountered tasks, the approximate value function, and the value function of a single cross-task policy. We evaluate each target empirically on a number of simulated domains.

Besides the target function the shaping function should approximate, another key issue to address when learning shaping functions, as with any function, is which representation it should be based on. Multi-task learning settings add another dimension to this problem.

Chapter 4 distinguishes between *task-relevant* and *domain-relevant* representations. A task-relevant representation is a single representation with which a different function is learned in each task. Its power lies in its compactness, which makes each task easier to learn. Domain-relevant representations can serve as a basis for a *single* cross-task function, such as a shaping function, that captures (approximately) invariant domain knowledge. Roughly speaking, such a function obviates the need to re-learn task-invariant knowledge and biases the agent's behavior. We introduce the notion of k-relevance, the expected relevance on a sequence of k tasks sampled from the domain, and argue that this is a unifying definition of relevance of which both task and domain relevance are special cases. We prove that, under certain assumptions, k-relevance is an approximately exponential function of k, and use this property to derive *Feature Selection Through Extrapolation of k-relevance* (FS-TEK), a novel feature-selection algorithm. The key insight behind FS-TEK is that change in relevance observed on task sequences of increasing length can be extrapolated to more accurately predict domain relevance. Furthermore, we demonstrate empirically the benefit of FS-TEK on three artificial domains.

Chapter 5 extends the study of Duan et al. [27] to benchmark five RNN architectures: DTRNN, CTRNN, ESN, LSTM and GRU, and a deep and shallow FNN as baselines on a set of OpenAI Gym [12] simulated locomotion tasks. Networks are trained under two regimes: on flat terrain, corresponding to the regular OpenAI Gym tasks, and on a hilly terrain variation that we introduce. After learning, the network policies are fixed, and each architecture is tested on performance both on the learning task and on two types of perturbation: sensor noise, and a switch from flat to hilly terrain. While the FNNs learn fastest, no single architecture is best at test time. However, we show that the FNNs and CTRNN are most robust to dynamics changes on average, with the CTRNN significantly outperforming the others under noise perturbation. In addition, we show that training on the hill task benefits the RNN, but not the FNN, performance on noise perturbation.

Finally, chapter 6 concludes with a discussion of the results and their implications, and ideas for further work.

1.5 List of Publications

Work that culminated in Chapter 3 and 4:

- Snel, M., & Whiteson, S. (2010). Multi-task evolutionary shaping without pre-specified representations. In *Proceedings of the 12th annual conference on Genetic and Evolutionary Computation (GECCO)* (pp. 1031-1038).
- Snel, M., & Whiteson, S. (2011). Multi-Task Reinforcement Learning: Shaping and Feature Selection. In *European Workshop on Reinforcement Learning (EWRL)* (pp. 237-248).
- Snel, M., & Whiteson, S. (2014). Learning potential functions and their representations for multi-task reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 28(4), 637-681.

Work that culminated in Chapter 5:

- Snel, M., Whiteson, S., & Kuniyoshi, Y. (2011). Robust central pattern generators for embodied hierarchical reinforcement learning. In *2011 IEEE International Conference on Development and Learning (ICDL)*, (Vol. 2, pp. 1-6).
- Snel, M. (2017). Benchmarking Recurrent Networks for Robust Continuous Control. Submitted to *International Conference on Robotics and Automation (ICRA)*.

2 Background

This chapter provides background on concepts required for understanding this thesis. Section 2.1 discusses dynamical system theory, and section 2.3 discusses Markov decision processes and reinforcement learning.

2.1 Dynamical Systems

A *dynamical system* refers to a time-dependent system, or a mathematical model that describes such a system – this latter interpretation is the one used in this thesis. The model consists of (1) a state space, (2) a continuous or discrete measure of time and (3) a time-evolution rule that governs how a given state transitions into the next. The *state* the system is in at any given moment is commonly modeled by a real-valued vector $\mathbf{s} \in \mathbb{R}^d$, $\mathbf{s} = (s_1, s_2, \ldots, s_d)$, where each s_i is the value of state variable S_i (unless specified otherwise, uppercase letters (S) denote random variables, and lowercase letters (s) denote their values; similarly, lowercase boldface denotes vectors of values ($\mathbf{s}$), and uppercase boldface ($\mathbf{S}$) denotes either a random vector, the set of all possible $\mathbf{s}$, or a matrix). The set of all possible states comprises the state or *phase space* $\mathbf{S} \subseteq \mathbb{R}^d$.

The most general time-evolution rule determines the state of the system at time t from its states at all previous times; it is therefore time-dependent and has infinite memory [87]. It is common, however, to let the evolution of the system depend only on its current state. While not usually explicitly mentioned in the dynamical systems literature, this implies that the system's state has the *Markov property* (section 2.2 will discuss this in more detail).

Given the system's current state, transitions to the next state are given by a set of equations. For continuous-time models, these are differential equations; the system is then called a *flow* and takes the form

$$\dot{\mathbf{s}} = \mathbf{f}(\mathbf{s}), \tag{2.1}$$

where $\dot{\mathbf{s}}$ is the derivative of the state vector $\mathbf{s}$ with respect to time, and $\mathbf{f}$ is the time-evolution rule. Discrete-time models are described with difference equations; the system is then called a *map* and takes the form

$$\mathbf{s}_{k+1} = \mathbf{f}(\mathbf{s}_k). \tag{2.2}$$

Starting from a given initial state and repeatedly applying the equations describing the change of the system results in a *trajectory* through state space, also called an *orbit*. A dynamical system is *linear* if $\mathbf{f}$ is linear in the state, and nonlinear otherwise.

2.2 Markov Decision Processes

It is often desirable to control a dynamical system in order to achieve a predetermined objective. Prediction of how a system will change is essential to control. As mentioned earlier, the assumption that the state of the system includes all information necessary to predict the next state implies that it has the Markov property. A dynamical system that has the Markov property is a *Markov process*. More formally, in a Markov process,

$$p(\mathbf{S}_t|H_t) = p(\mathbf{S}_t|\mathbf{S}_{t-1}), \tag{2.3}$$

where $p(\cdot)$ denotes probability and $H_t = \{\mathbf{S}_{t-1}, \mathbf{S}_{t-2}, \ldots, \mathbf{S}_0\}$ is the history up to time t; as in most of this thesis, we assume time is discrete here. In other words, in a Markov process, the future is conditionally independent of the past given the present. Often, a system that is not a Markov process can be converted into one by augmenting the state representation such that it becomes a sufficient statistic for the system's history; for example, in the pendulum system, the inclusion of angular velocity in the state obviates the need for the angle history (at the cost of a larger state space size).

Control of Markov processes lies at the heart of the field of *optimal control*. In an optimal control problem, each state transition is assigned a *utility* or *reward* r, or equivalently a *cost* (negative reward). Transitions not only depend on the current state, but also on the *action* or *control* $\mathbf{a}$ exerted by the agent controlling the system; this turns the process into a **Markov decision process** (MDP)[8, 102]. The goal is to find the sequence of actions that maximizes the sum of rewards, or *return*, for a given time horizon. For an MDP, the Markov property implies that $p(\mathbf{S}_t, R_t|H_t) = p(\mathbf{S}_t, R_t|\mathbf{S}_{t-1}, A_{t-1})$, where the history H_t now includes past states, actions, and rewards.

Formally, an MDP is defined by a tuple $\langle \mathbf{S}, \mathbf{A}, P, R, \gamma \rangle$ with

- set of states $\mathbf{S}$.
- set of actions $\mathbf{A}$.
- transition function $P : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \to [0,1]$; $P^{\mathbf{sas}'} = p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$, the probability that the system transitions to state $\mathbf{s}' \in \mathbf{S}$ from state $\mathbf{s}$ when action $\mathbf{a} \in \mathbf{A}$ is taken.
- reward function $R : \mathbf{S} \times \mathbf{A} \to \mathbb{R}$, giving the expected reward $R^{\mathbf{sa}}$ when taking action $\mathbf{a}$ in state $\mathbf{s}$.
- discount factor $\gamma \in [0,1]$, which trades off the importance of future rewards versus that of immediate reward.

The aim is to find the sequence of actions that maximizes the return R_t, the discounted sum over rewards:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{T} \gamma^k r_{t+k} \tag{2.4}$$

where r_t is the immediate reward received at time t, and T is a time horizon. In the most general case, the horizon can be either finite or infinite; for a finite horizon of $T = 10$, for example, a solution would only "look ahead" ten steps, incurring a return that is maximal for the next ten time steps. In this thesis, however, we focus purely on the infinite horizon case, that is, $T = \infty$. Nevertheless, the discount factor γ is analogous to a time horizon, given a threshold value for discarding future rewards of which the discounted value is close enough to zero; for example, from (2.4) it follows that for $\gamma = 0$, an optimal solution is "myopic" and maximizes only immediate reward.

MDPs can be further divided into **episodic** and **continuing** MDPs. An episodic MDP contains an **absorbing state**; once the system enters such a state, it stays there indefinitely. Formally, $P^{\mathbf{s}^* a \mathbf{s}^*} = 1$ for absorbing state $\mathbf{s}^*$ and any action $\mathbf{a}$; furthermore, $R^{\mathbf{s}^* a} = 0$ for all $\mathbf{a}$. For example, many games can be formulated as episodic MDPs; any state at which the game ends, for example a checkmate in chess, transitions to the absorbing state. A continuing MDP is, for the purposes of this thesis, one that is not episodic.

Because of the Markov property, a solution to an MDP need only be a mapping from each state to the action(s) to take in that state, called a **policy**. In general, a policy π is a function $\pi : \mathbf{S} \times \mathbf{A} \to [0,1]$; $\pi(\mathbf{s},\mathbf{a}) = p(\mathbf{a}|\mathbf{s})$, the probability of taking action $\mathbf{a}$ in state $\mathbf{s}$. An optimal policy π^* achieves maximal *expected* return; that is, if $\mathrm{E}[R_t|\pi]$[1] is the expected return from time t under policy π, then $\mathrm{E}[R_t|\pi^*] \geq \mathrm{E}[R_t|\pi']$, for all t and π'. For MDPs, there always exists a *greedy* optimal policy, that is, one that chooses the best action with probability 1 (breaking ties evenly when there is more than one best action). *Soft policies*, on the other hand, assign a nonzero probability to suboptimal actions.

While there always exists a deterministic greedy optimal policy for MDPs, this may not hold for partially observable MDPs (POMDPs). As the name suggests, in a POMDP the state $\mathbf{s}$ is not fully observable: part of it is revealed in an *observation* $\mathbf{o}$, and part of it is hidden. Therefore, the Markov property may be lost – for example, if the angular velocity in the pendulum system is hidden. Many methods for solving POMDPs attempt to recover the Markov property by constructing a *belief state*, which represents a Bayesian belief about the true underlying state and implicitly factors in the state history [61]. Policies based on the belief state are *non-stationary* with respect to the observation; they may take different actions given the same observation if the belief about the underlying state has changed. Singh et al. [124] propose POMDP solution methods that do not incorporate history and learn stationary policies; they further show that for POMDPs, the best stationary policy might be stochastic. We briefly return to this issue in section 3.3.5, where we discuss a single stationary policy across multiple tasks.

For now, we stick to MDPs. Whenever we mention an optimal policy π^*, we mean the deterministic greedy optimal policy for an MDP, unless mentioned otherwise. The next section details strategies for computing π^*.

2.3 Solving Markov Decision Processes

There are two approaches for computing an optimal policy: *value-function* methods and *policy-search* methods. The latter search directly in policy space, either by estimating a gradient of the return with respect to the policy parameters [64, 113, 137], or by more

[1] $\mathrm{E}[R_t|\pi] = E_\pi[R_t] = \sum_{R_t} p(R_t|\pi) R_t$

direct search methods such as evolutionary algorithms [93, 154]. Value-function methods estimate the value of each state or state-action pair, where value equals the return to be expected from that state(-action pair) given a policy π. A better policy is then constructed by choosing the action that maximizes value for each state. All value function methods explicitly or implicitly iterate **policy evaluation** (estimation of the value of the policy) and **policy improvement** until convergence.

The following subsection explains how value is calculated given a policy; the sections thereafter address policy improvement, reinforcement learning, and policy gradient algorithms.

2.3.1 Value Functions

Unless specified otherwise, we will assume countable state and action spaces, which allows table lookup of state(-action) values. We will refer to table-based value functions and algorithms as *tabular*. The value V of state $\mathbf{s}$ under policy π is the expected return from $\mathbf{s}$ when following π:

$$V^{\pi}(\mathbf{s}) = \mathrm{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, \mathbf{s}_t = \mathbf{s}\right].$$

Using the definition of the reward and transition functions of an MDP, one can calculate the expected immediate reward at t by summing over all possible actions and the expected reward $R^{\mathbf{sa}}$ associated with each action. The expected return equals the expected immediate reward plus the expected return at the next time step, which is, by the definition of value, the weighted (by the transition probabilities $P^{\mathbf{sas}'}$) average of the values of all possible next states, discounted by γ. We therefore obtain:

$$V^{\pi}(\mathbf{s}) = \sum_{a \in \mathbf{A}} \pi(\mathbf{s}, \mathbf{a}) \left[R^{\mathbf{sa}} + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P^{\mathbf{sas}'} V^{\pi}(\mathbf{s}')\right], \tag{2.5}$$

known as the *Bellman equation* [8], which is central to optimal control and reinforcement learning.

Similarly, the value Q of state-action pair $(\mathbf{s}, \mathbf{a})$ equals the expected immediate reward when taking $\mathbf{a}$ in $\mathbf{s}$ plus the expected return at the next time step when following π:

$$\begin{aligned} Q^{\pi}(\mathbf{s}, \mathbf{a}) &= R^{\mathbf{sa}} + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P^{\mathbf{sas}'} V^{\pi}(\mathbf{s}') \\ &= R^{\mathbf{s},a} + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P^{\mathbf{sas}'} \sum_{a' \in \mathbf{A}} \pi(\mathbf{s}', a') Q^{\pi}(\mathbf{s}', a'). \end{aligned} \tag{2.6}$$

For an optimal policy π^*, it follows that $V^{\pi^*}(\mathbf{s}) \geq V^{\pi'}(\mathbf{s})$, for all $\mathbf{s}$ and π'. Note that there can be more than one policy that is optimal; however, they all share the same optimal value function $V^* = V^{\pi^*}$. An optimal policy takes an action with maximum expected return for each state; therefore, $V^*(\mathbf{s}) = \max_{a \in \mathbf{A}} Q^*(\mathbf{s}, \mathbf{a})$, and

$$Q^*(\mathbf{s}, \mathbf{a}) = R^{\mathbf{sa}} + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P^{\mathbf{sas}'} \max_{a' \in \mathbf{A}} Q^*(\mathbf{s}', a'), \tag{2.7}$$

the *Bellman optimality equation*.

2.3.2 Dynamic Programming

Dynamic programming methods compute V^* and π^* given a complete MDP model $\langle \mathbf{S}, \mathbf{A}, P, R, \gamma \rangle$. Given an initial policy π, the general idea behind these methods is to iteratively evaluate (compute the value of) π and improve π based on the new value function, until convergence. Policy improvement occurs by "greedifying" the policy with respect to the new value function: for every state, the new policy chooses the greedy action, that is, the action that maximizes the newly estimated value. This idea is exemplified in the *policy iteration* algorithm (Algorithm 1).

Algorithm 1 Policy Iteration

Initialize $V(\mathbf{s})$ and $\pi(\mathbf{s})$ arbitrarily, for all $\mathbf{s}$

//Policy Evaluation
repeat
  $\Delta \leftarrow 0$
  for all $\mathbf{s} \in \mathbf{S}$ **do**
    $v \leftarrow V(\mathbf{s})$
    $V(\mathbf{s}) \leftarrow R(\mathbf{s}, \pi(\mathbf{s})) + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P(\mathbf{s}, \pi(\mathbf{s}), \mathbf{s}') V(\mathbf{s}')$
    $\Delta \leftarrow \max(\Delta, |v - V(\mathbf{s})|)$
  end for
until $\Delta < \theta$ (a threshold value)

//Policy Improvement
$policystable \leftarrow true$
for all $\mathbf{s} \in \mathbf{S}$ **do**
  $b \leftarrow \pi(\mathbf{s})$
  $\pi(\mathbf{s}) \leftarrow \text{argmax}_a [R(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P(\mathbf{s}, a, \mathbf{s}') V(\mathbf{s}')]$
  if $b \neq \pi(\mathbf{s})$ **then** $policystable \leftarrow false$
end for
if *policystable* **then** stop; **else** go to 3

In the policy evaluation step, policy iteration approximates the value of the current policy until convergence, which might require several passes (sweeps) through the whole state space. Often, however, the policy can be improved after just one sweep of policy evaluation. *Value iteration* effectively combines one step of policy evaluation and one of policy improvement into a single sweep, as detailed in Algorithm 2. All value-function methods work by the principle of *generalized policy iteration*, alternating or otherwise interleaving steps of policy evaluation and improvement until convergence.

Dynamic programming methods require the full MDP model. Often, such a model is not available and would be impossible or impractical to calculate. The next section provides background on learning algorithms that do not require a model to reach a solution.

Algorithm 2 Value Iteration

Initialize $V(\mathbf{s})$ arbitrarily, for all $\mathbf{s}$
repeat
  $\Delta \leftarrow 0$
  for all $\mathbf{s} \in \mathbf{S}$ **do**
    $v \leftarrow V(\mathbf{s})$
    $V(\mathbf{s}) \leftarrow \max_a [R(\mathbf{s},\mathbf{a}) + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} P(\mathbf{s},a,\mathbf{s}')V(\mathbf{s}')]$
    $\Delta \leftarrow \max(\Delta, |v - V(\mathbf{s})|)$
  end for
until $\Delta < \theta$ (a threshold value)

2.3.3 Reinforcement Learning

Reinforcement learning (RL) methods are based on interaction of an agent with an initially unknown MDP, called the environment. The agent perceives the environment through the state $\mathbf{s}$, and acts upon it through one of the actions from $\mathbf{A}$. After each action, the agent receives a reward r and the environment transitions to the next state $\mathbf{s}'$, after which the process repeats.

Since agents can only learn about the environment through trial and error, they face an *exploration/exploitation dilemma*: at any given time step, the agent needs to decide whether it chooses the action that is currently estimated as best for the present state (exploitation), or chooses to gain more information about the environment by picking a different action that might lead to higher return, but might also have negative consequences (exploration). Various behavioral strategies are available to deal with the exploration/exploitation dilemma; this thesis mainly employs ε-*greedy* behavior policies, which choose the action that is currently estimated as best with probability $1-\varepsilon$ and a random action with probability ε. Of course, the application of prior knowledge from previously experienced tasks, the main subject of this thesis, also guides exploration.

Thus, the agent employs a specific policy for exploring the environment, but at the same time tries to learn an optimal policy, which is by definition greedy. Hence, two kinds of policies may be distinguished [135]: the *estimation policy*, which the agent is learning about; and the *behavior policy*, which is used for interaction with the environment, i.e., for generating experience samples. For example, the agent could employ a random behavior policy to generate samples, but from these samples learn a correct solution to the MDP, i.e., a correct estimation policy. Whether such an approach is viable depends on the type of learning algorithm used: *on-policy* methods learn about the agent's behavior policy, that is, the estimation policy equals the behavior policy. In contrast, in *off-policy* methods the estimation policy differs from the behavior policy; such methods can learn the optimal policy even though their behavior policy is not greedy. We will shortly see two concrete examples of these two different methods.

In RL problems, no MDP model is available and the agent must learn about its environment by interacting with it. *Model-based* RL methods do so by learning a model of the environment and improving the policy through dynamic programming on the estimated model [11, 136]; in contrast, *model-free* methods, which this thesis employs, estimate a value function or policy directly from interaction with the environment [135].

A widely used class of model-free methods is *temporal-difference* (TD) learning [134]. When used for control, the TD update takes the form

$$Q(\mathbf{s},\mathbf{a}) \leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha\delta, \tag{2.8}$$

where α is a learning rate and δ the *TD error*. There are various approaches to computing the latter, depending on which TD control algorithm is used. Two of the most popular are Sarsa [107, 121], and Q-learning [150]. For Sarsa,

$$\delta = r + \gamma Q(\mathbf{s}',a') - Q(\mathbf{s},\mathbf{a}), \tag{2.9}$$

where r is the reward incurred, $\mathbf{s}'$ the next state and a' the action the agent took at $\mathbf{s}'$. For Q-learning,

$$\delta = r + \gamma \max_{a^*} Q(\mathbf{s}',a^*) - Q(\mathbf{s},\mathbf{a}). \tag{2.10}$$

Algorithm 3 Sarsa

Initialize $Q(\mathbf{s},\mathbf{a})$ arbitrarily, for all $\mathbf{s},a$
Initialize $\mathbf{s}$
Choose action $\mathbf{a}$ from $\mathbf{s}$ based on Q
loop
    Take action $\mathbf{a}$, observe r and $\mathbf{s}'$
    Choose next action a' from $\mathbf{s}'$ based on Q
    $Q(\mathbf{s},\mathbf{a}) \leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha\,[r + \gamma Q(\mathbf{s}',a') - Q(\mathbf{s},\mathbf{a})]$
    $\mathbf{s} \leftarrow \mathbf{s}'$; $a \leftarrow a'$
end loop

Algorithm 4 Q-Learning

Initialize $Q(\mathbf{s},\mathbf{a})$ arbitrarily, for all $\mathbf{s},a$
Initialize $\mathbf{s}$
loop
    Choose action $\mathbf{a}$ from $\mathbf{s}$ based on Q
    Take action $\mathbf{a}$, observe r and $\mathbf{s}'$
    $Q(\mathbf{s},\mathbf{a}) \leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha\,[r + \gamma \max_{a^*} Q(\mathbf{s}',a^*) - Q(\mathbf{s},\mathbf{a})]$
    $\mathbf{s} \leftarrow \mathbf{s}'$
end loop

Algorithm 3 and 4 detail the Sarsa and Q-Learning algorithms. As explained earlier, the main difference between Sarsa and Q-Learning is the difference in TD error estimation: Sarsa uses the next action selected by the behavior policy, while Q-Learning uses the maximum Q-value of the next state, as if the agent were following a greedy behavior policy. This makes Sarsa an on-policy algorithm, and Q-Learning an off-policy one. As a result, Q-Learning agents may employ a wide range of behavior policies, ensuring sufficient exploration, while still converging to the optimal Q-values (belonging, by definition, to an optimal greedy policy), under mild assumptions [150]. Sarsa, on the other

hand, cannot converge to the optimal Q-values while exploring, since an optimal policy is by definition greedy. However, Sarsa may have an advantage over Q-Learning, particularly in scenarios where online return is important. Because Q-Learning implicitly assumes the agent is following a greedy policy, and therefore does not take the randomness from exploration into account, it may be overly optimistic. The cliff-walking domain in the next chapter provides a practical example of this difference between the two methods.

2.3.4 Vanilla Policy Gradient

The preceding sections described techniques for solving MDPs using value functions. When the MDP has continuous state and action spaces, such as for example in robotics applications, the value function-based approach can be problematic. For example, it might be difficult to compute $\text{argmax}_{\mathbf{a}\in\mathbb{R}^d} Q(\mathbf{s},\mathbf{a})$, where d is the action dimensionality. Therefore, in this setting, it is often simpler to parameterize the policy π_θ directly with parameter vector θ, and update the parameters based on the gradient of the return with respect to the parameters.

The problem now boils down to finding the optimal parameter vector θ^*:

$$\theta^* = \underset{\theta}{\text{argmax}}\, \mathrm{E}[R(\tau)|\pi_\theta],$$

where τ is a trajectory $(\mathbf{s}_0, \mathbf{a}_0, r_0, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T)$, generated by running π_θ on the MDP, and $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$, as before.

The value of a given θ is then

$$V(\theta) = \mathrm{E}[R(\tau)|\pi_\theta] = \int p(\tau|\theta) R(\tau) d\tau.$$

Policy gradient (PG) methods compute the gradient $\nabla_\theta V(\theta)$, after which one can use a gradient ascent algorithm to approximate θ^*. Of course, most work goes into, and different PG flavors arise from, derivation of estimators for this gradient. Some key points underlying a number of PG algorithms in general and the PG algorithm that this thesis employs in particular, are the following.

1. From the definition of an MDP, the likelihood $p(\tau|\theta)$ is given by

   $$p(\tau|\theta) = p(s_0) \prod_{t=0}^{T} p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \pi_\theta(\mathbf{a}_t|\mathbf{s}_t),$$

   which involves both a sizable product term and the system dynamics $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, which model-free RL methods do not have access to. The following paragraphs will show how both these problems disappear through the use of a stochastic policy combined with the *score function* estimator, also known as the *likelihood ratio* estimator [41]. It also has a number of other useful properties, such as being unbiased, and allowing sample-based approximations to $\nabla_\theta V(\theta)$.

2. In addition to ensuring system dynamics are not needed in the definition of the policy gradient, stochastic policies enable gradient-based methods for discrete action spaces. Since the policy is stochastic, no separate exploration mechanism is used.

3. The integral in the gradient expression is usually approximated using Monte-Carlo rollouts. Particularly in the case of large T (long rollouts), the approximation can suffer from high variance. One of the main mechanisms for variance reduction employed by PG methods is to replace $R(\tau)$ with the *advantage function* $A^\pi(\mathbf{s},\mathbf{a}) = Q^\pi(\mathbf{s},\mathbf{a}) - V^\pi(\mathbf{s})$; that is, A^π gives an indication of whether the action $\mathbf{a}$ is better or worse than average, under π.

Application of the above points yields the *vanilla policy gradient* algorithm. The state-of-the-art makes further improvements on this algorithm, mainly with respect to how optimization on the gradient is carried out. This will be explained later in the section; first, we will go through each of the above points in some more detail, although a full derivation is beyond the scope of this thesis.

The score function is the gradient of the log-likelihood with respect to θ: $\nabla_\theta \log p(x|\theta) = \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}$. This identity can be used to derive[2] the following expression for $\nabla_\theta V(\theta) = \nabla_\theta \mathrm{E}[R(\tau)|\theta]$:

$$\begin{aligned}\nabla_\theta \mathrm{E}\left[R(\tau)|\theta\right] &= \nabla_\theta \int p(\tau|\theta) R(\tau) d\tau \\ &= \int \nabla_\theta p(\tau|\theta) R(\tau) d\tau = \int p(\tau|\theta) \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)} R(\tau) d\tau \\ &= \int p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) R(\tau) d\tau = \mathrm{E}\left[R(\tau) \nabla_\theta \log p(\tau|\theta)\right].\end{aligned}$$

Expanding the log-likelihood gives

$$\begin{aligned}\nabla_\theta \log p(\tau|\theta) &= \nabla_\theta \left[\log p(\mathbf{s}_0) + \sum_{t=0}^{T} \log p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t) + \sum_{t=0}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right] \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t),\end{aligned}$$

where the last line follows because the other terms do not depend on θ. Note that this would not happen if the policy were deterministic, since then we would have $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \pi_\theta(\mathbf{s}_t))$, introducing a dependency of the system dynamics on θ. However, see Silver et al. [119] for a different derivation that arrives at deterministic policy gradient algorithms (DPG). Therefore,

$$\nabla_\theta V(\theta) = \mathrm{E}\left[R(\tau) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right], \tag{2.11}$$

which can be approximated using sample averages. A central issue is that the variance of these approximations tends to be high; for example, note the multiplication with the sum of rewards of the full path $R(\tau)$. Three important variance reduction techniques (where the first two come at the cost of increased bias) are to

1. Use discounting.

[2] The exchange of the order of differentiation and integration in line 2 is the operation that requires $p(\tau|\theta)$ to be continuous in θ.

2. For each t, only use the (discounted) return from t onwards, since naturally a policy decision at t does not influence rewards before t.

3. Subtract a baseline from $R(\tau)$, for example $\langle R(\tau)\rangle$, the average over collected trajectories. I.e., this indicates how a given trajectory compares to the average. In general, the baseline can depend on the state, and a near-optimal baseline is the state-value function $V^{\pi}(\mathbf{s})$ [44].

Putting all these together, we replace $R(\tau)$ with the *advantage estimate*

$$\hat{A}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(\mathbf{s}_t), \tag{2.12}$$

where $b(\cdot)$ is the baseline. As an interesting side remark, note that when the baseline is the state-value function this can also be written as $r_t + \gamma \sum_{t'=t+1}^{T} \gamma^{t'-t+1} r_{t'} - V(\mathbf{s}_t) = r_t + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)$: the TD error.

Putting everything together, the vanilla policy gradient estimate is given by

$$\hat{\mathbf{g}}_V = \left\langle \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \hat{A}_t \right\rangle, \tag{2.13}$$

where $\langle\rangle$ is the average over trajectories. The vanilla policy gradient (VPG) algorithm is given in Algorithm 5.

Algorithm 5 Vanilla policy gradient

1: Initialize policy parameters θ, baseline b
2: **loop**
3: Collect a set of trajectories by executing the current policy π_θ
4: At each timestep t in each trajectory, compute
5: the return $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
6: the advantage estimate $\hat{A}_t = R_t - b(\mathbf{s}_t)$
7: Update the baseline by regression on R_t: minimize $||R_t - b(\mathbf{s}_t)||^2$, summed over all trajectories and timesteps
8: Update the policy using gradient ascent on the vanilla policy gradient estimate $\hat{\mathbf{g}}_V$
9: **end loop**

2.3.5 Trust Region Policy Optimization

This section introduces the state-of-the-art PG algorithm that will be employed in chapter 5 of the thesis: Trust Region Policy Optimization (TRPO) [114]. The main difference with VPG is the significantly more sophisticated optimization machinery to update the policy based on the policy gradient. TRPO has been shown to significantly outperform a number of other PG methods on a benchmark set of continuous control tasks [27]; these same benchmark tasks will be employed in Chapter 5.

A central issue in RL that does not occur in supervised learning is that the data used for optimization depends on the quantity being optimized (the policy). Therefore, taking too large a gradient step could induce a "bad" policy, which will in turn generate bad data for the next optimization step. It is typically difficult to recover from this. One could take tiny gradient steps, but this would make the optimization process very inefficient. Ideally, then, the step size should be as large as possible within certain bounds on how much the policy can change.

TRPO and a class of related algorithms called *natural policy gradient* [62] algorithms restrict the policy change by introducing a constraint on the divergence between the probability distributions induced by the old and the new policy; typically, this divergence is measured using the Kullback-Leibler divergence $D_{KL}(p||q) = \int p(x) \log \frac{p(x)}{q(x)} dx$, where p and q are probability densities.

Specifically, TRPO solves the following constrained optimization problem at each iteration k:

$$\begin{aligned} \text{maximize}_\theta \qquad & L_{\theta_{k-1}}(\theta) = \left\langle \sum_{t=0}^{T} \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta_{k-1}}(\mathbf{a}_t|\mathbf{s}_t)} \hat{A}_t \right\rangle \\ \text{s.t.} \qquad & \left\langle D_{KL}(\pi_\theta(\cdot|\mathbf{s})||\pi_{\theta_{k-1}}(\cdot|\mathbf{s})) \right\rangle \leq \delta_{KL}, \end{aligned} \tag{2.14}$$

where δ_{KL} is a tuning parameter of the algorithm. In summary, it makes the following two main changes with respect to VPG:

1. It replaces the log-likelihood with an importance sampling ratio.
2. It places a hard constraint on the maximum KL divergence δ_{KL} between the old and new policy.

Other classes of algorithms, such as the natural policy gradient mentioned earlier, can be shown to be special cases of the TRPO formulation [114]. One main difference between these two algorithms is that TRPO uses a fixed, hard constraint on δ_{KL}, while Natural Policy Gradient uses a penalty. This turns out to make a significant difference [112], and is where TRPO derives its name from: it restricts the search for θ^* to a trust region around θ_{k-1}[3]. (Recently however, Schulman et al. [115] proposed an improvement to TRPO called proximal policy optimization (PPO), which uses a soft constraint that is simpler to compute.)

In practice, TRPO uses a quadratic approximation to D_{KL}:

$$D_{KL} \approx \Delta\theta^\top \mathbf{F}_\theta \Delta\theta,$$

where $\Delta\theta$ is the change in parameters and $\mathbf{F}$ is the *Fisher information matrix*

$$F_\theta = \left\langle \nabla_\theta \log p(\tau|\theta) \nabla_\theta \log p(\tau|\theta)^\top \right\rangle.$$

The Natural Policy Gradient estimate is then given by [100]

$$\hat{\mathbf{g}}_N = \mathbf{F}^{-1} \hat{\mathbf{g}}_V, \tag{2.15}$$

where TRPO uses $\nabla_\theta L_{\theta_{k-1}}(\theta)$ instead of $\hat{\mathbf{g}}_V$, and uses the conjugate gradient algorithm with Hessian vector products to approximate the inversion $\mathbf{F}^{-1}$. The algorithm is shown in Algorithm 6.

[3] See Nocedal & Wright [96]for a general description of trust region optimization methods.

Algorithm 6 Trust Region Policy Optimization

1: Initialize policy parameters θ, baseline b
2: **loop**
3: Collect a set of trajectories by executing the current policy π_θ
4: At each timestep t in each trajectory, compute
5: the return $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
6: the advantage estimate $\hat{A}_t = R_t - b(\mathbf{s}_t)$
7: Update the baseline by regression on R_t: minimize $||R_t - b(\mathbf{s}_t)||^2$, summed over all trajectories and timesteps
8: Use the conjugate gradient algorithm with Hessian vector products to compute the step direction $\hat{\mathbf{g}}_N$
9: Do a line search on $L_{\theta_{k-1}}(\theta)$ and D_{KL} to determine the step size
10: **end loop**

This concludes the background on MDPs and reinforcement learning methods. The next section provides background on (recurrent) neural networks, which will be used as policies in Chapter 5.

2.4 Recurrent Neural Networks

A neural network is a computation graph organized into layers of nodes, where each node computes a (usually non-linear) transformation of its inputs. The feedforward pass from one layer to the next can typically be written as

$$\mathbf{o} = \mathbf{f}(\mathbf{W}\mathbf{x}),$$

where $\mathbf{x}$ is an n-dimensional vector of inputs (here taken to include the bias, which is always 1), $\mathbf{W}$ is an $m \times n$-dimensional matrix of connection weights where m equals the number of nodes in the layer, $\mathbf{f}$ is a *transfer* or *activation* function, and $\mathbf{o}$ is the resulting m-dimensional output vector. Often, $\mathbf{f}$ takes the form of a saturating nonlinearity such as the hyperbolic tangent, which is the transfer function we will use in this thesis unless specified otherwise. Layers i and j, $j > i$, can be chained together by setting $\mathbf{x}_j = \mathbf{o}_i$. Layers in between the network input and output are called *hidden* layers. The left panel of Fig. 2.1 shows the computation graph for one layer, where we have decomposed it into $\mathbf{o} = \mathbf{f}(\mathbf{z})$ and $\mathbf{z} = \mathbf{g}(\mathbf{x}) = \mathbf{W}\mathbf{x}$.

Commonly, feedforward neural networks (FNNs) are trained to approximate a given target function. In order to do so, a loss function is defined to calculate the error on data sample i between the FNN's prediction $\hat{\mathbf{y}}_i$ and the desired output $\mathbf{y}_i$, for example the mean-squared error $L = \frac{1}{N}\sum_i^N \frac{1}{2}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$. The FNN is then trained using some variant of the backpropagation (BP) algorithm [106, 151] (see also Schmidhuber [111] for a detailed account of BP history). BP calculates the gradients of a layer's weights with respect to the error by using the chain rule from calculus to propagate the error of a network's output layer back through the hidden layers in the reverse order of the feedforward pass, akin to dynamic programming algorithms.

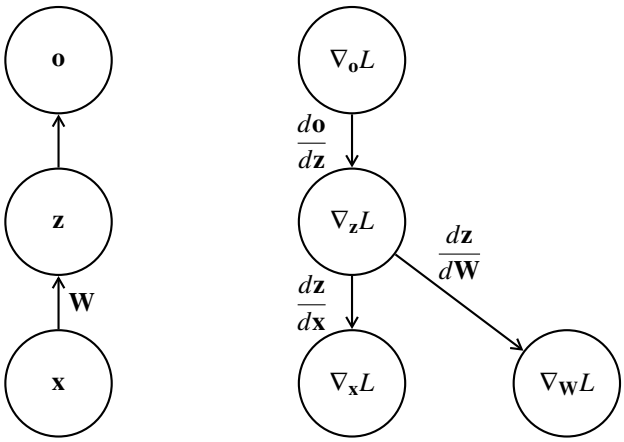


Figure 2.1: (a) Computation graph corresponding to one layer in a feedforward neural net. (b) The backward pass corresponding to chain rule operations.

This is demonstrated in Fig. 2.1b. Note that the derivatives on the edges can be matrices or tensors (arrays with more than two axes). For example[4],

$$\nabla_{\mathbf{W}} L = \frac{d\mathbf{z}}{d\mathbf{W}} \nabla_{\mathbf{z}} L = (\nabla_{\mathbf{z}} L)\, \mathbf{x}^{\top}$$

and similarly for the other nodes. Weight updates take place by taking a small step down the gradient, $\Delta W = -\eta \nabla_{\mathbf{W}} L$, where η is a step size.

A *recurrent* neural network (RNN) introduces recurrent connections, typically on the hidden layer, and thereby becomes a nonlinear dynamical system. One of the simplest examples is a vanilla discrete-time RNN (DTRNN), also known as an Elman network [30], which is the map

$$\mathbf{h}_n = \mathbf{f}(\mathbf{K}\mathbf{h}_{n-1} + \mathbf{W}\mathbf{x}_n),$$

where $\mathbf{K}$ is a $p \times p$ square matrix of recurrent connections, and $\mathbf{h}_n$ is the p-dimensional output of the hidden layer at update n. See Figure 2.2a.

An RNN introduces dependencies that extend through time as well as through layers, since the layer at the end of a recurrent connection uses the outputs of the source of the connection at the previous timestep. Weight gradients can be calculated using the backpropagation through time (BPTT) algorithm [152]. Conceptually, BPTT "unrolls" the cycles in the RNN graph by representing each time step as a layer. Using this representation, we once again have a directed acyclic computation graph on which we can run BP. Figure 2.2b illustrates the BPTT concept for calculation of the gradient of the input weights with respect to the loss, $\nabla_{\mathbf{W}_i} L$. As with regular BP, errors are propagated backwards through the layers, but BPTT also backpropagates through timesteps, introducing

[4]Taking some notational shortcuts for brevity, the product of the tensor A with the vector $\mathbf{v}$ is taken to be the tensor dot product resulting in matrix $\mathbf{B}$ such that $B_{i,j} = \sum_k \mathsf{A}_{i,j,k} v_k$.

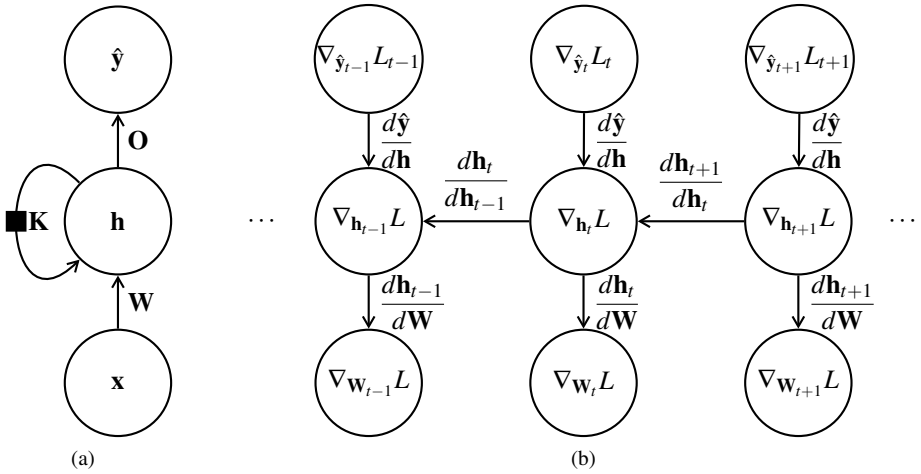


Figure 2.2: (a) A simple recurrent neural network, where a black square represents a one-step time delay. (b) Slice of the unrolled graph representing the BPTT computation on this network (not all computations are shown). Derivatives are propagated in the reverse direction of the forward pass.

an edge between $\mathbf{h}_t$ and $\mathbf{h}_{t-1}$. For example,

$$\nabla_{\mathbf{h}_{t-1}} L = \left(\frac{d\hat{\mathbf{y}}}{d\mathbf{h}}\right)^\top \nabla_{\hat{\mathbf{y}}_{t-1}} L_{t-1} + \left(\frac{d\mathbf{h}_t}{d\mathbf{h}_{t-1}}\right)^\top \nabla_{\mathbf{h}_t} L.$$

Training continuous-time RNNs, used in Chapter 5, works according to the same principles, but calculates the gradient using differential equations; see Pearlmutter [98] for an overview.

3

Learning Potential Functions for Multi-Task RL

Traditionally, the main aim of the reinforcement-learning algorithms discussed in the previous chapter has been to converge to the optimal policy, which maximizes expected return. However, due to the curse of dimensionality, delayed reward, and the exploration / exploitation dilemma, on some problems a naive RL agent can take a long time to converge to a solution. Expected *online return* (return incurred while the agent is learning and interacting with the environment), rather than convergence, is often equally or more important. The application of prior knowledge can help improve return both by guiding the agent's exploration strategy (for example, by suggesting actions to take) and reducing the related problem of delayed reward (by providing information on reward where in the original task none is given). In real-world scenarios, agents of any kind frequently have to deal with multiple related tasks. The detection of common patterns in these tasks is therefore a natural way of deriving prior knowledge. Methods that transfer knowledge from one task to another are called *transfer learning* methods.

In this chapter and the next, we consider how an RL agent facing a sequence of tasks can best exploit its experience with previous tasks to improve its online return on new tasks, via two complementary approaches. The first approach extracts knowledge from previous experience in a way that can be leveraged by learning algorithms; this is the subject of this chapter. The second approach automatically discovers good representations for the extracted knowledge; for example, a subset of the agent's sensory features. This is the subject of the next chapter.

Transfer learning aims to improve performance on a set of *target tasks* by leveraging experience from a set of *source tasks*. Clearly, the target tasks must be related to the source tasks for transfer to have an expected benefit. In *multi-task reinforcement learning* (MTRL), this relationship is formalized through a *domain*, a distribution over tasks from which the source and target tasks are independently drawn [140, 155]. Even so, relatedness can come in different forms. Here, we focus on discovering shared representational structure between tasks and the information captured by that structure. Roughly speaking, shared information between tasks means that the meaning of the information does not change from one task to the next.

This decomposes the multi-task learning problem into two separate (but related) subproblems: to learn a good policy for each separate task, and to capture which information

does not change meaning from one task to the next. For example, consider a robot-navigation domain. In each task, the robot is placed in a different building in which it must locate a bomb. An optimal policy for a given task need condition only on the robot's position in the building. However, in each task, the robot must re-learn how position relates to the bomb's location; the meaning of a particular position changes from one task to the next. Since position is useful within tasks but not across tasks, we call it a *task-relevant* feature. By contrast, while distance sensors are not needed in a task-specific policy, they can be used to represent the task-invariant knowledge that crashing into obstacles should be avoided; we call such features *domain relevant*. While the robot must learn a new value function for each task, task-invariant knowledge could be captured in a single *cross-task function* or rule set, for example. This chapter and the next focus on a scenario in which an agent, such as the robot, faces one task after the other; for example, one building after the other. The goal is to update the cross-task function and its representation after incorporating the knowledge from each newly observed task.

Some approaches to multi-task learning learn a single function that may combine task-specific and domain-wide information [2, 6, 14, 74, 143]. For example, in supervised learning, instead of training a separate neural network per learning task, Caruana [14] trains a single network on all tasks in parallel. Learning benefits from the shared hidden layer between tasks, which captures task-invariant knowledge, but the network can also represent task-specific solutions through the separate task weights. However, such approaches have several limitations. First, task solutions may interfere with each other and with the task-invariant knowledge [15]. Second, any penalty term for model complexity [2, 74] affects both task and domain-relevant representations simultaneously. Lastly, mapping the original problem representation to a new one makes it harder to interpret the solutions and decipher which knowledge is task-invariant.

These problems can be avoided by maintaining separate functions for task-specific and task-invariant knowledge. In each task, the agent learns a task-specific policy while aided by a cross-task function that represents task-invariant knowledge. This approach also makes it easier to override the bias of the cross-task function, which is important when that bias proves incorrect [15]. The cross-task function can be represented as, e.g., advice rules [145], or a *shaping function* [69, 94], the approach we focus on in this chapter.

Shaping functions augment a task's reward function with additional informative artificial rewards, and are typically functions of state, state-action pairs, or transitions. They have proven effective in single-task [94], multi-agent [4] and multi-task [69, 126, 127] problems, with applications to, e.g., foraging by real robots [28], robot soccer [21], and real-time strategy games [90]. In MTRL, shaping functions can be learned automatically by deriving them from the source tasks [69, 126, 127], e.g., the navigating robot described above could learn a shaping function that discourages bumping into obstacles.

The primary aim of this chapter is to address a key issue that arises when deriving shaping functions in MTRL: What target should the function approximate? This question does not arise in supervised learning, since the targets are given and thus any single cross-task function strives to minimize a loss function with respect to all tasks' targets simultaneously. An intuitive choice of target in MTRL is the optimal value function of each task; approximating this target leads to the solution that is closest in expectation to the optimal solution of the unknown next task. However, there is no guarantee that using

a shaping function that approximates this target leads to the best online return.

After providing additional background on shaping functions and outlining the particular problem setting we are concerned with in sections 3.1 and 3.2, we propose three different targets for the shaping function in section 3.3. Furthermore, in section 3.4, we show empirically that which one is best depends critically on the domain and learning parameters.

3.1 Shaping

The concept of shaping stems from the field of operant conditioning, where it denotes a training procedure of rewarding successive approximations to a desired behavior [125]. In RL, it may refer either to training the agent on successive tasks of increasing complexity, until the desired complexity is reached [32, 49, 104, 108, 116, 123], or, more commonly, to supplementing the MDP's reward function with additional, artificial rewards [3, 23, 28, 46, 72, 86, 88, 94, 153]. This chapter employs shaping functions in the latter sense.

Because the shaping function modifies the rewards the agent receives, the shaped agent may no longer learn an optimal policy for the original MDP (e.g. [104]). Ng et al. [94] show that, in order to retain the optimal policy, a shaping function should be based on a *potential function* over states. Like a value function, a potential function $\Phi : \mathbf{S} \mapsto \mathbb{R}$ specifies the desirability of a given state. Potential-based shaping functions take the form $F^{\mathbf{s}}_{\mathbf{s}'} = \gamma\Phi(\mathbf{s}') - \Phi(\mathbf{s})$; hence, a positive reward is received when the agent moves from a low to a high potential, and a negative reward when moving in the opposite direction. Similarly to potentials in physics and potential field methods in mobile robot navigation [70], a shaping potential thus results in a "force" encouraging the agent in a certain "direction".

State potential functions might miss additional information provided by the actions for a state. To address this, Wiewiora et al. [157, 158] introduced shaping functions of the form $F^{\mathbf{s}a}_{\mathbf{s}'a'} = \gamma\Phi(\mathbf{s}', a') - \Phi(\mathbf{s}, a)$, where a' is defined as in the learning rule. In this form, shaping functions closely resemble advice-giving methods [80, 145] in that they bias an agent's policy towards the actions that the shaping function estimates as most valuable[1]. Wiewiora et al. show that using F is equivalent to initializing the agent's Q-table to the potential function, under the same experience history. However, some important differences remain. Unlike shaping, initialization biases the agent's actions *before* they are taken. In addition, shaping can be applied to RL with function approximation and in cases where the experimenter does not have access to the agent's value function [158]. Either way, the results in this chapter apply equally well to shaping as to initialization methods.

[1] The authors termed these *potential-based advice*; specifically, *look-ahead advice* for the formula introduced here. We use the term "shaping" for both methods, and let function arguments resolve any ambiguity.

3.2 Problem Setting

We define a multi-task *domain* d to be a pair $d = \langle D, \mathsf{M} \rangle$, where D is a distribution over tasks $D(m) = p(m)$, and $D(m) > 0$ for all $m \in \mathsf{M}$, the set of all MDPs in the domain. This definition matches the *generalized environment* proposed in [155]. Since we are now dealing with multiple MDPs, we define an MDP to be a tuple $m = \langle \mathbf{X}_m, P_m, R_m, \gamma \rangle$; i.e., the state-action space $\mathbf{X}_m = \mathbf{S}_m \times \mathbf{A}_m$, transition function P_m, and expected reward function R_m are task-dependent and therefore subscripted by m. The action set $\mathbf{A}$ and the discount factor γ are the same for all tasks in a given domain. The domain state space is $\mathbf{S}_d = \bigcup_{m \in \mathsf{M}} \mathbf{S}_m$, and similarly $\mathbf{X}_d = \bigcup_{m \in \mathsf{M}} \mathbf{X}_m$. As noted previously, we assume the same feature and action set for all tasks. However, results presented here can be extended to different feature and action sets through the use of inter-task mappings: functions that map states and/or actions from one task to the equivalent state and/or action in another (see [140] for an overview of these methods).

Given a domain d and one or more source tasks drawn with replacement from d, the agent's goal is to maximize expected online return on an unknown target task sampled from d; this measure also implicitly captures the time to reach a good policy. In general, we are interested in a scenario in which the agent is "interacting sequentially" with the domain, similar to lifelong learning [143]. That is, starting with an empty history h and potential function $\Phi : \mathbf{X}_d \to 0$, it goes through the following steps:

1. Receive a task m sampled with replacement from the domain according to $D(m)$. The task model is unknown.
2. Learn the solution to m with a model-free learning algorithm, aided by Φ. Add the solution to the solution history h.
3. Update Φ based on h.
4. Go to step 1.

Nonetheless, this chapter applies equally well to batch scenarios in which the agent receives a sequence of source tasks sampled from the domain up front; the solutions to this sequence then just become the history h.

For the moment, we assume the agent has perfect knowledge of each domain and thus computes each potential function using all tasks in the domain. Our goal is to demonstrate the *theoretical* advantages of each type of potential function. Assuming perfect domain knowledge enables comparisons of the potential functions' maximum performance, untainted by sampling error. From Chapter 4 onward, we consider the more realistic setting in which only a sample sequence of tasks is available. In this case, generalization to unseen next tasks, and thus learning a good representation for the potential function, is important.

3.3 Potential Functions for Multi-Task Learning

In this section, we formulate a definition of an optimal potential-based shaping function and, since we cannot solve this expression exactly, derive three approximations to this

function for the learning case. For simplicity, we assume a tabular learning algorithm L. Since we are interested in maximizing online return, an optimal shaping function is one based on a potential function that maximizes expected online return across target tasks:

$$\Phi_L^* = \operatorname*{argmax}_{\Phi} \mathrm{E}\left[R_m|\Phi,L\right] = \operatorname*{argmax}_{\Phi} \sum_{m\in\mathsf{M}} D(m)\mathrm{E}\left[\sum_{t=0}^{\infty}\gamma^t r_{t,m}\middle|\Phi,L\right], \tag{3.1}$$

where R_m is the return accrued in task m and $r_{t,m}$ is the immediate reward obtained on timestep t in task m. Note that the task is essentially a hidden variable since the potential function does not take it as input. Thus the potential function that satisfies (3.1) may perform poorly in some tasks, though it performs best in expectation across tasks.

Since shaping with Φ is equivalent to initializing the Q-table with it, solving (3.1) is equivalent to finding the best cross-task initialization of the Q-table. Unfortunately, because of interacting unknown task and learning dynamics, there is no obvious way to compute such a solution efficiently in the learning case, and search approaches quickly become impractical. However, it is possible to derive a solution for the planning case that provides the lowest bounds on the number of iterations needed to converge.

In the following sections, we first derive an expression for the optimal value table initialization given that the task models are available and solved using value iteration. We show that the optimal initialization in this case minimizes, in expectation, the weighted geometric mean max-norm with the optimal value function in the target task. We then discuss three strategies for efficiently approximating Φ_L^* for the learning case.

3.3.1 Optimal Initialization For Value Iteration

In the planning case, an optimal initialization is one that minimizes the expected number of iterations to solve the target task.

Theorem 1. *The initial value function Q_0^* that in expectation minimizes the number of iterations needed to solve a given task m from a domain d by value iteration is, for $\gamma \in (0,1)$,*

$$\begin{aligned} Q_0^* &= \operatorname*{argmax}_{Q_0} \sum_m D(m) \log_\gamma \|Q_m^* - Q_0\|_\infty \\ &= \operatorname*{argmin}_{Q_0} \log_\gamma \prod_m \|Q_m^* - Q_0\|_\infty^{D(m)} \end{aligned}$$

Proof. By Banach's theorem, the value iteration sequence for a single task converges at a geometric rate [9]:

$$\|Q_m^* - Q_m^n\|_\infty \leq \gamma^n \|Q_m^* - Q_0\|_\infty,$$

where Q_m^n is the value function on task m after n iterations and $\|\cdot\|_\infty$ denotes the max-norm. This equation provides a lower bound on the number of iterations n needed to get within an arbitrary distance of Q_m^*, in terms of γ and the initial value function Q_0. That is, to get within ε of Q_m^*, then $\|Q_m^* - Q_m^n\|_\infty \leq \varepsilon$, which is satisfied if $\gamma^n \|Q_m^* - Q_0\|_\infty \leq \varepsilon$.

Let $\delta_m = \|Q_m^* - Q_0\|_\infty$. Assuming $\delta_m > 0$ and $\varepsilon < \delta_m$, then

$$\begin{aligned} \gamma^n \delta_m &\leq \varepsilon \\ n &\geq \log_\gamma \frac{\varepsilon}{\delta_m} \\ &= \log_\gamma \varepsilon - \log_\gamma \delta_m. \end{aligned}$$

For multiple tasks the expected lower bound is

$$\begin{aligned} \bar{n} &\geq \sum_m D(m) \left[\log_\gamma \varepsilon - \log_\gamma \delta_m \right] \\ &= \log_\gamma \varepsilon - \sum_m D(m) \log_\gamma \delta_m. \end{aligned} \tag{3.2}$$

Thus $\bar{n}$ can be minimized by maximizing $\sum_m D(m) \log_\gamma \delta_m = \sum_m D(m) \log_\gamma \|Q_m^* - Q_0\|_\infty$.

Note that $\sum_m D(m) \log_\gamma \|Q_m^* - Q_0\|_\infty = \log_\gamma \prod_m \|Q_m^* - Q_0\|_\infty^{D(m)}$, which equals the log weighted geometric mean of the max-norm between Q_0 and Q_m^*. Therefore, since $\gamma < 1$, the optimal Q_0 *minimizes* the weighted geometric mean of the max-norm.

□

Unfortunately, this expression is in general non-linear and non-convex. However, it seems that good approximations are obtainable by simply minimizing the average distance, according to some norm, between Q_m^* and Q_0. We exploit this intuition in the heuristic approaches to initialization for the *learning* setting that we propose below.

3.3.2 Initialization for the Learning Case

Intuitively, a good initialization of the Q-function is the one closest in expectation, according to some norm, to some desired target value function Q_m of the unknown next task m the agent will face. This can be seen as a definition of a *cross-task value* function Q_d that predicts the expected value of a given state-action pair $\mathbf{x}$ on an unknown new task sampled from the domain, given the target values observed for $\mathbf{x}$ on previous tasks. If the norm is Euclidean, Q_d gives the least-squared-error prediction of the value of $\mathbf{x}$ on the new task. That is, it minimizes the mean squared error (MSE) across tasks:

$$Q_d = \operatorname*{argmin}_{Q_0} \left(\sum_{m \in \mathsf{M}} D(m) \sum_{\mathbf{x} \in \mathbf{X}_m} p(\mathbf{x}|m) \left[Q_m(\mathbf{x}) - Q_0(\mathbf{x}) \right]^2 \right), \tag{3.3}$$

where $p(\mathbf{x}|m)$ is a task-dependent weighting over state-action pairs that determines how much each pair contributes to the error. As is common in least squares, one may want to weight non-uniformly, in our case for example if some state-action pairs occur more often than others. It is not immediately clear how to define $p(\mathbf{x}|m)$. In section 3.3.6, we discuss four possible options for this distribution. For now, we assume $p(\mathbf{x}|m)$ is given. By setting the gradient of (3.3) to zero and solving, we obtain:

$$Q_d(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x}) Q_m(\mathbf{x}). \tag{3.4}$$

Note that $p(m|\mathbf{x})$ is a natural way of selecting the right tasks to average over for a given pair: it is zero for any $(\mathbf{x}) \notin \mathbf{X}_m$, leaving such pairs out of the average.

For linear binary function approximators, solving (3.3) leads to

$$\mathbf{w}_d^i = \sum_{m \in \mathsf{M}} p(m|\mathbf{f}^i = 1)\mathbf{w}_m^i, \tag{3.5}$$

where $\mathbf{w}_d^i$ and $\mathbf{w}_m^i$ are the weight of feature i across tasks and in task m respectively, and $\mathbf{f}^i$ is the value of feature i (either 0 or 1). Thus, instead of averaging per state-action pair as in (3.4), Eq. 3.5 averages per binary feature. In the following sections, we follow the format of (3.4) and use table-based value functions, unless mentioned otherwise.

In supervised learning, the targets Q_m are given. In reinforcement learning, an intuitive choice of target might be the optimal value function of each task. However, there is no guarantee that using a potential function that approximates this target leads to the best online return. In the following three subsections, we propose three different types of Q_m to use as target, each leading to a different potential function.

3.3.3 Least Squared Error Prediction of Q_m^*

By setting the target to be the optimal value function Q_m^* of each task, we obtain:

$$Q_d^*(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x})Q_m^*(\mathbf{x}). \tag{3.6}$$

While approaches equivalent to (3.6) have been used successfully as potential functions or initializations in previous work [126, 138], they are not guaranteed to be optimal. Since they make a prediction based on the optimal policy, they may be too optimistic during learning, which the shaping function is meant to guide. Consequently, they may cause the agent to over-explore the target task to the detriment of online return, a phenomenon that we observe experimentally in section 3.4.

3.3.4 Least Squared Error Prediction of $\tilde{Q}_m$

In some cases, the agent's Q-function on a task m may never reach Q_m^*, even after learning. This may happen, for example, when using function approximation or an on-policy algorithm with a soft policy. When this occurs, it may be better to use a less optimistic potential function based on an average over $\tilde{Q}_m$, the value function to which the learning algorithm converges. The derivation is the same as for the previous section, yielding:

$$\tilde{Q}_d(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x})\tilde{Q}_m(\mathbf{x}). \tag{3.7}$$

3.3.5 Value of the Optimal Cross-Task Policy

The previous two approaches to defining cross-task value for the potential function both rely on value function for tasks that have already been solved. Such approaches may be too optimistic, even if (3.7) is used instead of (3.6), since they are based on the result of learning and implicitly assume a (near-) optimal policy will be followed from the next time step onward, which is not typically the case during learning.

In this section, we propose another definition which, in a sense, more closely resembles the traditional definition of value in that it estimates the value Q_d^μ of a *single* fixed cross-task policy $\mu : \mathbf{X}_d \to [0,1]$, $\mu(\mathbf{x}) = p(a|\mathbf{s})$ that assigns the same probability to a given state-action pair, regardless of the current task. This definition might also be more suitable for use as potential function since, like a potential function, it is fixed across tasks. The value function of the best possible cross-task policy, μ^*, will typically make more conservative estimates than either Q_d^* or $\tilde{Q}_d$, since μ^* is usually not optimal in every (or any) task. For example, consider a domain with a goal location in an otherwise empty room. If the distribution over goal locations is uniform, and the state provides no clue as to the goal position, then μ^* is a uniform distribution over actions in every state, which is suboptimal in any task. We define the cross-task value of a state under a stationary policy μ as

$$Q_d^\mu(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x}) Q_m^\mu(\mathbf{x}). \tag{3.8}$$

Like (3.6) and (3.7), this follows (3.4), except that it averages over the values of a *single* policy instead of multiple task-dependent ones.

The fact that μ is task-independent makes the task essentially a hidden variable and MTRL similar to a POMDP for which μ is a *memoryless policy* that conditions only on the current observation and Q_d^μ is similar to the value of such a policy as defined in [124]. Singh et al. [124] showed that for POMDPs, the best stationary stochastic policy may be better than the best stationary deterministic policy and there need not exist a stationary policy (stochastic or deterministic) that maximizes the value of each state simultaneously. The same facts apply to the multi-task case; since the proofs from Singh et al. do not translate without modification to the multi-task case, we have modified their proofs to fit the multi-task case in Appendix A. Because of these facts, traditional dynamic programming methods may not apply. One way to overcome this problem is to assign a scalar, i.e. state-independent, value to each possible policy:

$$V_d^\mu = \sum_{\mathbf{s} \in \mathbf{S}_d} p(\mathbf{s}) V_d^\mu(\mathbf{s}) \tag{3.9}$$

$$V_d^\mu(\mathbf{s}) = \sum_{a \in \mathbf{A}} \mu(\mathbf{x}) Q_d^\mu(\mathbf{x}) \tag{3.10}$$

where V_d^μ is the domain-wide value of μ and $p(\mathbf{s})$ is a distribution that assigns an appropriate measure of weight to each state $\mathbf{s}$. Two options for $p(\mathbf{s})$ are the start-state distribution or the occupation probability of $\mathbf{s}$, $p(\mathbf{s}|\mu)$. For the POMDP case, Singh et al. [124] show that defining the optimal policy as

$$\mu^* = \operatorname*{argmax}_\mu \sum_{\mathbf{s}} p(\mathbf{s}|\mu) V_d^\mu(\mathbf{s}) \tag{3.11}$$

is equivalent to maximizing the average payoff per time step.

3.3.6 Choosing $p(\mathbf{x}|m)$

All three proposed potential function types take the general form

$$Q_d(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x}) Q_m(\mathbf{x}),$$

where $p(m|\mathbf{x}) = p(\mathbf{x}|m)D(m)/p(\mathbf{x})$. As indicated in section 3.3.2, it is not immediately clear how to define $p(\mathbf{x}|m)$. Four options are:

1. **The stationary distribution induced by the policy corresponding to each Q_m.** For Q_d^* and Q_d^μ, this would be π_m^* and μ^*, respectively. For $\tilde{Q}_d$ it would be the soft policy (e.g. ε-greedy). Since Q_d averages over all Q_m, this seems a good option. However, it may be problematic for Q_d^* and $\tilde{Q}_m$ as it represents only the distribution over $(\mathbf{x})$ *after* learning. There may be state-action pairs that the policy never, or rarely, visits in some tasks, but clearly the values in these tasks should still be included in the average.

2. **The distribution induced by the learning process.** Since we are interested in improving performance during learning, another choice may be a distribution over $(\mathbf{x})$ *during* learning. However, it is unclear how to define this distribution, since it depends, among other things, on when learning is halted and the trajectory through state-action space taken during learning. One possibility is to take one sample from all possible such trajectories and define

$$p(\mathbf{x}|m) = \sum_{t=1}^{T} p(\mathbf{x}|m,\pi_t)p(\pi_t|m),$$

   where $p(\pi_t|m) = 1/T$ for the soft policy the agent was following at time step t when learning task m, $p(\mathbf{x}|m,\pi_t)$ is the stationary distribution over $(\mathbf{x})$ given π_t, and T is a predetermined stopping criterion, such as a fixed number of learning steps or a convergence threshold. Clearly, this option does not apply to Q_d^μ.

3. **The start-state distribution and uniform random policy**. This defines

$$p(\mathbf{x}|m) = p(\mathbf{s}_0 = \mathbf{s}|m)p(a),$$

   where $p(\mathbf{s}_0 = \mathbf{s}|m)$ is the start-state distribution of m and $p(a) = 1/|\mathbf{A}|$ for all a. This definition appropriately captures the distribution over $(\mathbf{x})$ when the agent enters a new task without prior knowledge.

4. **The uniform distribution.** This defines $p(\mathbf{x}|m) = 1/|\mathbf{X}_m|$ for all $(\mathbf{x}) \in \mathbf{X}_m$.

For Q_d^* and $\tilde{Q}_d$, the latter option seems the most sensible one. Under the first and third definitions, a probability of zero might be assigned to a state-action pair in some tasks, or even in all, which is clearly not desirable. Also, since (3.6) and (3.7) are concerned with prediction of the optimal (approximate) value of a pair $(\mathbf{x})$ in a new target task, the underlying assumption is that *after* taking a in $\mathbf{x}$, the optimal (soft) policy for that task will be followed. However, the first two definitions in the list above make assumptions about the policy that has been followed *so far*, which is irrelevant in this context. Empirical comparison of the options revealed no significant difference for Q_d^* and $\tilde{Q}_d$. For Q_d^μ, the stationary distribution induced by μ was found to work best in most cases.

3.4 Potential Functions: Empirical Evaluations

This section empirically compares agents applying the three potential functions proposed above to a baseline agent that does not use shaping, and introduces the three domains used throughout this chapter and the next. While these domains are simple, they make it feasible to illustrate critical factors in the performance of shaping functions in MTRL.

For comparison purposes, we assume the agent has perfect knowledge of each domain and thus computes each potential function using all tasks in the domain. Our goal is to demonstrate the *theoretical* advantages of each type of potential function. Assuming perfect domain knowledge enables comparisons of the potential functions' maximum performance, untainted by sampling error. From Chapter 4 onward, we consider the more realistic setting in which only a sample sequence of tasks is available, and in which generalization to unseen next tasks, and thus learning a good representation for the potential function, is important.

3.4.1 Episodic Cliff Domain

To illustrate a scenario in which Q_d^*, the average over optimal value functions, is not the optimal potential function, we define a *cliff domain* based on the episodic cliff-walking grid world from Sutton and Barto [135]. The agent starts in one corner of the grid and needs to reach a goal location in the corner that is in the same row or column as the start state, while avoiding stepping into the cliff that lies in between the start and goal.

The domain contains all permutations with the goal and start state in opposite corners of the same row or column with a cliff between them (8 tasks in total). Each task is a 4x4 grid world with deterministic actions N, E, S, W, states (x, y), and a -1 step penalty. Falling down the cliff results in -1000 reward and teleportation to the start state. The distribution over tasks is uniform. We compute each potential function according to the definitions given in section 3.3. Finding the cross-task policy μ^* has been shown to be NP-hard for POMDPs [148]. We use a genetic algorithm (GA) to approximate it[2].

To illustrate how performance of a given Φ depends on the learning algorithm, we use two standard RL algorithms, Sarsa and Q-Learning. Since for Q-Learning, $Q_d^* = \tilde{Q}_d$, we use Sarsa's solution for $\tilde{Q}_d$ for both algorithms. Both algorithms use an ε-greedy policy with $\varepsilon = 0.1$, $\gamma = 1$, and the learning rate $\alpha = 1$ for Q-Learning and $\alpha = 0.4$ for Sarsa, maximizing the performance of both algorithms for the given ε.

We also run an additional set of experiments in which the agent is given a cliff sensor that indicates the direction of the cliff (N, E, S, W) if the agent is standing right next to it. Note that the addition of this sensor makes no difference for learning a single task, since the information it provides is already deducible from the agent's position and the number of states per task is not affected. However, the number of states in the domain does increase: one result of adding the sensor is that tasks no longer have identical state spaces.[3]

[2] We employ a standard real-valued GA with population size 100, no crossover and mutation with $p = 0.5$; mutation adds a random value $\delta \in [-0.05, 0.05]$. Policies are constructed by a softmax distribution over the chromosome values.

[3] Note that the addition of this sensor is not the same as the manual separation of state features for the value and potential function as done in [68, 122] – see related work (section 3.5). In the experiments reported in this

	Q-Learning	Sarsa
Q_d^μ	-19.77 ± 2.43	-512 ± 130
No shaping	-5.86 ± 0.12	-3.86 ± 0.10
Q_d^*	-5.13 ± 0.17	-3.96 ± 0.11
$\tilde{Q}_d$	-4.74 ± 0.19	-3.93 ± 0.11

(a) Without sensor

	Q-Learning	Sarsa
Q_d^μ	–	–
No shaping	-5.85 ± 0.13	-3.96 ± 0.11
Q_d^*	-5.44 ± 0.12	-3.67 ± 0.12
$\tilde{Q}_d$	-4.75 ± 0.17	-3.37 ± 0.13

(b) With sensor

Table 3.1: Mean total reward and 95% confidence interval for various shaping configurations and learning algorithms on the cliff domain. All numbers $\times 10^4$.

For each potential function, we report the mean total reward incurred by sampling a task from the domain, running the agent for 500 episodes, and repeating this 100 times. Table 3.1a shows performance without the cliff sensor. On this domain, Q_d^μ performs very poorly; one reason may be that the GA did not find μ^*, but a more likely one is that, due to the structure of the domain, even μ^* would incur low return for each state, yielding a pessimistic potential function.

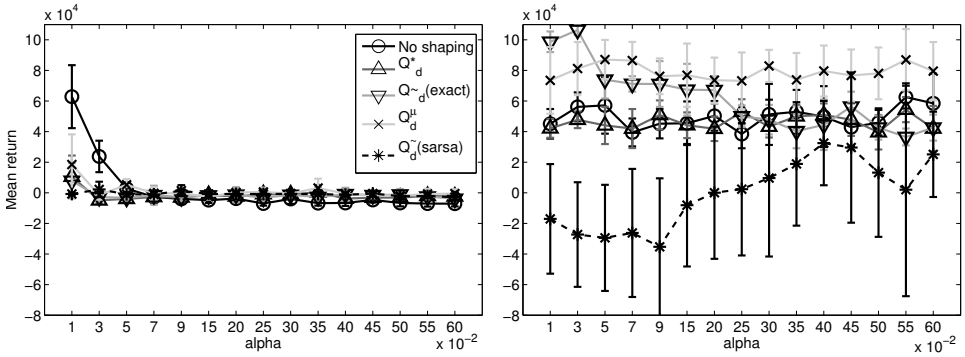
As expected, Sarsa outperforms Q-Learning on this domain due to its on-policy nature: because Q-Learning learns the optimal policy directly, it tends to take the path right next to the cliff and is thus more likely to fall in. For Q-Learning, Q_d^* and $\tilde{Q}_d$ do better than the baseline, with the latter doing significantly better.

The situation changes when the cliff sensor is added (we did not retest the potential function that did worse than the baseline), as shown in table 3.1b. Though the sensor does not speed learning within a task, it provides useful information across tasks: whenever the cliff sensor activates, the agent should not step in that direction. This information is reflected in the average of the value functions and thus in the potential function, More precisely, the state-action space $\mathbf{X}_d^+$ is enlarged and fewer state-action pairs are shared between tasks. Under these circumstances, both Q_d^* and $\tilde{Q}_d$ significantly outperform baseline Sarsa, with the latter, again, doing best. The picture for Q-Learning remains largely the same.

3.4.2 Continuing Cliff Domain

The continuing cliff domain is the same as the episodic version with the cliff sensor, except that there are no terminal states. Instead there is a reward of 0 on every step and 10 for passing through the goal state and teleporting to the start state. We hypothesized an additional benefit for shaping here, since the reward function is sparser (see e.g. [73] for a formal relation between the benefit of shaping and sparsity of reward). To our surprise, however, this was not always the case. Ironically, one main reason seems to be the sparse reward function. In addition, the presence of an area of large negative reward next to the goal state makes the task even more difficult to learn. For increasing exploration rate ε, the optimal ε-greedy agent takes ever larger detours around the cliff, partly because of the sparse reward function; from around $\varepsilon = 0.05$, it huddles in the corner of the grid

section, both functions use the exact same set of features.



(a) Sarsa

(b) Q-Learning

Figure 3.1: Mean return and 95% confidence intervals of Sarsa and Q-learning on the continuing cliff domain, under various shaping regimes. Return is cumulative reward over 10^5 steps and averaged over all tasks in the domain (i.e. 8 runs).

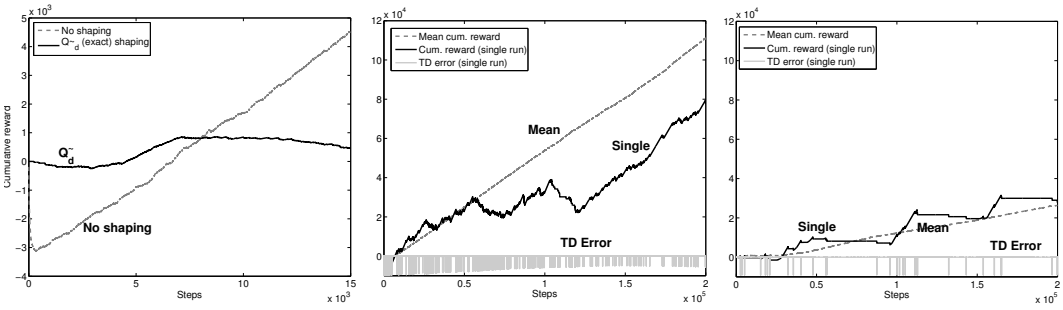
indefinitely, without ever attempting to reach the goal. For this reason, we used $\varepsilon = 0.01$ for all experiments.

Figure 3.1 shows the mean cumulative reward of Sarsa and Q-learning under various learning rates and shaping regimes. Here, we used two different methods for computing $\tilde{Q}_d$: one uses the exact values of the optimal ε-greedy policy for each task for $\varepsilon = 0.01$, as computed by a soft version of policy iteration[4], and the other uses solutions as computed by Sarsa run on each task in the domain with $\varepsilon = 0.01$, $\alpha = 0.07$, for 10^7 steps. These two value functions are different since, as it turns out, Sarsa converges to the wrong solution.

The figure shows a markedly different picture from the episodic cliff world results. Even at its optimal setting, Sarsa does not significantly outperform Q-learning on this domain. Perhaps even more surprising, especially since Sarsa converges to the wrong solution, is that unshaped Sarsa at its optimal learning rate outperforms shaped Sarsa at its optimal setting; the shaped version dominates only from around $\alpha = 0.15$, and generally not significantly.

Fig. 3.2 reveals what is happening. Fig. 3.2a makes clear that, even though shaping has a disadvantage when measured over a large number of timesteps, it does provide an initial performance boost which lasts up to around 10^4 steps. However, as the two rightmost graphs show, the long-term learning dynamics of shaped Sarsa ultimately result in inferior performance: it is plagued by long periods of stasis, in which it keeps far from the cliff and thus the goal, and no reward is incurred. The likely reason is the same as for the eventual stasis of unshaped Sarsa (not shown), which results from Sarsa's inability, under the low exploration rate, to escape from the strong local optimum in the corner of the grid (recall that under higher exploration rates, the corner of the grid becomes the global optimum). Since shaped Sarsa is closer to convergence than unshaped Sarsa, it discourages the agent from approaching any location that might contain a cliff, resulting in an initial performance boost but also earlier onsets of stasis.

[4] In the policy improvement step, the policy is made only ε-greedy w.r.t. the value function.



(a) Shaped vs unshaped Sarsa (b) No shaping (c) $\tilde{Q}_d$ (exact) shaping

Figure 3.2: Cumulative reward of unshaped and shaped Sarsa on the continuing cliff domain. Mean curve (light gray, dashed) represents average over 100 runs; also shown are example cumulative reward (black,solid) and TD error from a single run.

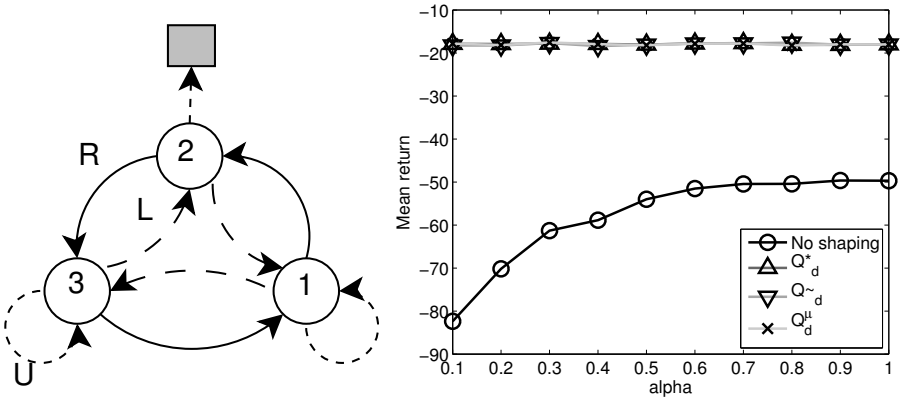
3.4.3 Triangle Domain

Each task in the episodic Triangle domain consists of three non-terminal states, in which three actions can be taken (Fig. 3.3a), and one terminal state. In addition to feature x_1, which corresponds to the state numbers shown, the agent perceives two additional features x_2 and x_3. Feature x_2 is the inverse of the square of the shortest distance to the goal, i.e. in the figure, the states would be $(1, 0.25), (2, 1), (3, 0.25)$. Feature x_3 is a binary feature that indicates task color: red (1) or green (0); if red, the agent receives a -10 penalty for self-transitions, in addition to the -1 step reward that is default in every task. x_3 is constant within a task, but may change from one task to the next. The goal may be at any state and the effect of actions L (dashed) and R (solid) may be reversed. Action U (dotted) always results in either a self-transition or a goal-transition (when the goal is next to the current state). There are thus 12 tasks in total.

We compare performance of the different shaping regimes on this domain with a Q-learning agent with discount factor $\gamma = 1$. Not surprisingly, there is no significant difference between the potentials in this domain: while Q_d^* estimates values higher than $\tilde{Q}_d$, which estimates higher than Q_d^μ, differences in estimates are minimal and the ordering of actions is the same.

3.4.4 Stock-Trading Domain

The binary stock-trading domain is an attractive domain for comparison since it is an established benchmark [22, 71, 130], is stochastic, and has an easily varied number of states and tasks. An example task is displayed in Fig. 3.4a. The domain consists of a number of sectors S, such as telecom ($s1$ in the example) and pharmaceuticals ($s2$). Each contains E items of equity (stock). Each stock is either rising (1) or falling (0). An agent can buy or sell all stocks of one whole sector at a time; sector ownership is indicated by a 1 (owned) or 0 (not owned) in the state vector. Therefore, if the agent owns pharma but not telecom, the part of the state vector pertaining to stock in the example would



(a) Example triangle task

(b) Performance

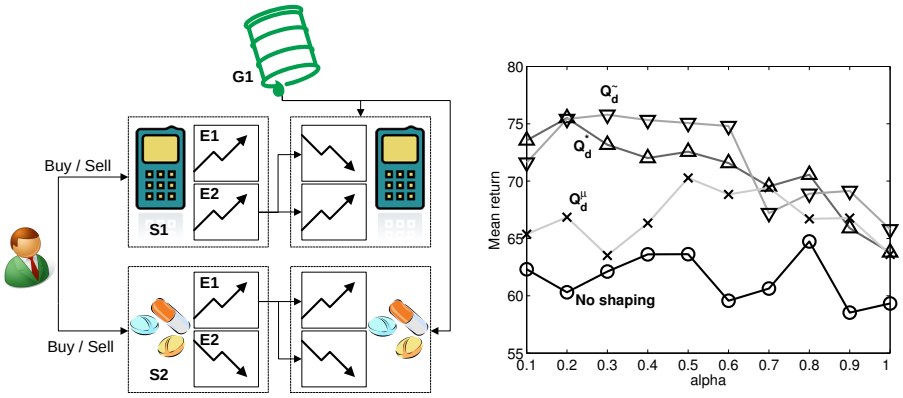
Figure 3.3: Example task and mean return of q-learning on the triangle domain, under various shaping regimes. Return is cumulative reward over 10 episodes and averaged over all tasks in the domain (i.e. 12 runs). All differences between the shaping methods are significant, except between $\tilde{q}_d$ and q_d^{μ}. Points shown are for $\varepsilon = 0.01$, the best-performing setting.

be $(0,1,1,1,0,1,0)$: the first two elements indicate sector ownership, the next two what telecom stocks are doing, the two thereafter what pharma stocks are doing, and the final entry what the global factor is doing (explained in the next paragraph). At each timestep, for each sector that it owns, the agent receives a reward of +1 for each stock that is rising and -1 for each stock that is falling. Thus in the example, the agent would earn a reward of 1, since there is 1 stock rising in pharma.

The probability of stocks rising in a given sector s, P_s, depends on two factors: the number of stocks rising in s in the previous timestep, and the influence of G global factors (in the example, oil is the only global factor). How stocks and globals influence P_s is task-dependent. In the example, the only telecom stock of influence is e_2; in pharma, it is e_1. In a given task, stocks within a sector may be influenced by any combination of stocks in that sector. Stocks that are rising increase P_s; stocks that are falling decrease it.

Globals behave just like stocks in that they can rise (1) or fall (0). However, globals always affect all sectors simultaneously. The effect of a global varies per task; for a given rising global, it increases P_s in half the tasks, and decreases it in the other half, making its net cross-task effect zero. The exact formula for determining the probability that stocks in a given sector s will rise in a given task m is:

$$P_s^m = 0.1 + 0.8 \frac{R_s^m + 3R_g^m}{I_s^m + 3G},$$



(a) Stock domain (b) Performance

Figure 3.4: Schematic overview of, and mean Q-learning return on, the stock-trading domain under various shaping regimes. a) See main text. b) Return is cumulative reward over 300 learning steps and averaged over 20 runs for all tasks in the domain (i.e. 240 runs). Differences between the methods are generally significant, except between $\tilde{Q}_d$ and Q^*_d.

where
I^m_s = number of stocks of influence in s in m
G = number of globals
R^m_s = number of stocks of influence in s in m that are rising
R^m_g = number of globals that are rising and increase P^m_s in m when rising, plus number of globals that are falling and increase P^m_s in m when falling.

Thus, in the example, assuming that oil prices are on the rise and increase the probability that stocks rise when falling, for $s =$ telecom,

$$P^m_s = 0.1 + 0.8\frac{1+3\times 0}{1+3\times 1} = 0.3.$$

A domain is defined by the tuple $\langle S, E, G\rangle$; the number of tasks in the domain is $(2^E-1)2^G$. A state is represented by $S+SE+G$ binary features, so $|\mathbf{S}_m| = 2^{(S+SE+G)}$, and $|\mathbf{A}| = 2S$.

We ran a Q-Learning agent on a domain with $S = 1$, $E = 3$, and $G = 2$. As shown in Fig. 3.4, Q^*_d and $\tilde{Q}_d$ perform best in this domain; once again $\tilde{Q}_d$ seems to be slightly superior, although the difference is not significant. Q^μ_d lags behind, although its performance seems more resistant to changes in α; for higher learning rates, the difference with the other two shaping functions disappears.

3.5 Related Work

The two main research areas that relate to this chapter are shaping and multi-task (reinforcement) learning. This section reviews work done in these areas and discusses their relationship to our own work.

3.5.1 Potential-Based Shaping

The theoretical result of Ng et al. [94], which showed that potential-based shaping functions preserve the optimal policy of the ground MDP for model-free RL, has been extended in various ways. Grześ and Kudenko [47] demonstrate empirically that scaling the potential can affect learning performance, and relate performance of a distance-to-goal-based potential to the discount factor γ and task size, showing that as task size increases, so should γ. Asmuth et al. [3] show that R-max, a popular model-based RL method, is still PAC-MDP [63] when combined with an admissable potential function $\Phi(\mathbf{x}) \geq Q^*(\mathbf{x})$. In multi-agent RL, potential-based shaping provably preserves the Nash equilibrium in stochastic games [19, 79], and improves approximations to the Pareto front [83]. Preservation of optimal policy and Nash equilibrium have also been shown to hold for potential functions that change while the agent is learning[20]. Harutyunyan et al. [52] provide an algorithm that allows a human designer to express advice in an intuitive fashion while retaining the theoretical guarantees of potential-based shaping. Brys et al. [13] extract a shaping reward from human demonstrations, and Suay et al. [131] improve on this by using inverse reinforcement learning (IRL) [95] to create a general encoding of a demonstrator's policy (by learning a reward function), instead of using the observed demonstration samples directly. The theory underlying potential-based shaping has implications for IRL in general: IRL approaches that try to recover a reward function from an expert's policy may recover a potential-based shaping function instead. This directly follows from the result of Ng et al. [94]. Fu et al. [39] argue that IRL algorithms that learn a shaping function may not be robust to changes in dynamics, and propose an algorithm that can avoid learning a shaped reward in domains where the ground-truth reward only depends on state, and not on transitions.

In practice, potential-based shaping has also been shown to improve strategies for the iterated Prisoner's dilemma [4], robot soccer [21], Dynamic Economic Emissions Dispatch problems [82], and mapping visual observations and text instructions to actions [91].

There have been a number of successes in learning potentials automatically. In single-task RL, one approach is to construct an initial shaping function based on intuition [72] or an initial task model [46], and refine it through interaction with the task. Elfwing et al. [28, 29] evolve a shaping function that, when transferred to a real robot, results in better performance than when transferring Q-values. Other work has learned a shaping function on abstractions that are either provided [48] or also learned [86]. As we will see in the next chapter, which contains results on learning potential function representations, this work is related to ours in that it explores different representations for the potential and value function. However, our work benefits from the MTRL setting in that it learns the abstractions offline, in between tasks, and therefore does not incur a cost while interacting with the next task.

Konidaris and Barto [68] were the first to learn a shaping function automatically in a multi-task environment by estimating the value function based on optimal solutions of previously experienced tasks. They base the value function on the *problem space*, the Markov representation necessary for optimally solving tasks, and the potential function on the *agent space*, the representation that retains the same semantics across tasks. However, they pre-specify both spaces. Similar pre-designed separate representations were

employed in [122], in which an optimal reward function is searched on a distribution of tasks. This thesis substantially extends this work by comparing different potential functions and, in the next chapter, proposing a method for automatically discovering both task and domain-relevant representations.

3.5.2 Multi-Task Reinforcement Learning

The field of multi-task reinforcement learning has rapidly developed in recent years and is too large to survey comprehensively. Instead, we focus on approaches most similar to ours; see Taylor and Stone [140] for an extensive survey of the field.

In [141], the value function of the single source task is transformed via inter-task mappings and then copied to the value function of the target task. In [138], the average optimal Q-function of previously experienced tasks is used to initialize the model of a new task. Although the authors state that the average optimal Q-function is always the best initialization, this chapter has shown otherwise. Mehta et al. [89] assume fixed task dynamics and reward features, but different reward feature weights in each task. Given the reward weights of a target task, they initialize the task with the value function of the best stored source task policy given the new reward weights.

Under a broader interpretation, initialization can also be performed by transferring source task experience samples to a batch-learning method in the target task [76]. Similar to other initialization methods, bias towards the source task knowledge decreases as experience with the target task accumulates. Other forms of initialization are to use a population of evolved source task policies as initial population for an evolutionary algorithm in the target task [142], or to use source task information to set the prior for Bayesian RL [75, 159].

Advice-giving methods, which suggest an action to the agent given a state, are closely related to potential-based shaping. Torrey et al. [145] identify actions in source tasks with higher Q-values than others, and use this information to construct rules on action preferences that are added as constraints to a linear program for batch-learning Q-function weights in the target task. In [146], this work is extended by using inductive logic programming to extract the rules. Tayor and Stone [142] learn rules that summarize a learned source task policy and incorporate these as an extra action in the target task, to be learned by a policy-search method. All these approaches are flexible in that they can deal with different state features and actions between tasks by a set of provided inter-task mappings. However, these mappings have to be provided by humans, except in [142], where they can be learned if provided with a description of task-independent state clusters that describe different objects in the domain. As we will see in the next chapter, these clusters are similar to our notion of domain-relevant abstractions, which we discover automatically with the algorithm proposed in the next chapter.

3.6 Conclusion and Discussion

Given full domain knowledge, we have shown that the initial value function that results in the lowest number of value iterations on an unknown task sampled from the domain minimizes the weighted geometric mean max-norm between the initial value function

and optimal value functions. For the learning case, we argue that a sensible potential function minimizes the expected Euclidean norm between it and the solutions to tasks in the domain; i.e., it is the weighted average over task solutions. Previous work has not made this argument explicit, nor has it specified what kind of task solutions are best. One contribution of this chapter is to define three potential function types, based on the use of three different kinds of task solutions. The first, Q_d^*, is based on optimal task solutions. The second, $\tilde{Q}_d$, is based on approximate solutions or solutions that are optimal with respect to a soft policy, which is what the agent is usually using while learning. The third, Q_d^μ, is based on the value functions of the optimal cross-task policy μ^*.

Experiments showed that which type is best depends highly on the domain, learning algorithm, and learning parameters. This disproves the assumption made elsewhere [138] that the average optimal Q-function is always the best initialization. Since the values of $\tilde{Q}_d$ and Q_d^μ are inherently lower than those of Q_d^*, this may seem to contradict the assumption that optimistic initialization is a good heuristic [33, 135]. However, it is more likely that each potential type recommends a different policy, depending on the domain. In addition, relative differences between Q-values, which matter for shaping, are likely to play a role. Nonetheless, it is possible that scaling the potentials could have a positive effect on performance, as has been demonstrated for the single-task case [47]. Scaling might also improve performance of one potential type relative to another. One possibility for future work would be to try to derive a scaling factor from source tasks, e.g., based on maximum observed Q-values.

In some of the domains discussed in this chapter, the best potential function is based on solutions that are not produced by the learning algorithm[5]. For example, in two of the three domains, a Q-learning agent was helped more by $\tilde{Q}_d$ than by Q_d^*, and in the remaining domain there was no significant difference. Using $\tilde{Q}_d$ with a Q-learning agent would, in practice, entail having to solve each task twice, and using Q_d^μ is similarly problematic. One solution could be to learn a model of each encountered task, and use that to compute $\tilde{Q}_d$ or Q_d^μ offline, in between tasks; another idea is to learn two value functions simultaneously. While this would result in a slight increase in time and space requirements, neither idea increases the number of experience samples required.

It is well known that transfer learning in supervised learning and RL can hurt performance if tasks are not sufficiently related [14, 140, 143]. This chapter identified another potential source of negative transfer. On the continuing cliff domain, shaped Sarsa was outperformed by non-shaped Sarsa for low learning rates, showing that not only task relatedness, but also the learning algorithm and parameters may affect the benefit of transfer.

[5] We obtained them by running another agent with the appropriate learning algorithm for the potential function type.

4

Relevant Representations for Multi-Task RL

We now turn to the setting in which only a sample sequence of tasks is available to the agent. A central aim is to *generalize* well from the sample to new data. As in supervised learning, representation is key for generalization. This chapter proposes a definition of relevance that can be used for learning representations in MTRL.

In MTRL, new data may consist of both new state-action pairs and new tasks. Our central objective, therefore, is to discover which information is relevant across tasks, which we call *domain-relevant* information. While our definition also captures which information is relevant *within* tasks (task relevant), this subject has been extensively addressed in existing research. Therefore, we focus primarily on discovering domain-relevant representations.

The Triangle domain from section 3.4 illustrates the distinction between task and domain relevance. To describe an optimal policy for each task, only position, which has the Markov property, is needed to represent state. However, the value of a given position needs to be re-learned in every task. Since position does not represent information that can be retained between tasks, this feature is task relevant. Task color is not useful within a given task, since its value is constant. However, across tasks it represents information about self-transitions, which receive additional punishment in red tasks. Therefore, this feature is domain relevant, but not task relevant. Finally, distance to goal is useful both within and across tasks, and is therefore both task and domain relevant, as is the cliff sensor in the cliff domain.

Projecting the full feature set onto a subset of the task-relevant features may create an abstract state space that is smaller than the original state space, and can therefore help learn the task more quickly, under certain conditions on the projection [77]. In the multi-task setting, it is possible to define this abstract space *before* entering a new task, by identifying valid abstractions (task-relevant representations) of previously experienced tasks and transferring these to the new task (e.g. [35, 71, 74, 149]).

Domain-relevant features also allow for a higher level of abstraction, but in addition allow the agent to *deduce rules from the abstract representation and reason with them, right from the start of a new task*, thereby obviating the need to re-learn this task-invariant knowledge. Of course, it should be possible for the agent to override the heuristic rules given new information garnered from interaction with a specific task. Shaping functions

are a good candidate for these kind of rules, endowing the agent with prior knowledge that gradually degrades as the agent accumulates experience of a task.

In the following sections, we propose a definition of relevance that is based on a state representation's ability to predict expected return within tasks (for task-relevant features) or across tasks (for domain-relevant features). We argue that task and domain relevance are special cases of a single underlying concept, k-relevance, the expected relevance on a sequence of k tasks sampled from the domain. We show formally that k-relevance converges to a fixed point in the limit, and under certain assumptions does so monotonically. We use this property to introduce FS-TEK, a novel feature selection algorithm. The key insight behind FS-TEK is that change in relevance observed on task sequences of increasing length can be extrapolated to more accurately predict domain relevance.

4.1 Relevance

The notion of relevance that this section introduces applies generally, not just to sequences of tasks. Therefore, we start out by discussing abstraction and relevance in general, for any Q-function, and then extend the concept to sequences of tasks.

Several notions of predictive power on a given Q-function Q are possible. For example, the Kullback-Leibler divergence between the marginal distribution over Q and the distribution conditioned on the representation of which we wish to measure relevance [126], or the related measures of conditional mutual information [51] or correlation [84]. The disadvantage of these measures is that they do not take the magnitude of the impact of a representation into account; for example, two representations may cause equal divergence, but the difference in expected return associated with one set may be much larger than the other. To address this problem, we propose a measure of relevance[1] that is proportional to the squared error in predicting Q.

Li et al. [77] provide an overview and classification of several state abstraction schemes in reinforcement learning. We employ their definition of an abstraction function:

Definition 1 (Abstraction function)**.** *An abstraction function $\phi : \mathbf{X} \mapsto \mathbf{Y}$ induces a partition on the vector space $\mathbf{X}$; $\phi(\mathbf{x}) \in \mathbf{Y}$ is the abstract vector $\mathbf{y}$ corresponding to $\mathbf{x}$, and the inverse image $\phi^{-1}(\mathbf{y})$ is the set of ground vectors that corresponds to $\mathbf{y}$ under abstraction function ϕ.*

Here and in the following, by *ground* vectors or state-action pairs we mean those from the original vector space. Using this definition, we can define a Q-function on abstract state-action pairs:

Definition 2 (Abstract Q-function)**.** *Given abstraction $\phi : \mathbf{X} \to \mathbf{Y}$ and any Q-function $Q : \mathbf{X} \to \mathbb{R}$, the abstract Q-function $\bar{Q}(\Phi) : \mathbf{Y} \to \mathbb{R}$ is defined as the weighted average of the Q-values of ground pairs $\mathbf{x}$ corresponding to abstract pair $\mathbf{y}$:*

$$\bar{Q}_\phi(\mathbf{y}) = \sum_{\mathbf{x} \in \phi^{-1}(\mathbf{y})} p(\mathbf{x}|\mathbf{y}) Q(\mathbf{x}). \tag{4.1}$$

[1] Relevance is not a measure in the strict mathematical sense; because of dependence between feature sets, $\rho(\mathsf{F} \cup \mathsf{G}) \neq \rho(\mathsf{F}) + \rho(\mathsf{G})$ for some disjoint feature sets F and G and relevance ρ.

The Q-value of a given abstract state-action pair $\mathbf{y}$ generally differs from those of at least some ground state-action pairs corresponding to $\mathbf{y}$. Call this difference the *abstraction error* $\varepsilon_\phi(\mathbf{y}, Q)$. One example is to choose the weighted mean squared error (MSE) of ground pairs with respect to $\mathbf{y}$ as abstraction error:

Definition 3 (MSE Abstraction error $\varepsilon_\phi(\mathbf{y}, Q)$)**.** *Given abstraction ϕ and any Q-function Q, the MSE of a given abstract state-action pair $\mathbf{y}$ with respect to its corresponding ground pairs is given by*

$$\varepsilon_\phi(\mathbf{y}, Q) = \sum_{\mathbf{x} \in \phi^{-1}(\mathbf{y})} p(\mathbf{x}|\mathbf{y}) \Big[Q(\mathbf{x}) - \bar{Q}_\phi(\mathbf{y}) \Big]^2. \tag{4.2}$$

Whatever the choice of $\varepsilon_\phi(\mathbf{y}, Q)$, it follows naturally that the relevance of an abstraction with respect to a function Q is the total error incurred by applying the abstraction to Q.

Definition 4 (Relevance)**.** *Given abstraction ϕ and any Q-function Q, the relevance $\rho(\phi, Q)$ of ϕ with respect to Q is defined as the abstraction error on Q that results from applying ϕ:*

$$\rho_\phi(Q) = \sum_{\mathbf{y} \in \mathbf{Y}} p(\mathbf{y}) \varepsilon_\phi(\mathbf{y}, Q). \tag{4.3}$$

When using an abstract Q-function as defined in (4.1) and MSE as abstraction error (4.2), this equals the sum of weighted variances of the Q-values of ground state-action pairs corresponding to a given $\mathbf{y}$:

$$\rho_\phi(Q) = \sum_{\mathbf{y} \in \mathbf{Y}} p(\mathbf{y}) \operatorname{Var}(Q(\phi^{-1}(\mathbf{y}))). \tag{4.4}$$

Note that the aim is therefore to find an abstraction with *low* relevance: this abstracts away non-relevant parts of the original representation and keeps the relevant parts. This is in line with the terminology employed by Li et al. [77], who designate the abstraction that preserves Q^* as a Q^*-irrelevance abstraction. Similarly, an abstraction with low relevance on Q should be thought of as a Q-irrelevance abstraction.

4.1.1 k-Relevance

Now we are ready to expand the above to sequences of tasks. Since our main goal is to find good representations for potential functions, our targets for abstraction are Q_d^*, $\tilde{Q}_d$, or Q_d^μ, the three types of potential function proposed in section 3.3. Which of these is used does not matter for the theory, as any Q-function can serve as a basis for relevance. Therefore, in what follows, we remain agnostic to the target for relevance.

Since the agent has only experienced a sample sequence of tasks from the domain, it cannot compute this target Q_d exactly. Instead, it can approximate Q_d by computing a cross-task value function based on the sequence. Let $c = (c_1, c_2, ..., c_k)$ be a sequence of $|c|$ tasks sampled from M, and $\mathbf{X}_c = \bigcup_{c_i \in c} \mathbf{X}_{c_i}$.

Let Q_c be a cross-task Q-function computed on a sequence of tasks c, $Q_c : \mathbf{X}_c \mapsto \mathbb{R}$. If $|c| = 1$, $Q_c = Q_m$. All of the definitions for Q_d in section 3.3 follow the general form

$$Q_d(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x}) Q_m(\mathbf{x}).$$

Thus, we have for Q_c:

$$Q_c(\mathbf{x}) = \sum_{i=1}^{|c|} p(c_i|\mathbf{x},c) Q_{c_i}(\mathbf{x}). \tag{4.5}$$

Naturally, relevance often depends on both the length of the task sequence over which it is computed and on the tasks that the sequence contains. For example, in the Triangle domain as a whole, the position feature is irrelevant since, given a certain position, there is no way to know what direction to go. In other words, in Q_d, the average value function computed over the whole domain, Q-values will not vary with the position feature, and therefore the position feature will not result in any error on Q_d if it is discarded.

However, given a sample sequence of two Triangle tasks, it is likely that position is still relevant, e.g., if the goal is in the same direction in both tasks. In practice, the agent, while interacting with the domain, will have at its disposal a growing sequence of tasks based on which it must construct a domain-relevant representation. Doing so is challenging because representations may appear relevant given the observed sequence but actually not be domain irrelevant. Intuitively, the longer the sequence, the better sequence relevance approximates domain relevance. Therefore, instead of just computing relevance on the currently observed sequence, we should try to predict how it *changes* with increasing sequence length. Decreasing relevance could mean that the feature, while relevant on the current sequence, is not relevant in the long run.

However, it is not enough to simply observe that the relevance of a given representation is decreasing because it may plateau above zero. For example, Fig. 4.1 shows how average relevance of single features in the Triangle domain changes with increasing sequence length. While the relevance of the action a and distance feature x_2 decreases, these features remain domain relevant. Relevance of the position feature x_1, on the other hand, disappears completely. This implies that in this domain, it is desirable to abstract away x_1.

To predict how relevance changes as more tasks are observed, we use the definitions in the previous section to define the *k-relevance* of a representation as the expected relevance ρ_k over all possible sequences of length k. Let C_k be the set of all possible sequences c of length k, i.e., $|\mathsf{C}_k| = |\mathsf{M}|^k$. The probability of sampling a given sequence c from C_k is $p(c) = \prod_{i=1}^{|c|} D(c_i)$. Then we have the following definition for k-relevance:

Definition 5 (k-relevance)**.** *The k-relevance of an abstraction $\phi : \mathbf{X}_d \to \mathbf{Y}_d$ in a domain $d = \langle D, \mathsf{M} \rangle$ is the expected relevance of ϕ on a given Q-function computed from a task sequence of length k sampled from M according to D*

$$\rho_k(\phi) = \mathrm{E}[\rho(\phi, Q_c)|c \in \mathsf{C}_k] = \sum_{c \in \mathsf{C}_k} p(c) \rho_\phi(Q_c). \tag{4.6}$$

Given this definition, an abstraction ϕ is strictly domain relevant, $\mathrm{DR}(\phi)$, if and only if its k-relevance ρ_k remains positive as k goes to infinity:

$$\mathrm{DR}(\phi) \Leftrightarrow \lim_{k \to \infty} \rho_k(\phi) > 0, \tag{4.7}$$

and strictly domain irrelevant otherwise:

$$\neg\mathrm{DR}(\phi) \Leftrightarrow \lim_{k \to \infty} \rho_k(\phi) = 0. \tag{4.8}$$

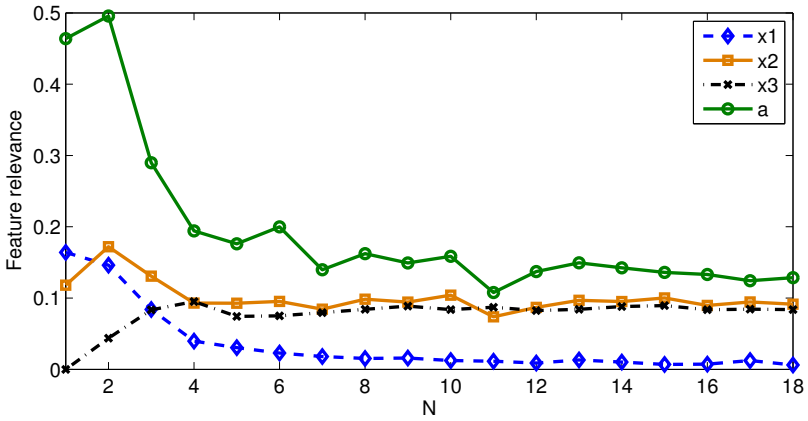


Figure 4.1: Average relevance of single features in the Triangle domain based on task sequences of length N. Relevance is averaged over a sample of all possible sequences of length N.

Our goal is thus to find a domain-irrelevant abstraction, as applying such an abstraction yields a domain-relevant representation. In section 4.1.2, we show that ρ_k converges to the true domain relevance as k goes to infinity.

As k tends to infinity, the average over all sample sequences of length k will approach the true distribution over tasks in the domain ever more closely. However, the value of k at which k-relevance is a reasonably accurate approximation of the true domain relevance depends largely on the number of tasks in the domain: for example, the relevance of the position feature in Fig. 4.1 is practically 0 from around $k = 12$.

Similarly, an abstraction may be called task relevant, $\mathrm{TR}(\phi)$, if and only if its 1-relevance is greater than 0:

$$\mathrm{TR}(\phi) \Leftrightarrow \rho_1(\phi) > 0. \tag{4.9}$$

Thus, a sensible abstraction to use in the domain for the value function or policy would be one that is (near-)irrelevant according to this definition and thus the expected error on the value function of a task introduced by the abstraction is negligible. We believe this is a more natural definition than, e.g., preserving any feature that has been found relevant in any task in the domain [149].

Thus, k-relevance is a unifying notion of relevance that naturally captures both task and domain relevance and can therefore be used for finding abstractions for both the value function of a new task and the potential function to use on that task. In the next section, we show that k-relevance converges to a fixed point as k tends to infinity, a result which makes it possible to extrapolate domain relevance from observed tasks, as we show in section 4.2.

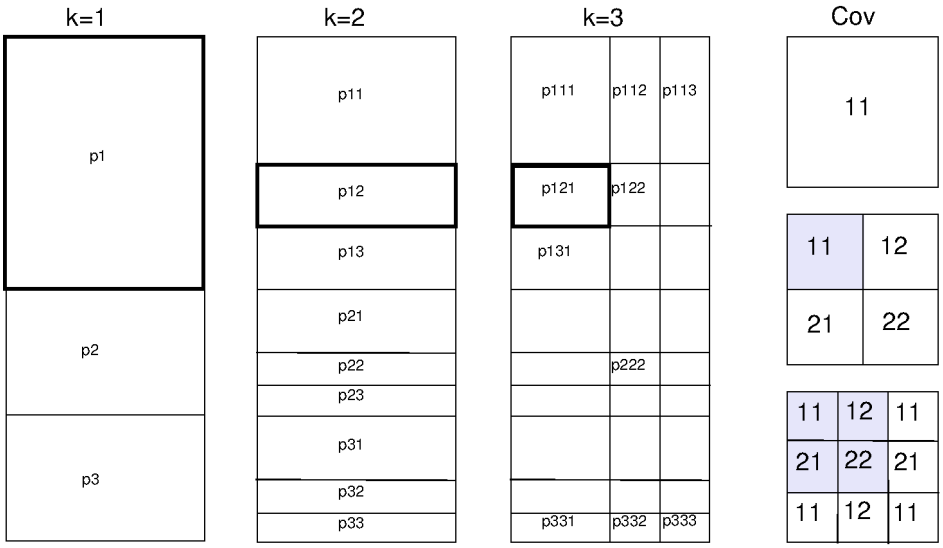


Figure 4.2: Left three columns ($k = 1..3$): probability of each sequence as k increases, shown as portions of a rectangle with area 1, on a 3-task domain with $p1 = 0.5$, $p2 = p3 = 0.25$. Probability of sequence $(1,2)$ is indicated as $p12$. For each $k+1$, the area of each sequence for k is split into new sequences with area proportional to D. Right column: covariance matrices for the sequences marked in bold on the left. The weight of each matrix element is equal, namely $1/k^2$. Relevance on a given sequence is the sum of all matrix elements. In turn, k-relevance is the sum over all sequences.

4.1.2 Properties of k-Relevance

Fig. 4.2 illustrates how k-relevance changes with k. Relevance of a given ϕ is the same on each single-task sequence for a given task m, i.e. sequences for which each element $c_i = m$. For $k = 1$, k-relevance comprises only single-task sequences, but the share of these sequences decreases exponentially with k; in the example, it is $(p1)^k + (p2)^k + (p3)^k$. In general, the sequences represent the true distribution over tasks ever more closely.

For $k = 2$, every task is combined with every task in the domain. Hence for $k > 2$, no new task combinations arise; however, we need a way to quantify relevance on, e.g., $c = (1,2,3)$ given the relevance on $(1,2)$, $(1,3)$, and $(2,3)$. We show in Appendix B that it follows, from the expression of relevance in terms of variance (Eq. 4.4) that relevance on any sequence equals the sum of covariances between the Q-functions of all task pairs involved in the sequence. The right column of Fig. 4.2 visualizes this; the highlighted areas correspond to the covariance matrix (and hence relevance) for $k-1$; therefore, $k+1$-relevance is a function of k-relevance. The following two theorems follow directly from this, together with the fact that sequences approximate the true distribution over tasks ever more closely.

Theorem 2. *Let ϕ be an abstraction with abstract Q-function as in definition 2. Let $\rho_k = \rho_k(\phi)$ for any k, based on the MSE abstraction error as in definition 3. Let $d(x,y) =$*

$|x-y|$ be a metric on $\mathbb{R}$, and let $f(\rho_k) = \rho_{k+1}$ map k-relevance to $k+1$-relevance. Then f is a strict contraction; that is, for $k > 1$ there is a constant $\kappa \in (0,1)$ such that

$$d(f(\rho_k), f(\rho_{k-1})) \leq \kappa d(\rho_k, \rho_{k-1}).$$

Proof. See Appendix B. □

Corollary 1. *The sequence $(\rho_k)_{k=1}^{\infty}$, $k = 1, 2, \ldots$, converges to a fixed point, namely the domain relevance of ϕ.*

Proof. Since $(\mathbb{R}, d)$ is a complete metric space and f is a contraction, f has a unique fixed point by Banach's fixed-point theorem. □

Theorem 3. *If all tasks share the same distribution over state-action pairs $p(\mathbf{x}|m)$, then ρ_k, as defined in Theorem 2, is monotone.*

Proof. See Appendix B. □

Note that having the exact same distribution over state-action pairs also implies having the exact same state-action space. By these two theorems, ρ_k can be approximated by an exponential or power law function, as also suggested by Fig. 4.1. Even when tasks do not share the same distribution over state-action pairs, an exponential function can still approximate where the sequence $(\rho_k)_{k=1}^{\infty}$ will converge to, given sufficient k.

4.2 Feature Selection With k-Relevance

We now introduce FS-TEK (Feature Selection Through Extrapolation of k-relevance), a novel algorithm for finding relevant abstractions in multi-task reinforcement learning that exploits the notion of k-relevance introduced in the previous sections. FS-TEK focuses on finding domain-relevant representations, since determining task-relevant representations using k-relevance is relatively straightforward. As mentioned previously, the key idea behind FS-TEK is to fit an exponential function to a candidate representation's relevance based on an observed sequence of tasks and then extrapolate it to estimate the representation's domain relevance.

Since it is a feature selection algorithm, FS-TEK constructs abstractions of the form $\phi(\mathbf{x}) = \mathbf{x}[\mathsf{Y}] = \mathbf{y}$, where $\mathsf{Y} \subseteq \mathsf{X}$ and $\mathbf{x}[\mathsf{Y}]$ denotes the values of the features Y in vector $\mathbf{x}$. That is, the abstract state $\mathbf{y}$ to which a state $\mathbf{x}$ is mapped consists of the values of the features in some relevant set Y. Viewed another way, Y is the result of removing some irrelevant set $\mathsf{F} = \mathsf{X} - \mathsf{Y}$ from X. In the following, when we refer to the relevance of a set F, we mean the relevance of the abstraction that removes F from X.

FS-TEK's main loop is based on an iterative *backward elimination* procedure (e.g. [50, 66]). Because of interdependence between features, it is not always sufficient to select features based on their relevance in isolation. Features may be irrelevant on their own, but relevant together with another feature; similarly, a feature that seems irrelevant may become relevant once another feature is removed. Backward elimination starts with the full feature set and iteratively removes features according to a measure of relevance. In contrast, *forward selection* methods start with the empty set and iteratively add

features. While forward selection may yield a smaller final feature set, it may miss interdependencies between features (e.g. [50]). Therefore, FS-TEK's main procedure uses backward elimination. Nonetheless, it tries to combine the advantages of both methods by using forward selection to decide which feature to remove when more than one feature is marked for elimination.

Naturally, FS-TEK's measure of relevance is extrapolated k-relevance. Computing this involves two main steps:

1. For an observed sequence c of tasks of length $|c|$, compute k-relevance of each feature set for $k \in \{1, 2, \ldots, |c|\}$

2. For each feature set, fit an exponential function $f(k)$ to the relevance computed in step 1 and extrapolate to the point where $\frac{df}{dk} = 0$.

Theoretically, in step 1, k-relevance is the expected relevance taken over all possible sequences of length k given the tasks in the domain. In practice, the domain is unknown and computing relevance over all $|c|^k$ possible task sequences is infeasible. Therefore, the domain is assumed to consist of the tasks seen so far, and for each k, a sample of sequences is used to compute the k-relevance.

In step 2, a feature set's extrapolated relevance will often not be 0, even if it is truly domain irrelevant. Therefore, the algorithm conducts a statistical test in which the null hypothesis is that the feature set is not domain relevant. The set classified is as domain relevant only if the estimated relevance deviates significantly from 0.

Since applying steps 1 and 2 to each feature set in the powerset of X^+ is impossible, backward elimination adds features one at the time to the set R of features to be removed. In each iteration, each remaining feature is tested in conjunction with R. This process repeats until no more feature sets are judged domain irrelevant.

Algorithm 7 specifies the main body of FS-TEK. It takes as input the sequence s of solutions to observed tasks, a parameter *max_seqs* that specifies the maximum number of sequences the algorithm should sample for computing k-relevance, and a parameter $\alpha(k)$ that specifies the confidence level for the statistical test on relevance, possibly depending on sample sequence length k.

In each iteration, FS-TEK starts by computing the k-relevance of each feature united with the current set of features to be removed, with k ranging from 1 to the current number of observed tasks (line 6-9; the details of BKR, Backward-k-Relevance, are provided in algorithm 8 below). For each feature, this results in a dataset with k as input and relevance as output (each column of the matrix D). The algorithm subsequently does a nonlinear least-squares fit of the exponential function $f(k;\theta)$ to each feature's relevance data (line 13).

If the estimated function is increasing, the feature is kept (lines 14-16). Otherwise, the function value and confidence interval are computed for the point where the function asymptotes (lines 17-19; in line 19, $\mathrm{CI}(a, \alpha(|s|))$ is the confidence interval at point a for the length of the current observed sequence of tasks). If the confidence interval's lower bound is less than or equal to 0, the feature is classified as domain irrelevant and added to the set of features to be removed (line 20).

This procedure often marks more than one feature for removal. In order to detect interactions between features, it is not desirable to remove more than one feature at the

Algorithm 7 FS-TEK

Require: a sequence s of task solutions Q_{s_i}, $i \in \{1,2,\ldots,|s|\}$; *max_seqs*, maximum number of sequences to sample; $\alpha(k)$, the confidence level

Ensure: A set R of features to remove

$\mathsf{R} \leftarrow \emptyset$
$f(k;\theta) = \theta_1 + \theta_2 \exp(-\theta_3 k)$
repeat
  $\mathsf{F} \leftarrow \emptyset$
  $D \leftarrow$ empty $|s| \times |\mathsf{X}^+ - \mathsf{R}|$ matrix, i.e. a row per k-relevance and a column per feature
  for $k = 1$ to $|s|$ **do**
    $D(k) = \mathrm{BKR}(s, \mathsf{R}, max\_seqs, k)$
  end for

  // Each feature for which extrapolated function of relevance does not differ
  // significantly from 0 is marked for removal
  for all $X_i \in \mathsf{X}^+ - \mathsf{R}$ **do**
    $\hat{\theta} = \mathrm{argmin}_{\hat{\theta}} \sum_{k=1}^{|s|} \left[f(k;\hat{\theta}) - D(k,i) \right]^2$
    if $f(2,\hat{\theta}) - f(1,\hat{\theta}) > 0$ **then**
      Continue to next X_i
    end if
    $a \approx \min\{a : \frac{df(k;\hat{\theta})}{dk}(a) = 0\}$ //least-squares fit
    if $f(a,\hat{\theta}) - \mathrm{CI}(a, \alpha(|s|)) \leq 0$ **then**
      $\mathsf{F} \leftarrow \mathsf{F} \cup X_i$
    end if
  end for

  // Of all marked features, the one with weakest forward relevance is removed
  for all $X_i \in \mathsf{F}$ **do**
    $\mathsf{G} \leftarrow \mathsf{R} \cup X_i$
    $r(i) = \mathrm{FR}(s, \mathsf{G})$
  end for
  $\mathsf{R} \leftarrow \mathsf{R} \cup \mathrm{argmin}_i\, r(i)$
until $\mathsf{F} = \emptyset$

time. FS-TEK uses forward selection to decide which feature to remove (lines 24-26; in practice this step is skipped if F contains fewer than 2 features). In this check, the relevance of the set of features to remove is tested in isolation, without taking into account the features that remain. Contrary to BKR, which computes relevance for a range of k, FR (Forward Relevance) computes relevance only on the currently observed sequence of tasks. Using forward selection can enable FS-TEK to select a smaller subset of relevant features [2] and provides a "second opinion" to supplement the noisy estimate of BKR.

The forward and backward relevance methods operate essentially according to similar principles. Algorithm 8 describes BKR, which computes, for each feature not yet marked for removal, the k-relevance for the given k. To compute relevance, it uses either $\binom{|s|}{k}$ task combinations, or a sample of *max_seqs* of those combinations, whichever is smaller (lines 2-3). While k-relevance is defined in terms of sequences, the number of combinations is smaller and, for small sample sizes, more accurately reflects the distribution over tasks (since tasks are not repeated).

Given a set of task sequences C_k, the average Q-function for each sequence is computed according to equation 4.5 (lines 6-8). Each feature's k-relevance is then the average relevance taken over those Q-functions, as defined in (4.6) (lines 11-15). Relevance is computed for the abstraction ϕ_i that removes from the full feature set a feature set consisting of the features removed in previous iterations united with the given feature (lines 12-13).

FR (algorithm 9) is similar but only computes relevance based on the current task sequence. In addition, it adds features to the empty set instead of removing them from the full set. It first computes an abstract function based on the set G of features to be removed (lines 2-4). Forward relevance is then the relevance of the null abstraction (the empty set of features) with respect to the Q-function based on G: i.e., the difference between the error made by using G and that made by using no features.

Not taking into account the complexity of the nonlinear least-squares optimization procedure, FS-TEK's complexity is mainly determined by BKR. Let the number of features $|\mathsf{X}^+| = P$, the size of the state-action space $|\mathbf{X}_s| = N$, and the maximum number of sequences $max\_seqs = M$. BKR's worst-case time complexity is $O(MkN + (P - |\mathsf{R}|)(N + MN))$. However, in practice FS-TEK does not re-sample the sequences at each iteration and computes the average Q-functions based on the sequences only once at the start of the algorithm, so the MkN term can be removed. This results in a complexity of $O(N(P - |\mathsf{R}|)(1 + M))$ for BKR when used by FS-TEK.

In the worst case, all features in X^+ need to be removed. FS-TEK's first iteration incurs a cost of $O(|s|NP(1+M))$, since R is empty. On the second iteration, $|\mathsf{R}| = 1$, and thus the cost is $O(|s|N(P-1)(1+M))$. Since a total of P iterations is needed, the cost in terms of P progresses as $P + P - 1 + P - 2 + \cdots + 1 = P(P+1)/2$. Therefore, the total cost is $O(|s|N(P^2+P)(1+M))$, i.e., quadratic in the number of features. Also, while the cost is linear in the size of the state-action space N, in the worst case N scales exponentially with the number of features P. In practice, however, usually not all features need to be removed, and features frequently covary such that N does not scale exponentially with P.

[2] Imagine a set of features $\{A, B, C, D\}$. The target function can either be represented by $\{A, B\}$ or $\{A, C, D\}$. Since $\{C, D\}$ covers the same information as $\{B\}$, C and D are likely to be weaker in isolation and thus to be removed by forward selection; in addition, using backward elimination, B, C, and D would have relevance 0 since the information each feature provides is already covered.

Algorithm 8 BKR: Backward-k-Relevance

Require: a sequence s of task solutions Q_{s_i}, $i \in \{1,2,\ldots,|s|\}$; R the set of features currently marked for removal (possibly empty); *max_seqs*, maximum number of sequences to sample; k

Ensure: $\rho_{\mathbf{k}}$, a row vector with the k-relevance of each $X_i \in \mathsf{X}^+ - \mathsf{R}$

$N \leftarrow \min\left(max\_seqs, \binom{|s|}{k}\right)$

$\mathsf{C}_k \leftarrow N$ sample combinations from the $\binom{|s|}{k}$ possible combinations of length k from s

// Compute the average Q-function Q_c for each sequence

for all $c \in \mathsf{C}_k$ **do**

  $Q_c(\mathbf{X}) = \sum_{i=1}^{|c|} p(c_i|\mathbf{X},c) Q_{c_i}(\mathbf{X})$ // Equation 4.5

end for

// Compute k-relevance of each feature

for all $X_i \in \mathsf{X}^+ - \mathsf{R}$ **do**

  $\mathsf{Y} = \mathsf{X}^+ - (\mathsf{R} \cup X_i)$

  $\phi_i : \mathbf{X} \to \mathbf{X}[\mathsf{Y}]$

  $\rho_k(\phi_i) = \frac{1}{N} \sum_{c \in \mathsf{C}_k} \rho(\phi_i, Q_c)$ // Equation 4.6

end for

Algorithm 9 FR: Forward Relevance

Require: a sequence s of task solutions Q_{s_i}, $i \in \{1,2,\ldots,|s|\}$; the set of features to be tested G

$Q_s(\mathbf{X}) = \sum_{i=1}^{|s|} p(s_i|\mathbf{X},s) Q_{s_i}(\mathbf{X})$ // Equation 4.5

$\phi : \mathbf{X}_s \to \mathbf{Y}_s$, where $\mathbf{Y}_s = \mathbf{X}_s[\mathsf{G}]$

$\bar{Q}_s(\mathbf{y}) = \sum_{\mathbf{x} \in \mathbf{X}_s^{\mathbf{y}}} p(\mathbf{x}|\mathbf{y}) Q_s(\mathbf{x})$ // Abstract function based on G, according to (4.1)

$\phi_\emptyset : \mathbf{X}_s \to \emptyset$

return $\rho(\phi_\emptyset, \bar{Q}_s)$

4.3 Representation Selection: Empirical Evaluations

In this section, we test FS-TEK on the three domains of section 3.4. We compare to another backward elimination algorithm that also uses our definition of relevance, but does not make use of the multi-task information by extrapolating. This algorithm, IBKR (Iterated BKR) is identical to FS-TEK, except that it only computes relevance for $k = |s|$, the length of the sequence of experienced tasks. In addition to IBKR, we compare FS-TEK to shaping functions constructed without FS; with randomly selected features; and with fixed features, in which only the true domain-relevant features are used.

In each experiment, a Q-learning agent interacts sequentially with each domain. After each task, the agent constructs a new potential function based on the solutions of the tasks solved so far. Since the agent uses Q-learning, the potential function is based on Q_c^* as defined in (4.5), i.e., the sample sequence analogue of Q_d^*; likewise, relevance is computed based on Q_c^*. Although section 3.4 showed that, in some domains, using a potential based on $\tilde{Q}_d$ works better, doing so is impractical here as it would require solving each task twice (once using Q-learning, and then again using, e.g., Sarsa to compute the potential function).

For each domain, we use a fixed learning rate α and ε-greedy exploration, where α and ε are set to the best values found in section 3.4. For each new potential function (i.e., for each number of tasks seen), we test the agent on 10 tasks sampled from the domain; the whole procedure is repeated for 500 runs. Performance is measured only after three tasks, since FS-TEK requires at least this many tasks (since the exponential function to be estimated has three parameters).

While our implementation of FS-TEK largely corresponds to that outlined in algorithms 7 and 8, there are some practical tweaks. Especially for a small number of experienced tasks, the Jacobian of the estimated exponential may be ill-conditioned, often preventing reliable extrapolation and computation of the confidence interval. When this happens, we mark the feature as "unsure" until reliable estimates can be made in a subsequent iteration. In addition, for a given call to FS-TEK, any feature that is neither marked for removal nor "unsure" in any iteration, is kept indefinitely and not re-checked on subsequent iterations. This greatly improves speed and yields equal or better performance on the domains under consideration. We use the Levenberg-Marquardt algorithm [85] for the nonlinear least-squares fit.

Triangle Domain

For this domain, we set the confidence level for IBKR to $\alpha = 0.15$, while for FS-TEK, we used $\alpha(k) = \min(\max(k-4,0) \times 0.06 + 10^{-4}, 0.3)$, i.e., it is 10^{-4} up to $k = 5$, from which point it linearly increases with k until a maximum of 0.3. An increasing confidence level for FS-TEK of this kind, and a constant level for IBKR, were found to work best by a coarse parameter search on this domain. Recall that higher confidence means that features are more easily marked as relevant. Increasing α makes sense since as more tasks are observed, estimates become more certain and the confidence interval can be tightened.

The left panel of Fig. 4.3 shows the mean return of shaping functions constructed using the various FS methods. FS-TEK achieves a significant performance improvement

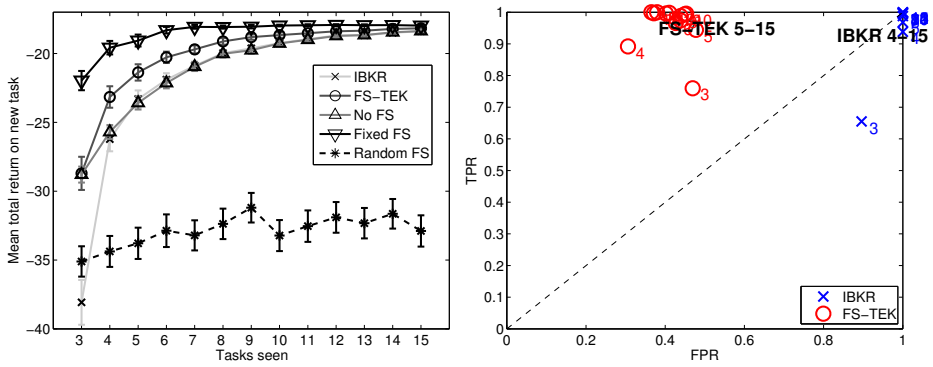


Figure 4.3: Performance of shaping functions constructed using various FS methods (left) and ROC space (right) of the BKR and FS-TEK methods.

over both regular shaping and shaping with IBKR and is the first to match the performance of fixed FS, in which the correct features were hard-coded. The other methods also eventually reach that performance since, once the experienced tasks and their observed frequency approach the true set of tasks and distribution, the need for generalization disappears.

The right panel shows the ROC (Receiver Operating Characteristic) space of the IBKR and FS-TEK method, plotting the False Positive Rate (FPR) against the True Positive Rate (TPR). In the context of this chapter, the TPR indicates the ratio of features correctly classified as domain relevant out of all domain relevant features, while the FPR indicates the ratio of features incorrectly classified as domain relevant out of all domain irrelevant features. Ideally, TPR=1 and FPR=0. From here on, we denote ROC by FPR/TPR, e.g., 0/1 in the ideal case. In the plot, the numbers next to the markers indicate the number of tasks seen.

The triangle domain contains one domain-irrelevant feature, namely the state number. The other three features are DR. In this domain, IBKR mostly achieves an ROC of 1/1, which amounts to never removing any features and is equivalent to the vanilla shaping function – indeed, their performance is nearly identical. FS-TEK does a better job, achieving an ROC of around 0.45/1 after five tasks seen, meaning it nearly always identifies the right DR features (as did IBKR), and in addition removes the irrelevant feature 55% of the time.

Cliff Domain

While the cliff domain contains no domain-irrelevant features, the first two features (encoding position) are only very weakly domain relevant. For the ROC space plot, therefore, we marked the position features as domain irrelevant; this shows that FS-TEK removes the position features about 40% of the time (Fig. 4.4), which explains its slight performance gain.

Because of the peculiar progress of performance with the number of observed tasks, we plotted performance from 1 observed task onward. The initial increase and subse-

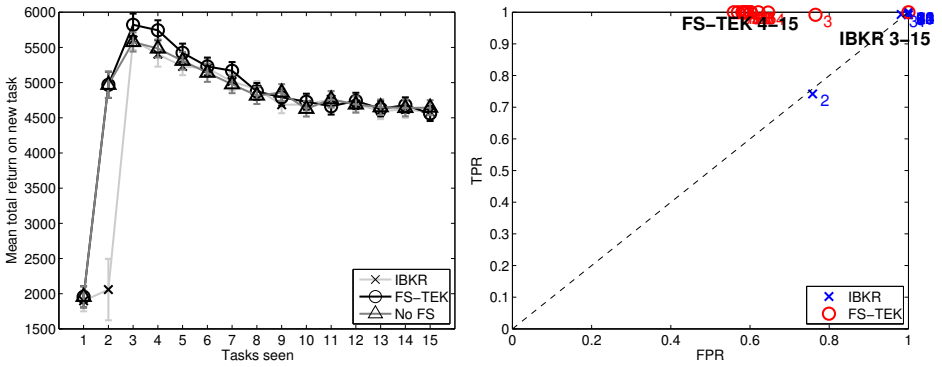


Figure 4.4: Cliff domain when Q_d^* is used as potential. No fixed FS is shown since there are no features to remove; random FS is not shown because of its inferior performance.

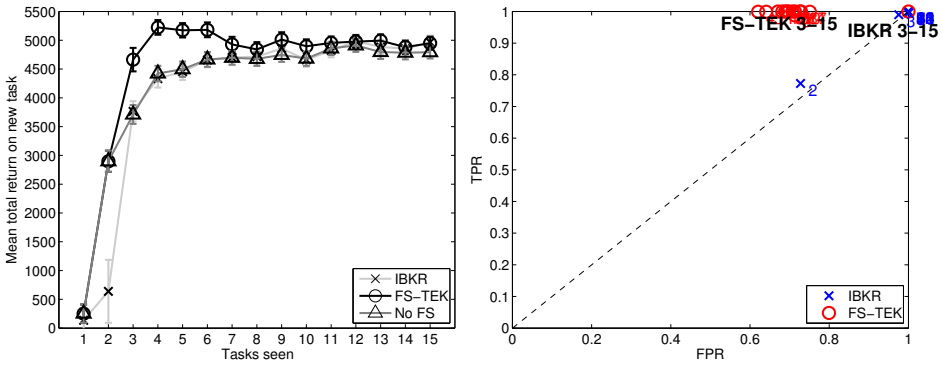


Figure 4.5: Cliff domain when $\tilde{Q}_d$ is used as potential. No fixed FS is shown since there are no features to remove; random FS is not shown because of its inferior performance.

quent decrease in performance with number of observed tasks for all methods is interesting. Since using $\tilde{Q}_d$ as potential was found to work better on this domain (section 3.4), we were curious if the same trend would happen for $\tilde{Q}_d$. Fig. 4.5 shows the results.

Clearly, the same trend does not happen for $\tilde{Q}_d$, and the benefit of FS-TEK is greater with this potential type. Moreover, while $\tilde{Q}_d$ does better than Q_d^* in the long run, as shown in section 3.4, Q_d^* outperforms $\tilde{Q}_d$ for a low number of observed tasks. The explanation must therefore be sought in the difference between $\tilde{Q}_d$ and Q_d^*. With respect to the cliff sensor, $\tilde{Q}_d$ encourages the agent to move away from the cliff, while Q_d^* does not. Of course, when a cliff direction has not been encountered yet in previous tasks, both potentials have uniform preference over actions. With respect to position (i.e. when the cliff sensor shows no reading), Q_d^* pushes the agent towards the center of the grid; $\tilde{Q}_d$, on the other hand, pushes the agent towards the edges of the world. In short, $\tilde{Q}_d$ encourages exploration, but to shy away from a cliff once one is encountered; Q_d^* encourages sitting in the center, but to stay near a cliff once one is encountered.

The likelihood that the agent has encountered a given cliff increases with k. Therefore, for Q_d^* the likelihood that the agent will stick close to a cliff and fall into it increases with k. At some point, this likelihood together with the tendency to push the agent towards the center gains critical mass and performance declines. For $\tilde{Q}_d$, performance increases, since as the agent encounters more cliffs it is less likely to fall into them; in addition, this potential function increasingly encourages exploring the edges of the world and thus discovering the cliff in the current task sooner.

Stock Domain

For the stock domain, we used the settings $S = 1$, $E = 2$ as detailed in section 3.4.4, but tested for G ranging from 1 to 5. For $G = 1$, the size of the state-action space $|\mathbf{X}| = 32$ and $|\mathsf{M}| = 6$; these numbers double every value of G until for $G = 5$, $|\mathbf{X}| = 512$ and $|\mathsf{M}| = 96$. Recall from section 3.4.4 that S represents sector ownership features, E represents the number of stocks per sector and G represents task-dependent global variables that positively or negatively (depending on the task) affect the probability that stocks rise. For the current settings, stock ownership is completely irrelevant, while the stock features and action are domain relevant. The domain is challenging because the G features are strongly task relevant, much more so than the other features, but domain irrelevant (across all tasks their effect cancels out). In addition, the stock and action features are only weakly domain relevant. This means that the G features appear strongly domain relevant when an insufficient number of tasks have been experienced; moreover, they add noise to the shaping function when selected.

For $G < 4$, we used a confidence level $\alpha = 10^{-4}$ for FS-TEK and $\alpha = 10^{-3}$ for IBKR. For higher G, we used an $\alpha(k)$ that increases with k for FS-TEK; this was found to result in better performance. On the other hand, IBKR performed equally well for varying and constant α, so we kept it constant at $\alpha = 10^{-2}$ for $G \geq 4$.

The results are shown in Fig. 4.6. Generally, FS-TEK significantly outperforms all other methods, but its performance deteriorates until, for $G = 4$ and higher, it performs about as well as IBKR and vanilla shaping. It may seem that this is caused by the change in ROC from $G = 4$ onward, which in turn is caused by the change in the confidence level setting. However, the opposite is true: earlier experiments showed that a constant $\alpha = 10^{-4}$ resulted in a similar ROC as for lower G, but also in a mean return that was significantly worse than any other method. Instead, the constant performance of FS-TEK in terms of ROC versus the deteriorating performance shows that the task dynamics, rather than FS-TEK's ability to identify the correct features, are the cause of the performance decline. These dynamics are such that for low G, it is more important to not mistakenly select domain irrelevant features than to select the relevant ones (this also explains why FS-TEK does better than fixed FS for $G = 1$); for higher G, having a low FPR decreases in importance while a high TPR increases in importance. This makes sense since the more G variables there are, the more their effect is diluted, and the more important (relatively) the domain relevant features become.

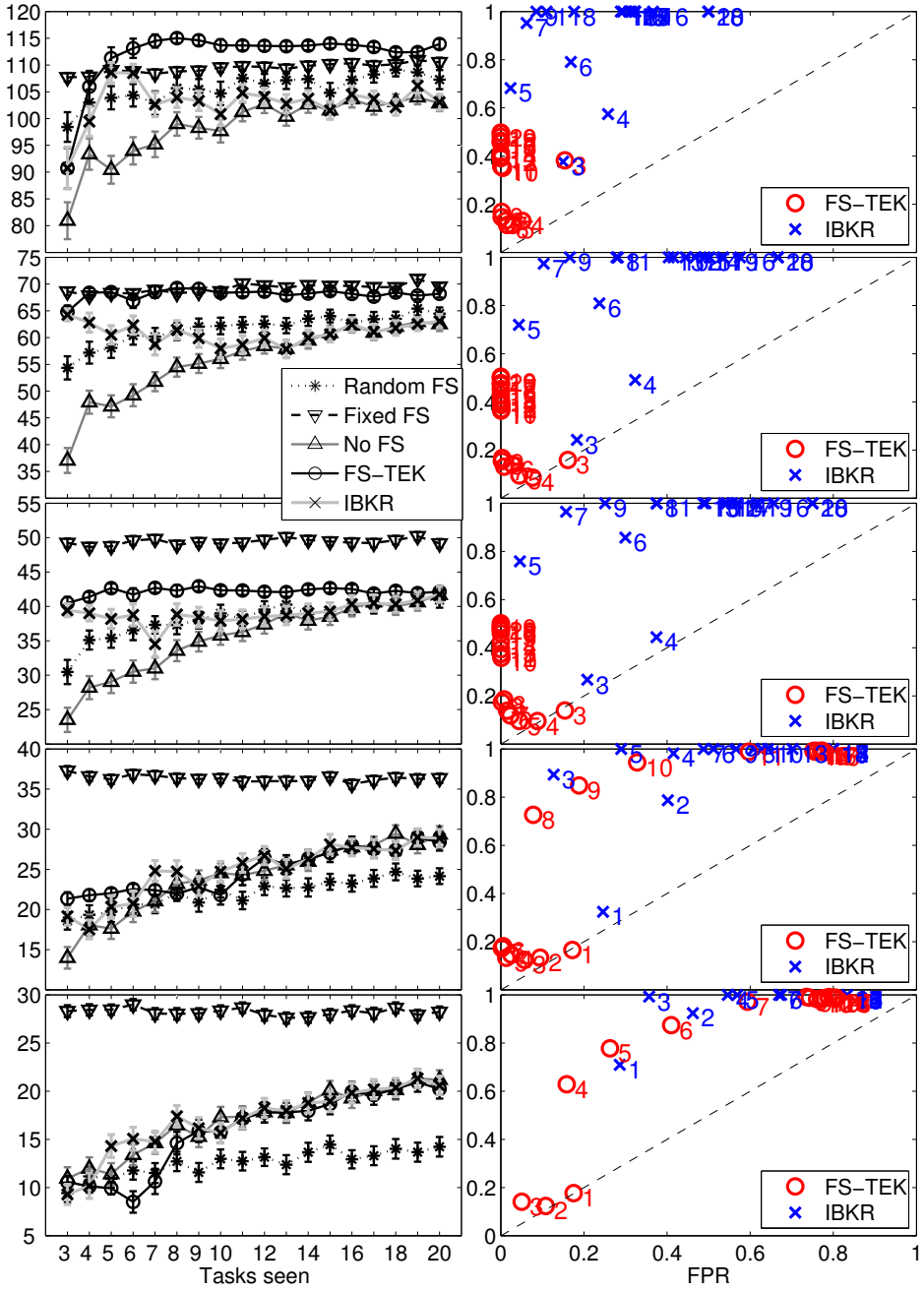
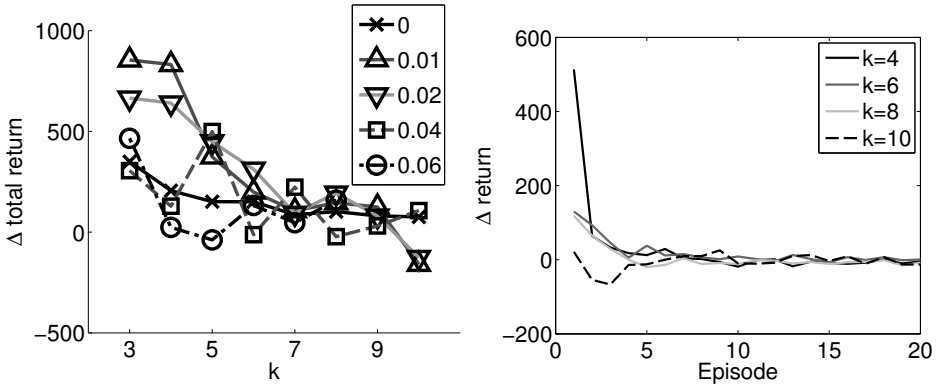


Figure 4.6: Stock domain for $G = 1$ (top) to 5 (bottom). Left column plots tasks seen versus mean total return. Right column plots FPR versus TPR.



(a) Improvement in total return per k

(b) Improvement per episode, for $\sigma_{sensor} = 0.02$

Figure 4.7: Heat domain for $\sigma_{slip} = 0.1$. On the left the improvement in total return that FS-TEK achieves over regular shaping for the first five episodes, per k. Each line represents a different value for σ_{sensor}. On the right the improvement in return that FS-TEK achieves over regular shaping per episode, for $\sigma_{sensor} = 0.02$.

Heat Domain

To assess FS-TEK's ability to perform in a more challenging setting requiring function approximation, we also consider the heat domain, in which a circular agent with a radius of 1 learns to find a heat source in a 10x10 walled-off area. Note that here, we are using Eq. (3.5) to compute the potential function, and are using an abstraction function $\phi(\cdot)$ that computes abstractions directly in feature space.

State is continuous and consists of (x,y) position, the robot's heading in radians, and the intensity of the heat emitted by the source. The agent moves by going forward or backward, or turning left or right by a small amount.

Reward per step is $-10/\sqrt{(}10 \times 10 + 10 \times 10) = -0.71$. An episode terminates when the heat source is within the agent's radius. The agent employs a jointly tile-coded Q-function approximator with 4 overlapping tilings, resulting in a total of 1664 features. We run a Sarsa(λ) agent with $\lambda = 0.9$ and $\varepsilon = 0.05$ for 500 episodes, and compare the performance of FS-TEK to regular shaping under various levels of noise. Transition noise adds $\xi \sim \mathcal{N}(0, \sigma_{slip})$ to the agent's action and sensor noise adds $\xi \sim \mathcal{N}(0, \sigma_{sensor})$ to all state features. Since noise increases the chance of overfitting, our hypothesis is that FS-TEK should result in a greater benefit for higher levels of noise.

Fig. 4.7 shows results for $\sigma_{slip} = 0.1$ and varying sensor noise. The left panel shows that FS-TEK achieves a significant jump in performance over the initial episodes for lower k, and as expected this benefit increases to some extent with the noise level. As observed previously, FS performs on par with regular shaping for higher k. The reason that FS has less benefit for noise levels above 0.02 seems to be that removing the right features becomes more difficult. This is confirmed by the ROC plots (Fig. 4.8), which show that FS-TEK's false positive rate increases as noise increases.

The right panel of Fig. 4.7 shows the benefit of FS per episode, for $\sigma_{sensor} = 0.02$ and

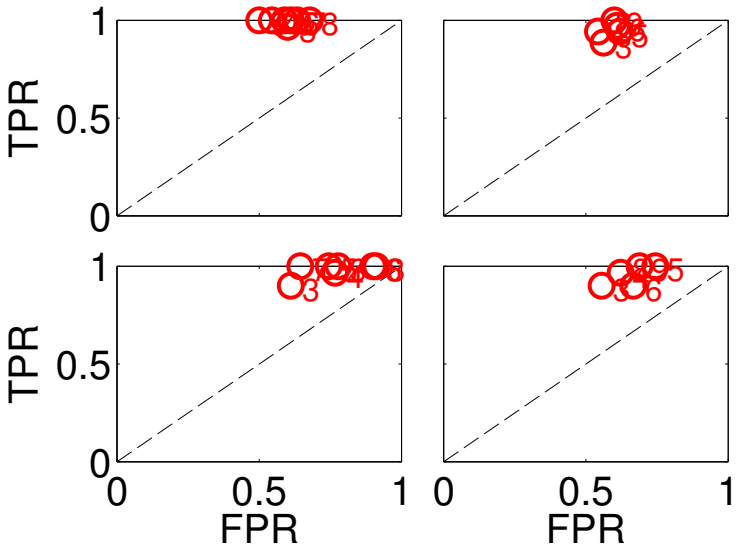


Figure 4.8: FS-TEK ROC space for various noise levels on the Heat domain. Clockwise starting in the top left, noise levels σ_{sensor} are 0.01, 0.02, 0.04, 0.06.

various k. As k increases, there is a dip for early episodes; this dip is also the reason that FS and no FS perform on par for higher k. The likely explanation is that there is some small amount of information contained in the features that FS-TEK discards; therefore the regular potential function is based on more information. The fact that this happens for higher k stems from the fact that there is less overfitting because of the larger amount of data (previous experience) available. Since only the potential function and not the value function has a reduced feature set, this slightly wrong bias is quickly overcome.

4.3.1 Summary

We have derived a single definition of relevance of representations that captures both task relevance (useful for value function and policy representations) and domain relevance (useful for capturing properties of the domain in a single function, for example a shaping function). In addition, we have formally defined how relevance changes with sample task sequences of increasing length k, and have shown that relevance converges to a fixed point in the limit. This property is exploited by the novel feature selection algorithm FS-TEK introduced here, which extrapolates change in relevance to predict true domain relevance.

Experiments have shown that FS-TEK compares favorably to selection methods that do not explicitly exploit the history of experienced tasks. As expected, the benefit is greatest when few tasks have been experienced, which is important in practice for jump-starting performance on new tasks. Although changes in potential function and increases in task and state space size affect the relative online return of FS-TEK compared to other

methods, FS-TEK's performance in terms of ROC remained constant. Results furthermore suggest that FS-TEK can be flexibly tuned for a low FPR or high TPR, according to what is best for the domain; IBKR, on the other hand, has more trouble in filtering out domain irrelevant features, as expected.

4.4 Related Work

Both feature selection (FS) and state abstraction methods have been applied to single-task RL, with especially FS seeing a recent surge in interest [51, 67, 81, 97, 101]. Although similar methods have also been applied to transfer learning, most of these learn task-relevant representations for supervised learning [2, 6] or reinforcement learning [34, 58, 71, 74, 129, 139, 147, 149]; the latter aim to reduce the state space of the target task, or find good representations for value functions or policies. However, none of these approaches learn domain-relevant representations.

In the supervised learning literature, work that does learn such representations includes lifelong [143] and multitask [14] learning. Both paradigms develop a representation that is shared between tasks by using training examples from a set of tasks instead of a single task, and show that this representation can improve performance by generalizing over tasks, if tasks are sufficiently similar. In RL, Foster and Dayan [36] identify shared task subspaces by identifying shared structure between task value functions; by augmenting the value function and policy representation with these subspaces, new tasks can be learned more quickly. While the idea of shared structure in task value functions is similar to ours, a limitation of this method is that it requires a single transfer function between tasks and only allows changes in the reward function. Similar to the domain-relevant features defined in this chapter and to the agent space of Konidaris and Barto [68], Frommberger and Wolter [38] define *structure space* as the feature space that retains the same semantics across tasks, and learn a structure space policy between tasks. Frommberger [37] applies the same concept to tile-coding functions for generalization within and across tasks. However, in both cases the structure space is hand-designed, as in [68].

While not explicitly concerned with representation learning, Sherstov and Stone [117] construct a task-independent model of the transition and reward dynamics by identifying shared outcomes and state classes between tasks, and using this for action transfer. This can be viewed as a kind of model-based domain relevance, and thus an interesting direction for future work on model-based MTRL.

There are two primary characteristics distinguishing our approach from the other cross-task methods discussed here. First, it captures within-task and cross-task relevance within a single definition. Second, it exploits multi-task structure by considering how representation relevance changes with increasing task sequence length and using that to predict relevance on the entire domain.

4.5 Conclusion and Discussion

We presented a unifying measure of relevance of MTRL representations based on the squared error in predicting cross-task value. This single definition can be used for finding both task-relevant and domain-relevant representations, and allows for approximately relevant representations that trade off error with compactness. It can also be used for context-dependent representations (abstractions that cluster based on feature values).

We captured the expected relevance of a representation on a sequence of k tasks in k-relevance, and showed formally that this converges to the domain relevance as k increases, and under certain conditions does so monotonically. This property can serve as a basis for more powerful multi-task representation learning algorithms. This chapter presents the first such algorithm, FS-TEK, which selects features by extrapolating k-relevance from the observed source sequence to form a better estimate of domain relevance. Using the algorithm for finding representations for the potential function generally benefited performance, although in the stock domain performance declined with increasing task space size. While this seemed due to the changing task dynamics rather than scalability, further studies on other domains are needed to verify this.

In our experience, FS-TEK's sole parameter, the confidence level, is an important tool for improving performance by tuning how aggressively FS-TEK discards features. In this sense, it is more an "aggressiveness" parameter than a confidence level, since setting this parameter to values near 10^{-3} is not uncommon. How to set the parameter is domain-dependent; increasing it tends to increase FPR but decrease TPR, and vice versa. Thus, the parameter setting should depend on whether it is more important to retain true positives or discard true negatives.

It is not immediately clear how to extend FS-TEK to non-binary linear or nonlinear value function approximators, since the abstractions that FS-TEK relies upon may not be well defined there. One approach could be to learn these approximators using supervised regression on multiple task solutions simultaneously on task sequences of increasing length, and extrapolate changes in feature weights to obtain a measure similar to k-relevance. Extending k-relevance and FS-TEK to these domains is an important direction for future work.

While FS-TEK is an algorithm for feature selection, our definition of relevance applies to more general abstractions, i.e., including context-dependent ones. Therefore, a promising direction for future work is to extend FS-TEK to discover such abstractions. One approach would be to employ multi-task regression trees, with extrapolated relevance as a measure for creating splits.

While we tested our algorithm and definition of domain-relevance on potential functions, it is likely that the definition is equally applicable to other related transfer approaches, such as advice and rule-based methods [145]. The hand-coded task-independent state-clusters that Taylor et al. [142] use for discovering rules are exactly the type of domain-relevant abstractions that an FS-TEK-like algorithm can discover. Another possible application is to function approximators such as cross-task tile coding [37], which is based on a *structure space* defined by domain-relevant features. Our definition and algorithm could therefore enable the automatic discovery and construction of the appropriate coding, instead of hand-designing it. Domain relevance might also be useful for constructing informed priors in Bayesian MTRL. For example, Wilson et al. [159] use a

hierarchical Bayesian approach in which distributions over tasks are drawn from a distribution over *classes* of tasks. Since purely domain-relevant features (i.e., features that are not task-relevant) similarly define task classes, they could form a useful basis for a prior.

5

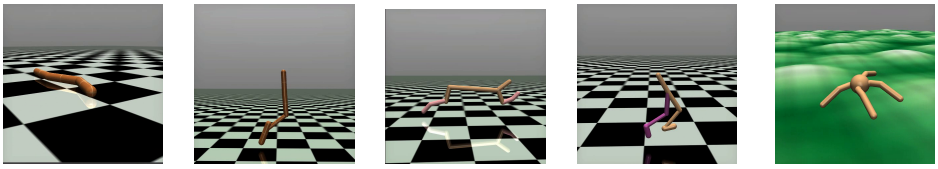
Benchmarking Recurrent Architectures for Robust Continuous Control

This chapter comprises the second part of the thesis, which explores a conceptually different approach to multi-task adaptivity. The previous two chapters focused on a setting in which tasks are MDPs sampled i.i.d. from an unknown distribution, and the agent is given the opportunity to learn each task to its best ability. While learning about dynamics changes – within or across tasks – is a natural approach, doing so may be expensive. On a real robot, for example, it may be both time-consuming and risky. Therefore, this chapter's approach to task generalization deviates from the previous ones, and from the rest of the literature on transfer learning, by investigating *fixed* controllers that nonetheless exhibit a degree of robustness to dynamics changes.

Fixed controllers, in the context of this chapter, are parameterized model-free policies π_θ of which the parameters θ are not updated after the agent is deployed for production purposes (θ may have been previously learned or designed offline). A robust controller is here defined as a controller of which performance, measured according to the objective function of the task on which the agent was trained, does not significantly deteriorate in the face of dynamics changes. Note that by model-free policy, we mean that the controllers we focus on are different from controllers in the control theory sense, which maintain a model of system dynamics. This is also what sets the work in this chapter apart from the control theory literature on robust control.

As an example application, imagine a robot controller trained to walk as quickly as possible in a given direction on a given terrain. At deployment, the learned policy is fixed, but the robot might have to deal with unforeseen circumstances: variations in terrain, sensor noise, failing actuators, etc. In this example, a controller is considered robust if it maintains speed and direction of locomotion, despite, for example, the terrain variations.

By virtue of their ability to exhibit rich dynamics and maintain internal state, recurrent neural networks (RNNs; see section 2.4) seem like an appropriate class of parameterized, learnable architectures for creating fixed, robust controllers. In supervised learning, RNNs, in particular the Long Short-Term Memory (LSTM, an RNN designed specifically to learn about long-term dependencies, and explained in more detail in Section 5.2.3) have achieved state-of-the-art results in for example machine translation [133] and speech recognition [43]; in RL, most recent work that learns the full RNN parame-



(a) Swimmer (b) Hopper (c) Half-Cheetah (d) Walker (e) Ant

Figure 5.1: Tasks. The ant is depicted on the hill task with difficulty 0.5.

terization[1] is also LSTM-based [5, 27, 53, 156].

This chapter makes the following contributions. Firstly, it presents the first benchmark study of modern RNNs in RL for continuous control, by extending the work of Duan *et al.* [27], on deep RL for continuous control, to a range of RNN architectures: an Elman network, continuous-time RNN (CTRNN), LSTM, GRU, and echo state network (these architecture will be explained in more detail in Section 5.2). While Duan's study benchmarked RL algorithms using a single FNN architecture, we benchmark RNN architectures using a single RL algorithm. We focus on a set of simulated locomotion tasks based on the MuJoCo simulator [144], using Trust Region Policy Optimization (TRPO) [114], which was found to perform best overall in Duan's study. We also include the deep FNN used by Duan, and an additional shallow feedforward neural network, as baselines. Results show that FNNs learn fastest, but test performance results using fixed, greedy policies show no single architecture outperforms the others.

Secondly, in addition to the regular tasks, controllers are also trained on a variation with rugged terrain. To test robustness, the best learned controller of each architecture is fixed and greedified, and tested on two types of perturbation: added sensor noise, and a switch from flat to rugged terrain. Here, the FNNs and CTRNN significantly outperform other architectures in terms of average robustness. Results also show that training on a hill task decreases the expected drop in performance due to perturbation for the RNNs, but not the FNNs.

Thirdly, all the RNN implementations and additional benchmark tasks constructed for this study are available at `github.com/sytham/rnn-benchmark`.

The remainder of the chapter is structured as follows. We start with sections that present the benchmarks tasks and architectures, followed by experimental results. The chapter concludes with a summary, discussion of findings, and suggestions for further work.

5.1 Tasks

Base tasks, used for learning without dynamics change, are the following five OpenAI Gym [12] environments[2], adopted and adapted from the study of Duan *et al.* [27]: Swimmer, Hopper, Half-Cheetah, Walker, and Ant. Agents are rewarded for moving forward

[1] This does not include the body of work that makes use of central pattern generators or other pre-designed recurrent systems and learns a limited set of parameters, e.g. [31].

[2] https://gym.openai.com/envs#mujoco

as quickly as possible, and penalized for energy expenditure (joint torque magnitude). Underlying dynamics are simulated by the MuJoCo engine [144]. See Figure 5.1.

In addition, we use the following variations on the base tasks to introduce changing dynamics.

Hills A hilly terrain parameterized by a difficulty parameter d, see Figure 5.1e for an example. The terrain is generated by a mixture of Gaussians with an average of 0.3 hilltops per unit square, each Gaussian with variance randomly sampled between 0.1 and 0.2 and 0 covariance. Height of the hills equals difficulty (note that higher difficulty implies both larger height and steepness). The hill tasks are used for both learning and testing fixed policies; in the latter case, terrain switches from flat to hilly at a given position.

Sensor noise Used for testing fixed policies, the noisy tasks inject white noise with $\sigma = 1.0$ into all sensors from a given timestep onwards. The effect of this depends on the sensor; for example, joint angles take values in $[-0.5\pi, 0.5\pi]$, while joint angular velocities have wider ranges. For simplicity, we chose a single noise value that represents a significant perturbation for most sensors.

5.1.1 Task details

Descriptions from Duan *et al.* [27], except the Swimmer, which me modified in order to also work on the hilly environment, and the Hopper, for which we used a lower "alive coefficient" (see description for more details).

Swimmer: The swimmer is a robot with 3 links and 2 actuated joints. Fluid is simulated through viscosity forces, which apply drag on each link, allowing the swimmer to move forward. This task is the simplest of all locomotion tasks, since there are no irrecoverable states in which the swimmer can get stuck, unlike other robots which may fall down or flip over. This places less burden on exploration. In the original task, the swimmer floats; we replaced this with what is perhaps more aptly called a glider: a swimmer on a frictionless plane (or frictionless hills for the hill environment). The 22-dim observation includes the robot's translation and rotation in three dimensions, joint angles, joint velocities, as well as the coordinates of the center of mass. The reward is given by $r(s,a) = v_x 0.005||a||_2^2$, where vx is the forward velocity. No termination condition is applied.

Hopper: The hopper is a planar monopod robot with 4 rigid links, corresponding to the torso, upper leg, lower leg, and foot, along with 3 actuated joints. More exploration is needed than the swimmer task, since a stable hopping gait has to be learned without falling. Otherwise, it may get stuck in a local optimum of diving forward. The 20-dim observation includes joint angles, joint velocities, the coordinates of center of mass, and constraint forces. The reward is given by $r(s,a) = v_x 0.005||a||_2^2 + 0.1$, where the last term is a bonus for being "alive". Duan *et al.* used an alive bonus of 1, but we found this to frequently result in networks that would learn to just stand upright without hopping. The episode is terminated when $z_{body} < 0.7$, where z_{body} is the z-coordinate of the body, or when $|\theta_y| > 0.2$, where θ_y is the forward pitch of the body.

Walker: The walker is a planar biped robot consisting of 7 links, corresponding to two legs and a torso, along with 6 actuated joints. This task is more challenging than

hopper, since it has more degrees of freedom, and is also prone to falling. The 21-dim observation includes joint angles, joint velocities, and the coordinates of center of mass. The reward is given by $r(s,a) = v_x 0.005||a||_2^2$. The episode is terminated when $z_{body} < 0.8$, $z_{body} > 2.0$, or when $|\theta_y| > 1.0$.

Half-Cheetah: The half-cheetah is a planar biped robot with 9 rigid links, including two legs and a torso, along with 6 actuated joints. The 20-dim observation includes joint angles, joint velocities, and the coordinates of the center of mass. The reward is given by $r(s,a) = v_x 0.005||a||_2^2$. No termination condition is applied.

Ant: The ant is a quadruped with 13 rigid links, including four legs and a torso, along with 8 actuated joints. This task is more challenging than the previous tasks due to the higher degrees of freedom. The 125-dim observation includes joint angles, joint velocities, coordinates of the center of mass, a (usually sparse) vector of contact forces, as well as the rotation matrix for the body. The reward is given by $r(s,a) = v_x 0.005||a||_2^2 C_{contact} + 0.05$, where $C_{contact}$ penalizes contacts to the ground, and is given by $5 \cdot 10^4 \cdot ||F_{contact}||_2^2$, where $F_{contact}$ is the contact force vector clipped to values between 1 and 1. The episode is terminated when $z_{body} < 0.2$ or when $z_{body} > 1.0$.

5.2 Architectures

This section describes the architectures used for comparison. We only provide the equations describing the input and hidden layers; the output layer for all architectures is the same as in Duan et al. [27] and the TRPO paper of Schulman et al. [114]: a linear map from the (last) hidden layer to the mean of a Gaussian distribution with zero covariance. The standard deviation of each element is specified by a separate parameter vector. Gradients for all architectures are calculated using the TRPO gradient definition (section 2.3.5) and the BPTT algorithm (section 2.4), except the continuous-time RNNs, for which continuous-time BPTT (section 2.4) is used.

5.2.1 DTRNN

A vanilla *discrete-time* RNN (DTRNN), also known as an Elman network [30], is the map

$$\mathbf{h}_n = \mathbf{f}(\mathbf{K}\mathbf{h}_{n-1} + \mathbf{W}\mathbf{x}_n), \tag{5.1}$$

where $\mathbf{K}$ is a pxp square matrix of recurrent connections, and $\mathbf{h}_n$ is the p-dimensional output of the hidden layer at update n. See Figure 5.2, panel A for a schematic representation of an example RNN.

This network architecture is included in the study because, as the "vanilla" architecture, it serves as a useful benchmark. In addition, we suspect that for the class of robotic locomotion tasks used for evaluation, long-term dependencies might be absent, and the more sophisticated machinery of the GRU and LSTM networks (explained below) might not be required.

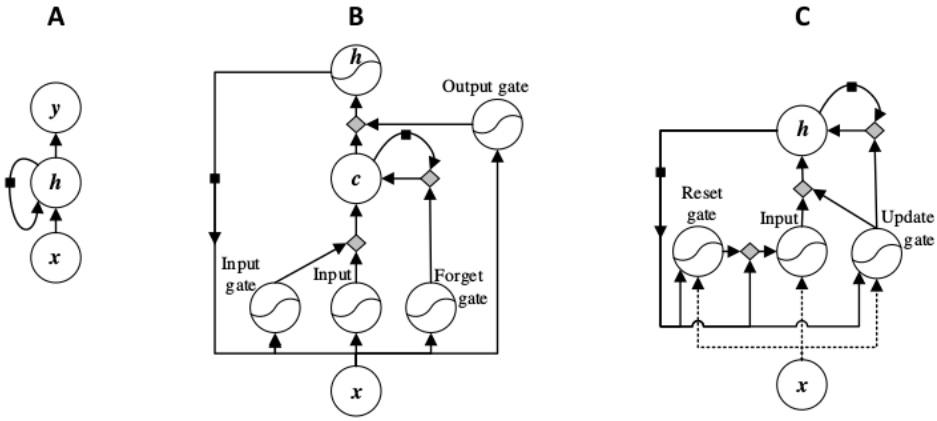


Figure 5.2: Schematic representation of architectures. A black square represents a time delay of one step; a wavy node represents nonlinearity. **A**: vanilla recurrent net. The recurrence is fully laterally and self-connected. An ESN has the same layout, but with the recurrent and input connections fixed (no learning). **B**: LSTM Cell. See main text. **C**: GRU Cell, where the input connections are dashed to highlight the distinction with the hidden recurrent connections. See main text for further details.

5.2.2 CTRNN

A vanilla *continuous-time* RNN (CTRNN) is a flow that in this chapter takes the form

$$\tau\dot{\mathbf{h}} = -\mathbf{h} + \mathbf{f}(\mathbf{K}\mathbf{h} + \mathbf{W}\mathbf{x}), \tag{5.2}$$

where τ is a time constant. This is a standard leaky integrator model (see e.g. [7, 26, 45]): it integrates input $\mathbf{f}(\cdot)$ over time and leaks activation through $-\mathbf{h}$. See Figure 5.2, panel A. Note that this is the exact continuous-time analog of the DTRNN; the first-order discrete-time approximation with timestep Δt comes down to $\mathbf{h}_{t+\Delta t} = \mathbf{h}_t - \frac{\Delta t}{\tau}\mathbf{h}_t + \frac{\Delta t}{\tau}\mathbf{f}(\cdot)$, which for $\Delta t = \tau$ reduces to (5.1).

Simulating continuous-time systems requires numerical integration of the differential equations, for example using an Euler or midpoint approximation. One can verify that taking the gradient of such an approximation for the forward equations results in the same updates as numerical integration of the differential equations for the gradient (see also Pearlmutter [98])[3]. This is therefore straightforward to implement in symbolic differentiation software (we used Theano): all that is needed is to include the numerical integration steps in the computation graph.

The vector of timeconstants τ is learned as well, constrained to be greater than or equal to the simulation timestep[4]. It is not immediately clear how to incorporate this constraint into TRPO, since it is fundamentally different from the KL constraint (for

[3]This is not always the case, though; for example, when using advanced numerical integration schemes that use an adaptive step size [99]. However, we use no such methods here.

[4]Theoretically, timeconstants need only be greater than 0, but a timeconstant smaller than the timestep leads to instability. Note that we set integration timestep = simulation timestep.

example, it does not make sense to incorporate it in the inverse Fisher matrix). We experimented with a number of approaches (one approach used a reparameterization of τ, another added a penalty to the loss), and in the end settled on the crudest one, which is equivalent to a rough gradient projection:

$$\tau = \max(\Delta t, \tau + \Delta\tau),$$

where $\Delta\tau$ is the update suggested by the learning algorithm. While this changes the descent direction, this approach surprisingly led to the best results.

We include CTRNNs because they seem particularly fit for this problem type, with its naturally smooth and continuous dynamics. In addition, CTRNNs allow to learn hidden node timeconstants explicitly, which may provide a simpler alternative to the dynamic gating provided by GRU and LSTM. Finally, because of their smoothness they may be more robust to disturbances.

5.2.3 LSTM

To date, one of the most successful recurrent architectures applied to sequential machine learning problems is the long short-term memory (LSTM) [40, 54] network. Explicitly designed to handle both long and short-term dependencies between input and desired output, it consists of "memory cells" where the recurrent connection is gated and dependent on input, allowing the network to adjust timescales dynamically. In addition, it has input and output gates to control propagation of input and output. Numerous variations on the basic theme exist; we use the same implementation as the one used by Duan *et al.*[5] [27]. See Figure 5.2, panel B. In the following it is convenient to think of the gating units arranged in their own layers, i.e., an input gate layer, a forget gate layer, and an output gate layer. For brevity, we also define the cell as a "gate layer", since it is updated by the same equation. Then output of each gate layer type is given by the usual

$$\mathbf{g}_n^i = \mathbf{f}(\mathbf{K}^i \mathbf{h}_{n-1} + \mathbf{W}^i \mathbf{x}_n),$$

where i is the gate type. Here, $\mathbf{f}$ is the logistic activation function $1/(1+e^{-x})$. Cell state is given by

$$\mathbf{c}_n = \mathbf{g}_n^{\text{forget}} \circ \mathbf{c}_{n-1} + \mathbf{g}_n^{\text{input}} \circ \mathbf{g}_n^{\text{cell}}, \tag{5.3}$$

where $\circ$ is the entrywise product, and hidden layer output by

$$\mathbf{h}_n = \tanh(\mathbf{c}_n) \circ \mathbf{g}_n^{\text{output}}.$$

Thus, the forget gate learns to decide when to "soft-clear" the current cell state, and the input gate learns to decide when to "soft-replace" cell content with new input. The idea is that in this way, the LSTM can hold on to interesting state features for arbitrarily long times, thereby allowing to learn long-term dependencies.

We included LSTMs since they are becoming the de-facto recurrent architecture for sequential tasks (together with GRU), and it should be enlightening to see how they fare on continuous control tasks.

[5] Duan did not use what are known as "peepholes"; neither do we, and to avoid clutter we do not explain them here.

One may wonder why the forget and input gate are separate, since it may seem that forgetting and replacing are related operations. This, indeed, is one of the ideas behind the GRU network.

5.2.4 GRU

The gated recurrent unit (GRU) network [16] is similar to the LSTM in its use of gating units. The two main differences with LSTM are that 1) a single unit does the job of LSTM's forget and input gate, by defining a weighted average between previous and "candidate" hidden state; and 2) it does not have a separate memory cell. See Figure 5.2, panel C. The GRU analog to (5.3) is

$$\mathbf{h}_n = \mathbf{g}_n^{\text{update}} \circ \mathbf{h}_{n-1} + (\mathbf{1} - \mathbf{g}_n^{\text{update}}) \circ \tilde{\mathbf{h}}_n,$$

with the candidate hidden state

$$\tilde{\mathbf{h}}_n = \tanh\left(\mathbf{g}_n^{\text{reset}} \circ \mathbf{K}\mathbf{h}_{n-1} + \mathbf{W}\mathbf{x}_n\right),$$

and the update and reset gate updated as in (5.2.3).

5.2.5 ESN

Echo state networks (ESNs; [57]) have the same architecture as vanilla recurrent networks, but with fixed input and hidden connections; only the (linear) output connections are subject to learning. ESN training views the network explicitly as a dynamical system and relies on creating the RNN dynamics by typically using a large number of hidden nodes (the "'reservoir"), and initializing the fixed connections in a way that is prone to creating rich dynamics. One way to view this approach [42] is as a kernel machine that transforms sequences of arbitrary length into a fixed-sized state vector $\mathbf{h}$; the linear transform from h to the output can then be learned with a simple learning algorithm.

ESN randomly initializes the hidden weights, possibly sparsely, and then scales them to obtain a given *spectral radius*: the largest absolute eigenvalue of the matrix of recurrent weights. I.e., it scales the weight matrix by $\frac{r}{|\lambda_{\max}|}$, where r is the desired spectral radius and $\lambda_{\max}$ is the largest current eigenvalue. On a high level, recall that the eigenvalues of a matrix are a measure of how much it contracts or stretches space; a $\lambda < 1$ contracts space along its corresponding eigenvector, while a $\lambda > 1$ stretches it. When $\lambda_{\max} < 1$, the mapping encoded by the network will tend to be contractive ("tend to" because the network's nonlinearities interfere with this), and the network will tend towards a fixed point, and quickly forget information about the past. On the other hand, with $\lambda_{\max} > 1$, the network will tend to magnify differences in input, and exhibit possibly oscillatory and/or chaotic dynamics. Because it is designed to exhibit rich dynamics from the start, and does not update the input and hidden weights, it hopefully remedies the "bifurcation issue" [24] in RNN training, where with a tiny parameter update a sudden, qualitative shift in network behavior occurs.

This network type is included in the study because of its growing popularity, and simplicity combined with its supposedly rich dynamics, which could enable it to adjust more easily to changing environmental dynamics.

In the learning experiments that follow, the spectral radius and layer density of the ESN are treated as hyperparameters.

5.3 Robustness

Controller robustness is measured per task pair $t = \{t_b, t_p\}$: on the one hand, greedy performance on the base task t_b, on the other hand, greedy performance on the corresponding perturbed task t_p (noise or terrain switch). Let $R_c(m)$ be the greedy return of controller c on task $m \in t$, and $R_{max}(t) = \max_{R_c}\{R_c(m) : c \in C, m \in t\}$ be the best performance across controllers on the task pair (essentially, just a benchmark score that could also be set manually). Then scaled return $r_c(m) = R_c(m)/R_{max}(t)$. Controller score $s_c(t) \in [0,1]$ on a given task pair is

$$s_c(t) = r_c(t_b)\Big[1 - \Big|r_c(t_p) - r_c(t_b)\Big|\Big], \tag{5.4}$$

where the absolute difference is the performance gap caused by the dynamics change. A score of 1 would be achieved by the controller that performs best on this task pair (either on the base or perturbed version), and is unaffected by the perturbation. Averaging this measure across tasks, $s_c = \frac{1}{T}\sum_{t \in T} s_c(t)$ is probably most informative.

Of course, this is just one of several possible "robustness" scores. In our eyes, the advantages are that it balances base performance with the performance delta caused by the dynamics change, that it can be averaged across tasks with different return ranges, and that it measures robustness in terms of the task's objective (for example, changing gait is in itself neither penalized nor incentivized). A possible downside is that controller score partially depends on other controllers. However, scoring a controller relative to itself has the downside that if it performs very poorly on the base task and equally poorly on the perturbed task, it nonetheless gets a high robustness score. Finally, note that this scoring method, combined with the reward criterion of the benchmark tasks, assigns a better score to a controller that e.g. significantly slows down when faced with a hill than to one that continues moving smoothly but in a perpendicular or reverse direction. In some scenarios, the latter may be preferred behavior. Nevertheless, this way of scoring is, again, in line with the task's objective.

5.4 Method

During learning, each controller is run on each of the base tasks for 1000 iterations of the learning algorithm. Results are averaged over 5 randomized runs, and performance is measured as the average of the average return per learning iteration. In addition to the RNNs, we also include the deep NN ("DFFWD") used by Duan *et al.*[6], and a shallow NN ("FFWD") with one hidden layer.

Number of hidden units N is fixed as follows. For LSTM and GRU, $N = \dim(\mathbf{a})$, the task action dimensionality; for DTRNN and CTRNN $N = 3 \times \dim(\mathbf{a})$, roughly the same

[6] Duan's study benchmarked algorithms, not architectures; they used the same deep NN architecture for the whole study, and one single LSTM architecture for POMDP experiments.

	CTRNN	DTRNN	GRU	ESN
Ant	0.3	0.2	0.05	0.09
Halfcheetah	0.5	0.1	0.07	0.02
Hopper	0.1	0.35	0.07	0.02
Swimmer	0.1	0.1	0.1	0.02
Walker	0.5	0.2	0.05	0.02

Table 5.1: Step sizes δ_{KL}; see Section 2.3.5 for further details. ESN spectral radius and hidden layer weight density were 1.1 and 1.0 respectively, except for Ant, where they were set to 1.5 and 0.7. For the Hopper and Swimmer tasks, all RNNs use a feature subset that does not include angular velocities (FNNs always use the full feature set for all tasks.)

number of free parameters as LSTM. For the same reason, for ESN $N = 3 \times \dim(\mathbf{o}) \times \dim(\mathbf{a}) + 4 \times \dim(\mathbf{a})^2$. DFFWD uses the same architecture reported by Duan *et al.* [27] ($100 \times 50 \times 25$); FFWD gets the same number as DTRNN. Further hyperparameters are tuned by running a coarse search, averaging over 2 randomized runs per set of hyperparameter values. The exception to this are the FNNs and LSTM, for which the settings reported by Duan *et al.* were used. In addition, for each task we test whether RNNs perform better with a subset of features. See Table 5.1 for detailed settings.

Next to the base tasks, each architecture is also trained on the hill version of each task with difficulty 0.5, with the same hyperparameters as for the base task.

Testing uses the best learned policy for each architecture and sets the standard deviation of the output layer to 0. Thus, for each architecture there are 2 controllers to be tested: one that was trained under regular conditions (flat terrain), and one that was trained on hill terrain. Each controller is tested on 3 conditions: 1 "regular" test, running the greedy policy on the task it was trained on; and 2 robustness tests, 1 for noise perturbation, and 1 for a terrain switch. Each test consists of 20 randomized runs of 1000 steps; for the hill environments, testing difficulty is set to 1.0, each network is tested on the same 20 randomly generated terrains, and terrain switches from flat to rugged at a task-dependent position. For the sensor noise test, white noise with $\sigma = 1$ is introduced at step 100.

5.5 Results

This section presents and discusses the results of training and testing the RNNs on the set of tasks described in section 5.1. It is divided in two sections; the first one discusses the learning results, and the second one the testing of fixed, greedy policies under changing dynamics.

5.5.1 Learning

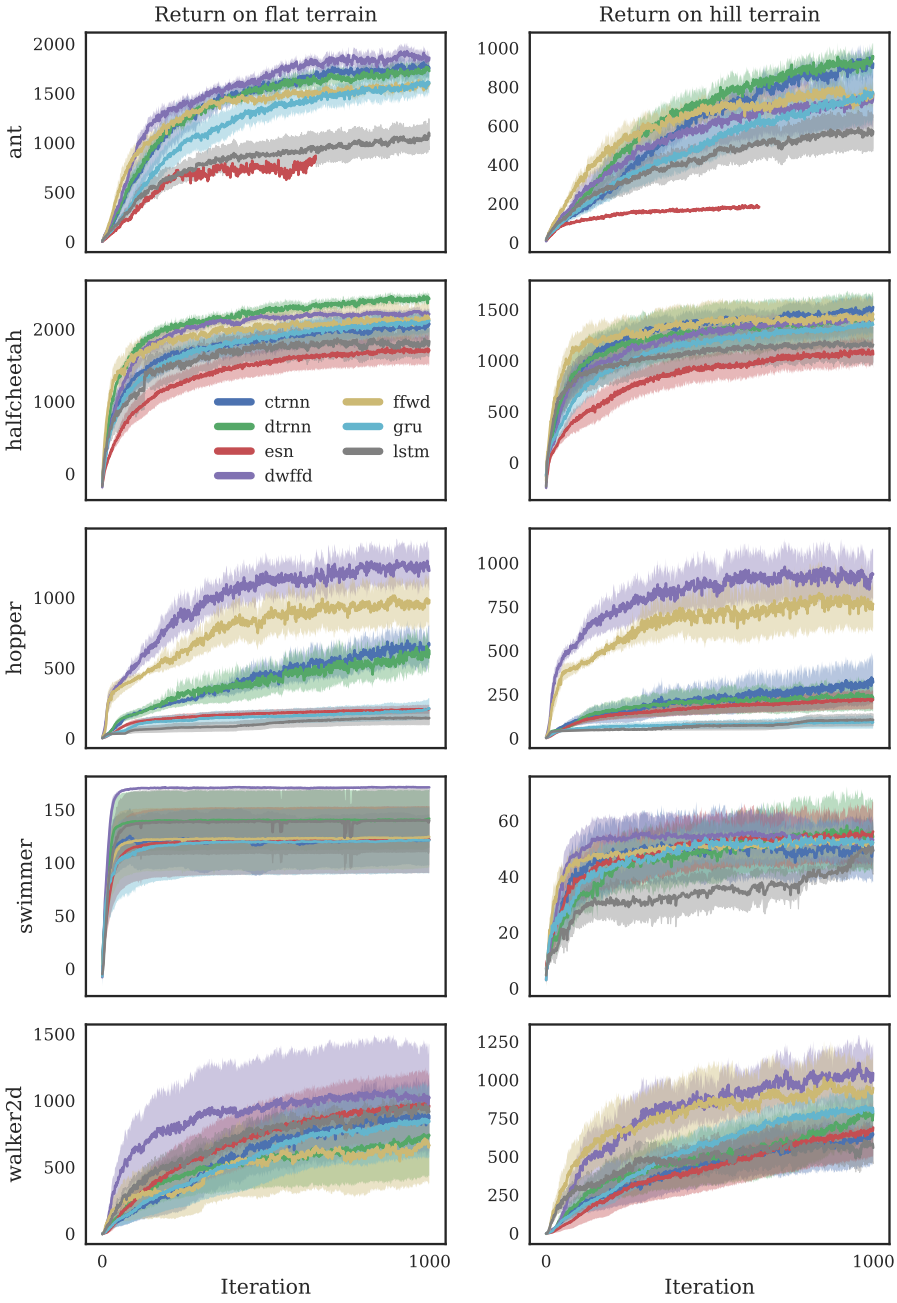


Figure 5.3: Mean learning curves. Left: learning on flat terrain. Right: learning on hilly terrain with difficulty=0.5. See Table 5.2 for quantitative data. The swimmer was run for only 500 iterations.

	CTRNN		DTRNN		ESN		DFFWD		FFWD		GRU		LSTM	
Ant	1072.5	(109.4)	1072.9	(30.1)	544.9*	(0.0)	**1212.1**	**(41.9)**	**1118.9**	**(38.2)**	999.2	(62.0)	551.4	(109.4)
Halfcheetah	1533.7	(167.3)	**1892.6**	**(62.9)**	1141.2	(130.9)	**1787.7**	**(76.6)**	**1765.8**	**(156.9)**	1275.3	(156.0)	1205.2	(252.4)
Hopper	266.9	(25.0)	273.4	(60.8)	134.3	(18.8)	**792.1**	**(73.4)**	601.6	(55.4)	143.4	(41.9)	69.4	(21.4)
Swimmer	118.8	(22.7)	137.4	(24.7)	117.0	(27.9)	**167.8**	**(0.2)**	120.1	(26.4)	109.2	(31.3)	135.2	(26.3)
Walker2d	362.3	(120.2)	392.0	(134.8)	469.7	(138.3)	**748.3**	**(256.9)**	364.0	(171.8)	233.4	(120.1)	517.3	(61.3)
Ant hill	362.0	(30.6)	**443.4**	**(52.0)**	135.6*	(0.0)	393.3	(69.8)	**476.1**	**(45.8)**	222.7	(37.2)	233.7	(51.4)
Halfcheetah hill	**1094.3**	**(109.7)**	**1053.4**	**(202.1)**	649.6	(106.2)	**991.8**	**(96.0)**	**1177.9**	**(142.1)**	**934.1**	**(109.3)**	**916.4**	**(145.2)**
Hopper hill	95.7	(22.5)	139.1	(47.1)	54.9	(28.4)	**704.8**	**(90.3)**	542.8	(50.1)	108.5	(38.5)	34.8	(3.4)
Swimmer hill	47.1	(9.8)	46.4	(8.9)	50.7	(13.3)	51.8	(6.5)	48.8	(4.2)	43.3	(8.6)	37.4	(0.0)
Walker2d hill	282.8	(86.4)	336.0	(107.5)	255.4	(92.2)	**595.0**	**(128.1)**	**619.0**	**(161.3)**	362.3	(67.3)	374.1	(67.6)

Table 5.2: Mean return per learning iteration step, up to the 500th iteration to be consistent with Duan *et al.*. Bold entries are significantly better than others (Kruskal-Wallis followed by Welch t-test with $p < 0.05$).
*Run could not be completed because machine ran out of disk space.

	CTRNN		DTRNN		ESN		DFFWD		FFWD		GRU		LSTM	
Ant	3214.6	(51.7)	3155.4	(64.3)	934.3	(36)	2307.7	(39.4)	2329.8	(21.7)	**3567.1**	**(52.3)**	1843.1	(35.8)
Halfcheetah	4382.4	(41.9)	**4888.9**	**(69.0)**	3105.1	(57.9)	3256.9	(8.9)	3692.0	(31.2)	3423.1	(45.1)	2510.1	(70.8)
Hopper	**2331.4**	**(64.8)**	**2149.7**	**(98.2)**	280.5	(8.7)	1665.3	(40.4)	1762.9	(23.4)	350.2	(6.8)	265.1	(3)
Swimmer	349.8	(0.3)	353.5	(0.2)	**355.5**	**(0.2)**	346.4	(0.2)	318.8	(0.8)	351.6	(0.2)	**355.3**	**(0.3)**
Walker	1409.2	(32.3)	1895.5	(21.1)	1717.6	(16.4)	**2075.1**	**(34.0)**	**2152.4**	**(43.8)**	1596.6	(79.5)	1148.1	(26.9)
Ant hill	1041.7	(47.3)	1482.9	(44.4)	196.9	(5.7)	1234.3	(61.8)	1872.0	(63.1)	**2245.3**	**(22.6)**	955.0	(12.8)
Halfcheetah hill	**3923.1**	**(22.6)**	3530.8	(55.1)	2396.0	(49.7)	2097.8	(19.3)	2613.0	(83.1)	1953.0	(15.9)	3110.6	(38.9)
Hopper hill	1157.6	(55)	420.8	(24.4)	265.8	(19)	1395.0	(56.9)	**1939.0**	**(24.4)**	84.1	(11.8)	136.8	(5.6)
Swimmer hill	231.9	(2.1)	**275.6**	**(1.4)**	240.0	(1.2)	113.7	(1.5)	177.0	(1.8)	137.2	(2.8)	225.3	(2.2)
Walker hill	1482.1	(70.6)	1195.6	(34.2)	1184.6	(29.4)	1400.2	(4.8)	**2555.8**	**(77.4)**	1115.8	(31.2)	974.4	(31.5)

Table 5.3: Mean and stderr of 1000-step return over 20 randomized runs with fixed weights on flat terrain.

Overall learning performance is presented in Table 5.2, which shows the mean and standard error of the mean return per training iteration. Figure 5.3 plots the mean learning curves. Unfortunately, DFFWD results cannot be directly compared to those of Duan et al.[27]: on two of the tasks, we made modifications (Swimmer and Hopper), and on others, changes have since been made to the robot model and/or software. Overall, the feedforward architecture learns fastest, while the DTRNN is on average the best RNN. The relatively strong performance of the two vanilla architectures compared to the two gated architectures is likely due to the fact that on this class of tasks the influence of long-term dependencies is minor, partly because of the cyclic nature of the task, and partly because of the reward density (an informative reward is received at every timestep).

The CTRNN results shown here were obtained with the midpoint numerical integration method; we found this to perform significantly better than simple Euler on most tasks, for the cost of increased training time.

Given that it only learns the output weights, the ESN performance, while mostly underperforming, may be called impressive. Contrary to what one may expect, a downside of this method used in conjunction with TRPO is that space and computation requirements are significant, due to its large reservoir. On the Ant task, where the network has 3328 hidden units, this caused the machine to run out of disk space after a wallclock time of about 38 hours. We also briefly attempted to train the ESN with SGD and vanilla policy gradient, but this led to significantly worse results.

Sutskever *et al.* [132] applied the ESN spectral radius initialization method to initialize the weights of a vanilla DTRNN with 100 hidden units and found that this, in combination with the Nesterov momentum, significantly improved performance on a set of tasks with long-range dependencies. We experimented with ESN initialization on the DTRNN as well, but did not find an improvement in performance on this set of tasks, possibly because the network is too small (see Section 5.4 for a description of network size depending on task; maximum DTRNN size is 24 hidden units).

Finally, it appears that the Hopper task is particularly difficult for RNNs to learn, with only the vanilla architectures showing some promise. One reason may be that the mixed dynamics (cyclic foot movement with an upper body that mostly needs to be kept upright and still) present difficulties. This would be interesting to further investigate as a class of dynamics that is difficult to learn for RNNs.

5.5.2 Testing

Overall results of running the fixed, greedy policies on each base task are presented in Table 5.3. For results on all tasks, see Appendix C. Due to the fact that the RNNs are more difficult to train, but narrow the gap near the end of training, relative performance of the FNN drops significantly. In addition, RNNs exhibit somewhat greater variance during learning, so the best RNN run may outperform the best FNN run, even though average learning performance is lower. Contrary to expectations (due to partial observability), the FNN performance on rugged terrain is still more or less on par with that of the RNNs.

Fig. 5.4 displays results of the robustness tests. Note that scores can not be compared across quadrants, only per quadrant, since robustness is a relative score per task pair (see Section 5.3). Noise perturbation results show that while there is no single best architecture on all tasks, the flat-trained CTRNN and DFFWD significantly outperform

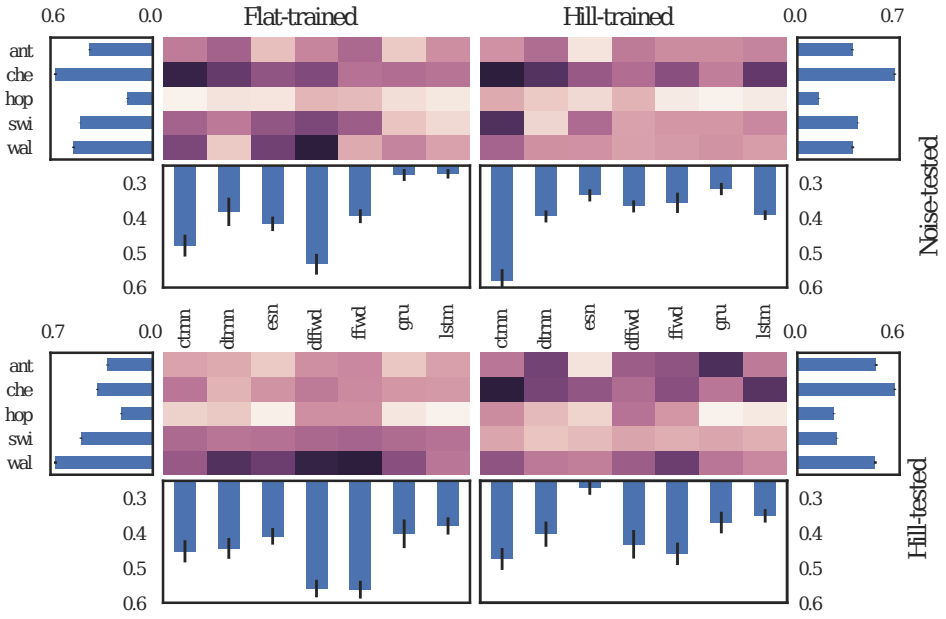


Figure 5.4: Test results on perturbation tasks. In each quadrant, the heatmap shows robustness per controller per task (darker is higher); bottom panel is average score per controller, side panel is average score per task. Note that scores can not be compared across quadrants, since robustness is a relative score per task pair (see Section 5.3). Left (right) half: controllers trained on flat (rugged) terrain. Top half: results on tasks with noise perturbation. Bottom half: results on tasks with a switch from flat to rugged terrain. Confidence intervals obtained by a bootstrap with $N = 1000$.

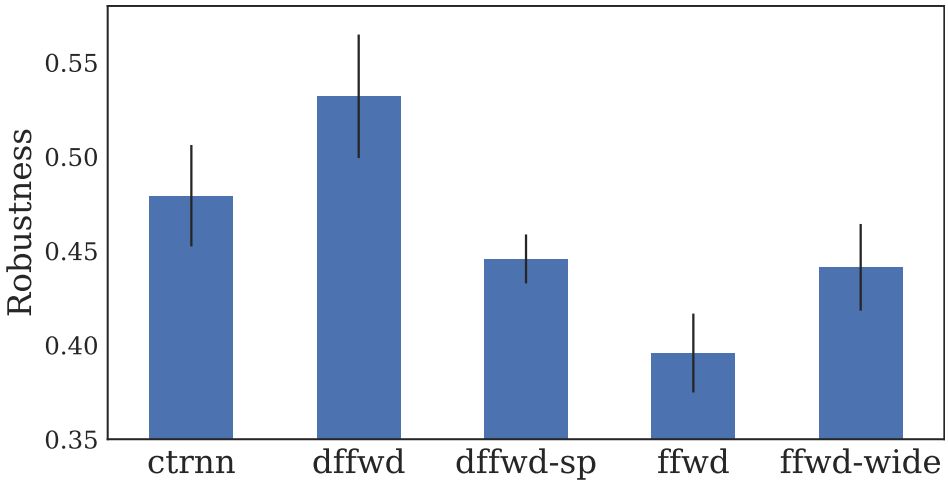


Figure 5.5: Robustness of flat-trained controller on noise perturbation, comparing to a deep FNN with softplus activation ("dffwd-sp") and a wide, shallow FNN ("ffwd-wide")

the other networks on average. The superior performance of the deep network with respect to the shallow network is somewhat surprising; in order to determine whether this is due to network depth or higher number of parameters, we trained a one-layer network with 400 nodes and tested it on robustness. We further compared DFFWD to a deep FNN with the same architecture but the softplus activation function $f(x) = \log(1 + \exp(x))$, resulting in the score shown in Fig. 5.5. While the truth seems to be somewhere in the middle, it appears that the benefit is in network depth combined with the hyperbolic tangent transfer function, perhaps due the squashing properties of the latter.

When trained on rugged terrain, the CTRNN significantly outperforms all others. We hypothesize that by training on rugged terrain, the networks encounter a larger selection of states, which reduces overfitting and increases robustness; beneficial results of noise injection and/or wider initial state distribution have been shown before [65, 103]. In addition, RNNs trained on flat terrain have little incentive to learn to maintain internal state; training on rugged (or any kind of POMDP) terrain might provide this incentive, which carries over to other types of tasks where this is useful. Training on rugged terrain also helps the gated RNNs, which perform quite poorly when trained on flat terrain. Superior performance of the CTRNN over the others can be explained by the fact that the CTRNN is stiff in time[7], and thus acts as a more efficient low-pass filter for the high-frequency perturbation that white noise represents.

One can see the effect of hill-training more clearly as follows. If hill training would have no effect, one would expect noise perturbation to cause the same relative drop in performance on flat-trained controllers as on hill-trained controllers. Therefore, the ratio of the drops should show how much effect hill-training has had; if it is 1, there is no discernible effect. Table 5.6a shows bootstrap estimates of this ratio per controller; for

[7] I.e., it imposes a certain smoothness on its output in the sense that, loosely speaking, outputs that are close in time should also be close to each other

	2.5 pct	97.5 pct
CTRNN	1.31	1.68
DTRNN	1.97	2.64
ESN	1.02	1.33
DFFWD	0.85	1.19
FFWD	0.95	1.20
GRU	1.55	2.59
LSTM	1.82	2.54

(a) Bootstrap estimate ($N = 1000$) of mean improvement ratio on noise perturbation.

	2.5 pct	97.5 pct
CTRNN	1.14	1.61
DTRNN	1.76	2.57
ESN	1.04	1.49
DFFWD	1.00	1.39
FFWD	0.76	0.99
GRU	1.12	1.76
LSTM	1.03	1.33

(b) Bootstrap estimate ($N = 1000$) of mean improvement ratio on hill perturbation.

Figure 5.6: Mean improvement ratios on perturbation tasks by training on hills.

the feedforward architectures, there is no significant effect, providing evidence for the hypothesis that RNNs are helped by learning to maintain internal state. The ESN improvement is also low, since this architecture only learns the output weights and not the recurrent weights.

On the hill-test scenario, the FNNs trained on flat terrain performs surprisingly well (lower left quadrant of Fig. 5.4). Partly based on visualization of rollouts, we speculate that on the hill test, the quality of the original gait learned by the controller is more important than on the noise test, and perhaps more important than the controller itself. Another supporting argument for this idea is that the hill environment is locally flat; (some limbs of) the robot body may be inclined at an angle, but correlations between joint angles and velocities should be largely preserved, and more similar to observations seen during training than for sensor noise, which decorrelates inputs. Since FNNs learning results are quite stable, this may give it an edge over the other controllers. For the hill-trained networks, results on this scenario are less distinct, with the FNN and CTRNN nonetheless significantly outperforming the rest. The strong performance of the FNN could be due to the limited information contained in the information history. If this is the case, then training networks on noisy tasks or other "stronger" partially observable MDPs should result in a larger performance difference. Hill training has similar effects on hill perturbation as on noise perturbation; see Table 5.6b.

It is also clear, however, that some tasks lend themselves much more readily to learning robust policies; for example, Halfcheetah scores highly in this regard, while Hopper proves particularly difficult. As mentioned before, one reason that sets Hopper apart may be that the mixed dynamics (cyclic foot movement with an upper body that mostly needs to be kept upright and still) present difficulties. We can think of no obvious reason why Halfcheetah controllers should be more robust than on other robots. Ant controllers trained on hills seem robust as well (ESN performance may be considered an outlier due to the technical issues there), which may make sense given the Ant body morphology.

5.6 Related Work

There has been an explosion of work on RNNs, especially LSTM and GRU, in the supervised learning literature. We pick a few key works and ones that are similar to ours. Chung et al. [17] compared performance of GRU, LSTM and DTRNNs on sequence modeling tasks, specifically music and raw speech signal modeling. They found that the architectures perform similarly on the music task, but in raw speech signal modeling, where long-term dependencies supposedly play a larger role, the gated units outperformed the DTRNN. Jozefowicz et al. [59] performed an evolutionary search over RNN architectures and an ablative study of LSTM in order to find out which components are required, and found that adding a bias of 1 closes the performance gap between GRU and LSTM. Sutskever et al. [132] use an initialization method inspired by echo state networks (ESNs), a network type also included in this comparison study, to improve the performance of vanilla RNNs (see Section 5.2.5 for further details).

One of the earliest works that employed RNNs in RL[8] was performed by Robinson and Fallside [105], who use a recurrent neural network to provide a reinforcement signal to another network encoding the policy, akin to an actor-critic setup. However, their method only takes into account the dependency of reward on past inputs and outputs, i.e., it ignores state transitions. Schmidhuber [109, 110] builds upon this by using a similar setup with two (recurrent) networks, but by employing an adaptive state transition model in order to compute reward gradients, and applying this architecture to a partially observable pole-balancing task. Lin and Mitchell [78] compared Q-learning with an Elman RNN, history window and model on two partially observable tasks and found that the recurrent approach outperformed the others for tasks with relatively short-term dependencies and reward delay.

Bakker [5] was the first to use the LSTM introduced by Hochreiter and Schmidhuber [54] for reinforcement learning to memorize important past observations. Wierstra et al. [156] develop an LSTM-based recurrent policy gradient, and Hausknecht and Stone [53] use it in a value function-based approach. Huh and Todorov [55] use a CTRNN for optimal control of a robot arm in a reaching and drawing task. For a more comprehensive overview of RNNs and deep FNNs in RL, see Schmidhuber [111]. We are not aware of a comparative study of modern RNNs in RL for continuous control (not considering work on central pattern generators and other fully or partly pre-designed recurrent systems).

Finally, our work builds on, and is closely related to, the benchmark study of Duan *et al.* [27], who benchmark deep reinforcement learning algorithms on a large set of tasks that also includes the robotic locomotion tasks studied in this chapter. While Duan *et al.* study only feedforward architectures and the LSTM, this chapter extended the study (and the software) to additional RNN types, and studied the robustness of the fixed, greedified policies represented by the networks under various changes in dynamics.

[8] As pointed out by Schmidhuber [111], two separate networks feeding into each other, as used by some earlier works in RL, are essentially also an RNN. Nonetheless, we do not review that work here. We also do not review search-based methods, such as using evolutionary algorithms to evolve RNN weights.

5.7 Conclusion and Discussion

There are a few key conclusions from the results presented here.

Simple architectures seem more robust. While certainly powerful, gated RNNs are not a panacea. In line with the case made by Rajeswaran et al. [103], simpler architectures, such as the vanilla RNNs and FNNs used here, may perform equally well and/or be more robust, depending on the task at hand; this certainly seems to hold for "pure" locomotion tasks with a limited amount of features and no requirement for more higher-level planning, where long-term dependencies might come into play.

CTRNNs perform well. CTRNNs perform well on this class of tasks, are fairly robust to changing dynamics, and can readily be combined with state-of-the-art policy gradient methods. Implementation in symbolic differentiation software is relatively straightforward if a fixed stepsize for numerical integration is used. We think this is exciting and worthy of further study, since we did not spend a significant amount of time on finding the best possible integration with TRPO.

Training on one class of POMDPs may help RNNs perform better on another class of POMDPs. Results on noise perturbation showed that training on hills helped all RNN architectures, while not helpful to FNNs, supposedly by providing the RNNs with an "incentive" to learn recurrent connections that capture history. This may be especially useful to cheaply train RNNs to be more robust to situations that are expensive or difficult to train on. For example, injecting noise during training may be a cheap method to make RNNs more robust to different terrain types.

One should be careful with generalizing the conclusions from this chapter, however, since a number of caveats apply. Firstly, as also pointed out by Islam et al. [56], hyperparameters used here can significantly affect performance, and even though having an openly accessible set of benchmark tasks greatly helps objectivity, comparison of results is made difficult by this sensitivity. In addition, and perhaps more importantly, each architecture this work studied can be tweaked in numerous ways (LSTM, in particular). It is impractical to explore and compare all of these in a single study. Indeed, an important open question seems how to carry out such a comparison efficiently across studies. Furthermore, as discussed in Section 5.5.2, there is significant variance in both performance and robustness across tasks. Therefore, either an as wide as possible range of tasks should be used, or one should resort to optimizing algorithms and architectures geared towards a particular task. Hopper, in particular, seemed to represent a special class of dynamics in this study. One reason for that may be that the mixed dynamics (cyclic foot movement with an upper body that mostly needs to be kept upright and still) present difficulties.

Secondly, as discussed in Section 5.3, the measure of robustness employed here places particular emphasis on balancing robustness with raw performance, and staying in line with the task objective. While we believe this is useful, other measures, which might for example place stronger emphasis on gait stability, might lead to different results and insights.

Lastly, it is difficult to say how well these results generalize to transfer to a real robot, or indeed to other simulators. RL methods have a way of exploiting simulator idiosyncrasies [65], and we have on occasion observed behavior that to the eye seemed unrealistic (see video for an example).

These caveats notwithstanding, we are convinced that this work sheds some much-

needed light on performance of different RNN architectures in RL for continuous control,
and hope that these results and the reference implementation at `github.com/sytham/rnn-benchmark`
will form the basis for further discussion and study.

6 Conclusions and Future Work

A reinforcement-learning agent learns through trial and error by interacting with the environment and observing the effect of its actions and the reward that it receives after each action. Generally, the goal of the agent is to identify the sequence(s) of actions that lead to the maximal sum of rewards, or maximal *return*. For practical purposes, it is useful if the agent can generalize from tasks it has solved to new, but similar, tasks it might encounter in the future. This thesis investigates two strategies for generalization in reinforcement learning.

In the first, we automatically learn a shaping function that captures invariant properties of the domain. We empirically evaluated three different approaches to learning the shaping function, and show that which is best depends highly on the domain, learning algorithm, and learning parameters. In addition, we presented a novel feature selection algorithm, FS-TEK, that extracts the invariant properties of the domain that the shaping function can be based on. We demonstrate empirically the benefit of FS-TEK on four artificial domains.

The second generalization strategy focuses on neural controllers that exhibit robustness to changes in task dynamics. We show that feedforward neural networks (FNNs) and a continuous-time recurrent neural network (RNN) are most robust to dynamics changes on average, with the CTRNN significantly outperforming the others under sensor noise perturbation. In addition, we show that training on a hill task decreases the expected drop in performance due to dynamics changes for the RNNs, but not the FNNs.

The following sections answer the research questions posed in Chapter 1 of the thesis, and propose directions for future work.

6.1 Evaluation of Research Questions

What kind of targets could a cross-task shaping function approximate, and under which settings do these targets perform well?
We evaluated three target choices:

- **The optimal value function of each task**. While this choice may seem natural, it did not outperform the other choices on the domains we evaluated. One pitfall of this target that we observed empirically is that it may be too optimistic; this may

cause the agent to over-explore the task, which is detrimental in high-risk tasks where some actions lead to large negative reward.

- **The approximate value function of each task**. This may be a value function approximator, or the value function based on a soft policy, such as in Sarsa. On the domains we evaluated, this choice performed roughly on par with or better than the other choices. However, in high-risk domains such as the continuing cliff domain, on-policy methods such as Sarsa may converge to the wrong value function, in which case this choice may be detrimental.

- **The value function of the optimal cross-task policy**. Fundamentally different from the other two choices, this option is the value function corresponding to a *single* cross-task policy, which is analogous to the value function of a memoryless policy on a POMDP. While it is in line with a cross-task shaping function, this option did worse on average. However, on a high-risk domain we observed it to perform well, because it is more risk-averse than the other two choices.

These choices of target, while not the only possible ones, are natural: the first two constitute the actual solutions to the tasks that a value-function based agent will produce, depending on its learning algorithm; and the last is, just like the shaping function, a single cross-task function.

What state representation should a cross-task shaping function be based on, and what distinguishes this state representation from the representation that should be used for the value function of each individual task?
What we call *task-relevant* representations are representations that, when used for the value function of a task, incur low or zero expected error with respect to the true value function of the task (i.e., the one based on the full state representation). That is, the features in this representation are predictive of return within a task. Therefore, the value function or policy of an individual task should be based on a task-relevant representation.

Domain-relevant representations are predictive of return *across* tasks in the domain. In other words, when given a task from the domain, it is possible to predict how return on that task is expected to vary with the domain-relevant representation before seeing the task. This is not possible with a representation that is not domain relevant. Being a cross-task function, the shaping function should be based on a domain-relevant representation: its main goal is to help the agent learn a given task more quickly, by providing the agent with synthetic rewards right from the start of the task.

Is it possible to develop an algorithm for finding both kinds of representation?
Yes. We introduce k-relevance, ρ_k, which unifies task and domain relevance: it is the expected relevance of a representation on a sequence of k tasks sampled from the domain. Task relevant representations have $\rho_k > 0$ for $k = 1$, and domain relevant representations have $\rho_k > 0$ as k tends to infinity; we have shown formally that k-relevance does indeed converge asymptotically. Using this property, we have devised a novel feature selection algorithm, FS-TEK (Feature Selection Through Extrapolated k-relevance), which looks at how representation relevance changes with growing task sequence length, and

uses that to select a domain relevant representation after observing each new sampled task. Empirical results demonstrate that FS-TEK significantly outperforms the other feature selection methods we studied for a small number of observed tasks. It therefore improves the agent's generalization ability with limited data (tasks). While FS-TEK focuses on domain relevant representations, one can use the concept of k-relevance equally well for finding task relevant representations, by averaging over relevance on each individual task observed so far.

How do RNN architectures relate to each other in terms of learning performance, and why?
Of the neural network architectures evaluated in Chapter 5 of this thesis, on the robotic locomotion tasks we studied, a deep feedforward network has the best learning performance in terms of mean return per learning iteration step. A shallow feedforward architecture is a close second. These networks have better learning performance than RNN architectures due to the inherent challenge of training RNNs: since they share a single weight matrix across many timesteps, an update to the weights can have undesired consequences many timesteps later (or earlier). On the tasks we studied, long-range dependencies and partial observability are limited, allowing FNNs to do better during learning.

Of the RNN architectures, the simplest architecture, the DTRNN, has on average the best learning performance. Since long-range dependencies and partial observability are limited, and the DTRNN has the simplest gradient of all architectures except the ESN, it learns fastest. While ESN has a simpler gradient (only based on the output layer), it implicitly needs to learn about the dynamics of the high-dimensional state space represented by its hidden nodes.

How do RNN architectures relate to each other in terms of testing performance when the policies are fixed, and why?
Testing the fixed, greedified policies showed mixed results, with no single architecture (RNN or FNN) outperforming the others. The reason that this is different from the learning results is that, as mentioned, RNNs are harder to train, but the result of training might be equivalent or better than an FNN network. As of yet, it is unclear what task properties cause one network type to do better than another. This is an interesting direction for future work.

How robust is each of the RNN architectures to a change in dynamics in the form of sensor noise and a terrain switch?
On average, the FNN architectures and the continuous-time RNN are most robust, according to our measure of robustness. In addition, we show that training on a hill task decreases the expected drop in performance due to perturbation for the RNNs, but not the FNNs, by providing the RNNs with more incentive to learn to maintain internal state. The two types of perturbation represent different classes of dynamics: sensor noise is high-frequency, and decorrelates inputs, while hill terrain varies more slowly and is locally flat. Since continuous-time RNNs are stiff in time, they appear to be better-equipped than the other networks for smoothing sensor noise, provided they have been trained appropriately. On the hill perturbation task, the FNNs showed strong performance. The

CTRNN was only able to match this when trained on the hill task. Due to the more slowly varying nature of the hill perturbation, the quality of the original gait learned on the task is more important, and since FNNs learn faster, with more stable gaits on average, they do well on this class of dynamics change.

6.2 Future Work

This section presents a few of the directions for future work we think are most important.

- **Multi-task shaping in practice**. As Chapter 3 has shown, the choice of target for the cross-task shaping function to approximate can have a significant impact on learning performance. Applying this insight in practice comes with some challenges. For example, a Q-Learning agent outputs Q^* as task solutions, so using these as target will present no additional overhead. Future research could address efficient methods for using as target a value function that is not produced by the learning agent; one idea could be for the agent to learn multiple value functions simultaneously.

  In addition, the domains used in the chapter, while useful to demonstrating the idea, are artificial and small-scale. Future work should investigate how well these ideas scale up to larger domains.

- **"Universal" cross-task representation learning**. Chapter 4 demonstrated the benefit of our definitions of relevance and FS-TEK on cross-task shaping functions for table-based and linear value functions. There are at least two areas of research for making this work more generally applicable: other types of cross-task functions, and other function approximators.

  In the first category, we think it is likely that our definition of domain relevance also applies to related transfer approaches, such as advice and rule-based methods. In addition, shaping is closely related to inverse reinforcement learning (e.g. [39]), in which a reward function to guide the agent is inferred from demonstrations. Learning reward functions that are robust to changes in task is challenging; for example, Fu et al. [39] observed the problem of learning a reward function that is robust to changes in the transition function. Using better cross-task representations could help alleviate this problem. Future work could investigate if and how our definition of relevance generalizes to these other applications.

  With regards to the second category, it is not yet clear how well our definition of relevance generalizes to for example nonlinear function approximators. In addition, FS-TEK in its current form discards features as a whole, whereas our definition of relevance also applied to context-dependent abstractions such as for example used in trees.

- **Continuous-time RNNs for RL**. Chapter 5 demonstrated that continuous-time RNNs (CTRNNs) perform strongly on simulated locomotion tasks, and are fairly robust to task perturbations, in particular sensor noise. While there is work on

continuous-time systems in RL, for example for solving continuous-time MDPs [10, 25], or using central pattern generators [31], most of these approaches require specialized algorithms. If a fixed stepsize for numerical integration is used, a simpler approach can be taken, which is the one we have implemented in the chapter: adding the numerical integration steps to the computation graph, which can then be differentiated using symbolic differentiation software. While this is not as general as other approaches, it does allow one to easily integrate CTRNNs with state-of-the-art policy gradient algorithms. We think it would be exciting to further study the use of CTRNNs in RL, and explore other domains in which they can be useful.

- **Robust controllers in hierarchical RL**. A potential benefit of robust controllers that this thesis has not explored yet is their use in hierarchical learning systems: a robust low-level controller could (partially) shield high-level learning modules from changes in task dynamics. For example, in a task where a robot needs to navigate a maze, locomotion could be controller by a low-level controller, while the higher-level module learns to navigate the maze by modulating the low-level controller. We explored such an application previously in the context of central pattern generators [128], but the RNNs investigated here could provide a simpler alternative.

A

Stationary Memoryless Multi-Task Policies

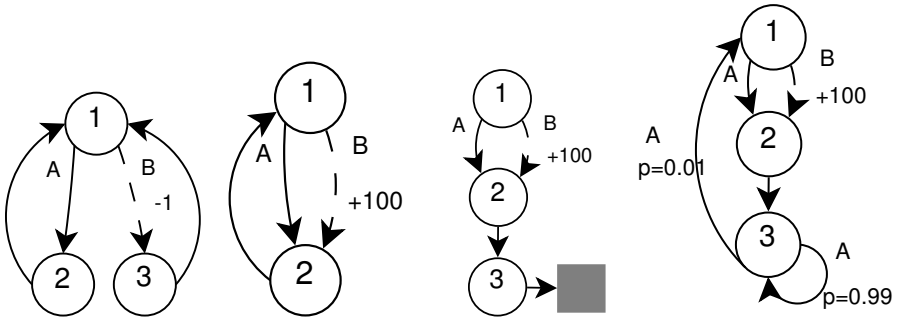
Section 3.3.5 derived a value function for a stationary memoryless cross-task policy μ. Such a policy assigns the same probability to a given state-action pair, regardless of the current task or history. In this sense, it is similar to a stationary memoryless policy for a POMDP where tasks serve as hidden states, or to such a policy for a fixed belief. Singh et al. [124] analyzed stationary memoryless policies for POMDPs, and showed that the best stationary stochastic policy may be better than the best stationary deterministic policy and there need not exist a stationary policy (stochastic or deterministic) that maximizes the value of each state simultaneously. This is important for defining what it means for a stationary memoryless cross-task policy to be optimal. The same facts apply to the multi-task case; however, since the proofs from Singh et al. do not translate without modification to the multi-task case, we have modified those proofs appropriately and reproduce them here. Recall from Section 3.3.5, Eq. 3.8 that the cross-task value of a state under a stationary policy μ is defined as

$$Q_d^\mu(\mathbf{x}) = \sum_{m \in \mathsf{M}} p(m|\mathbf{x}) Q_m^\mu(\mathbf{x}).$$

Fact 1. *: In a multi-task scenario, the best stationary stochastic policy can be arbitrarily better than the best stationary deterministic policy. (Fact 2 from [124]).*

Proof. : Consider Fig. A.1. We denote by $V_m(s)$ and $V(s)$ the value of state s in task m and in the domain, respectively. Without loss of generality, consider the deterministic

Figure A.1: A domain with two tasks, both with absorbing state 2. In the leftmost task, action A in state 1 transitions to state 1 and incurs reward $-R$. In the rightmost task, the effects of actions A and B are swapped with respect to task 1.



(a) Task 1 (b) Task 2a (c) Task 2b (d) Task 2c

Figure A.2: Two-task domains. Each domain is composed of task 1 and a variation of task 2. In each of these, there need not exist a stationary policy that maximizes the value of each state simultaneously. The only policy decision, between action A (solid) and B (dashed), is taken in state 1. $+x$ and $-x$ denotes reward. $p = x$ denotes transition probability.

policy μ that always chooses action A, and assume both tasks are equally likely. Then the domain value of state 1, $V^{\mu}(1) = -\frac{1}{2}\frac{R}{1-\gamma} + \frac{1}{2}\cdot 0 = -\frac{R}{2-2\gamma}$. Under a stochastic policy $\pi(A|1) = \pi(B|1) = 0.5$, for a single task $V_m^{\pi}(1) = -\frac{1}{2}R + \frac{1}{2}\gamma V_m^{\pi}(1)$. Therefore for both a single task and for the domain, $V^{\pi}(1) = -\frac{R}{2-\gamma} > V^{\mu}(1)$ for $\gamma > 0$. □

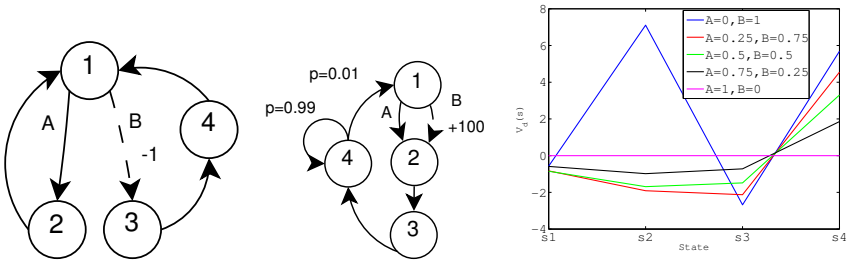
Fact 2. *For each of the following cases:*

1. *domains where tasks need not have the same state spaces;*
2. *domains where tasks need not be ergodic;*
3. *the distribution $p(\mathbf{x}|m)$ is uniform, except where $\mathbf{x} \notin \mathbf{X}_m$, in which cases it is 0;*
4. *the distribution $d(m|s)$ is policy-dependent,*

there need not exist a stationary policy that maximizes the value of each state simultaneously.

Proof. : See Figure A.2. The only policy decision is made in state 1, where either action A or B can be chosen. In Fig. A.2a and A.2b, the tasks have dissimilar state spaces: state 3 does not occur in task 2a. Increasing the probability of choosing action B in state 1 increases the value of state 1 but decreases the value of state 3; increasing the probability of choosing action A in state 1 has the opposite effect. Therefore, in this particular example there is no policy that maximizes the value of all states simultaneously. So in general in domains where tasks may have dissimilar state spaces there *may* not be such a policy.

In Fig. A.2c, the tasks have the same state spaces, but task 2b is not ergodic. The same effect as in Fig. A.2a can be observed. Lastly, in Fig. A.2d the tasks have the same state space, and are ergodic (for any policy). If we now define $d(m|s)$ to be uniform and



(a) Task 1 (b) Task 2 (c) Some policies

Figure A.3: Domain composed of task 1 and 2 for which there does not exist a stationary policy that maximizes the value of each state simultaneously, given that value is averaged over tasks using a policy-dependent distribution.

policy-independent, increasing the probability of choosing action B or A in state 1 has the same effect as before.

The last case is slightly less straightforward. See Fig. A.3. Values for five policies are plotted in Fig. A.3c. For the policy that always takes action A in state 1, values are flat at 0. Increasing the probability of taking B decreases the values of all states but increases the value of state 4. This happens because in task 2, the value of s4 increases because of the (large) increase in s1. Since $d(m_2|s_4,\mu) \gg d(m_1|s_4,\mu)$, $V_d^\mu(s_4)$ increases; $V_d^\mu(s_1)$ decreases since $d(m_2|s_1,\mu) \ll d(m_1|s_1,\mu)$. Further increasing the probability of B increases s4 further. From around $p(B) = 0.95$, $V_d^\mu(s_2) > 0$, since $d(m_1|s_2,\mu)$ goes to 0; however, $V_d^\mu(s_3)$ keeps decreasing. □

B
Proofs

B.1 Theorem 2

Theorem 2. *Let ϕ be an abstraction with abstract Q-function as in definition 2. Let $\rho_k = \rho_k(\phi)$ for any k, based on the MSE abstraction error as in definition 3. Let $d(x,y) = |x-y|$ be a metric on $\mathbb{R}$, and let $f(\rho_k) = \rho_{k+1}$ map k-relevance to $k+1$-relevance. Then f is a strict contraction; that is, for $k > 1$ there is a constant $\kappa \in (0,1)$ such that*

$$d(f(\rho_k), f(\rho_{k-1})) \leq \kappa d(\rho_k, \rho_{k-1}).$$

B.1.1 Preliminaries

As stated in Theorem 2, we are concerned with the case where the abstract Q-function is defined as in (4.1), i.e., the weighted average over state-action pairs in a given cluster. In this case, relevance equals the sum of weighted variances of the Q-values of ground state-action pairs corresponding to a given cluster $\mathbf{y}$ (4.4). Before proving the theorems, we show how relevance can be rewritten as a sum of covariances between Q-functions.

Variance of a weighted sum of n correlated random variables equals the weighted sum of covariances. We start by showing that any Q_c is a weighted sum of random variables (namely the Q-functions of each task in the sequence), and that therefore relevance can be written in terms of a weighted sum over covariances. Equation 4.5 is already a weighted sum, but we require a constant weight per random variable (task). Thus we rewrite (4.5) as

$$Q_c(\mathbf{x}) = \sum_{i=1}^{k} p(c_i|c) Q^c_{c_i}(\mathbf{x}) \tag{B.1}$$

$$Q^c_{c_i}(\mathbf{x}) = \frac{p(\mathbf{x}|c_i)}{p(\mathbf{x}|c)} Q_{c_i}(\mathbf{x}), \tag{B.2}$$

where the last line is just a rescaling of Q_{c_i} depending on c, and $k = |c|$, the sequence length. Similarly, we define

$$Q^c_{\mathbf{y},c_i}(\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{y},c_i)}{p(\mathbf{x}|\mathbf{y},c)} Q_{c_i}(\mathbf{x}), \tag{B.3}$$

where $p(\mathbf{x}|\mathbf{y},c_i) = 0$ for all $\mathbf{x} \notin \mathbf{X}^{\mathbf{y}}_{c_i}$, as the values of $Q^c_{c_i}$ on the domain $\mathbf{X}^{\mathbf{y}}_{c_i}$.

For ease of notation, we write $\mathrm{Var}(Q_c)$ for the variance $\mathrm{Var}(Q_c(\mathbf{X}_c))$, leaving the domain implicit, and similarly for the covariance. Note that $p(c_i|c) = 1/k$. Then relevance (4.4) can be written as

$$\begin{aligned}\rho(\phi, Q_c) &= \sum_{\mathbf{y}\in\mathbf{Y}_c} p(\mathbf{y}|c)\,\mathrm{Var}\left(\frac{1}{k}\sum_{i=1}^{k} Q^c_{\mathbf{y},c_i}\right) \\ &= \frac{1}{k^2}\sum_{\mathbf{y}\in\mathbf{Y}_c} p(\mathbf{y}|c)\sum_{i=1}^{k}\sum_{j=1}^{k}\mathrm{Cov}\left(Q^c_{\mathbf{y},c_i}, Q^c_{\mathbf{y},c_j}\right) \end{aligned} \tag{B.4}$$

To see how relevance changes from one sequence to the next, we need to know how the covariance between two given tasks changes. For this purpose it is easier to write the covariance as $\mathrm{Cov}(X_i,X_j) = \mathrm{E}[X_iX_j] - \mathrm{E}[X_i]\,\mathrm{E}[X_j]$; then all we need to do is quantify both expectations. Let (c,m) be the new sequence formed by appending a task $m \in \mathsf{M}$ to a given sequence c. In the following, for ease of notation, assume an abstraction that leaves the original Q-function intact, i.e. $Q^c_{\mathbf{y},c_i} = Q^c_{c_i}$. The results can be extended to general abstractions by substituting $\mathbf{X}_c = \mathbf{X}^{\mathbf{y}}_c$, $\mathbf{X}_m = \mathbf{X}^{\mathbf{y}}_m$, $p(\mathbf{x}|c) = p(\mathbf{x}|\mathbf{y},c)$, and $p(\mathbf{x}|m) = p(\mathbf{x}|\mathbf{y},m)$.

B.1.2 Change in $\mathrm{E}[Q^c_{c_i}]$

Lemma 1. *The expected value of a given Q-function for a given task c_i in any sequence c is the same as the expected value of the original Q-function on c_i. That is,*

$$\mathrm{E}[Q^c_{c_i}] = \mathrm{E}[Q_{c_i}]. \tag{B.5}$$

Proof. The expected value of any $Q^c_{c_i}$ is

$$\begin{aligned}\mathrm{E}[Q^c_{c_i}] &= \sum_{\mathbf{x}\in\mathbf{X}_c} p(\mathbf{x}|c)Q^c_{c_i}(\mathbf{x}) \\ &= \sum_{\mathbf{x}\in\mathbf{X}_{c_i}} p(\mathbf{x}|c)Q^c_{c_i}(\mathbf{x}) + \sum_{\mathbf{x}\in\mathbf{X}_c/\mathbf{X}_{c_i}} p(\mathbf{x}|c)Q^c_{c_i}(\mathbf{x}) \\ &= \sum_{\mathbf{x}\in\mathbf{X}_{c_i}} p(\mathbf{x}|c)\frac{p(\mathbf{x}|m)}{p(\mathbf{x}|c)}Q_{c_i}(\mathbf{x}) \qquad (\text{Since } Q^c_{c_i} = 0\ \forall \mathbf{x} \notin \mathbf{X}_{c_i}) \\ &= \sum_{\mathbf{x}\in\mathbf{X}_{c_i}} p(\mathbf{x}|m)Q_{c_i}(\mathbf{x}) = \mathrm{E}[Q_{c_i}]. \end{aligned}$$

□

B.1.3 Change in $\mathrm{E}[Q^c_{c_i}Q^c_{c_j}]$

To put bounds on the change in $\mathrm{E}[Q^c_{c_i}Q^c_{c_j}]$, we have

Lemma 2. *For a given sequence c and new sequence (c,m) formed by appending a task $m \in \mathsf{M}$ to c,*

$$0 \le \left|\mathrm{E}\left[Q^{(c,m)}_{c_i}Q^{(c,m)}_{c_j}\right]\right| \le \frac{k+1}{k}\left|\mathrm{E}\left[Q^c_{c_i}Q^c_{c_j}\right]\right|,$$

where $|\cdot|$ denotes absolute value.

Proof. Let $Q_{i\cdot j}(\mathbf{x}) = Q_{c_i}(\mathbf{x})Q_{c_j}(\mathbf{x})$. Then

$$\begin{aligned}\mathrm{E}[Q^c_{c_i}Q^c_{c_j}] &= \sum_{\mathbf{x}\in\mathbf{X}_c} p(\mathbf{x}|c)Q^c_{c_i}(\mathbf{x})Q^c_{c_j}(\mathbf{x}) \\ &= \sum_{\mathbf{x}\in\mathbf{X}_c} \frac{p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)}{p(\mathbf{x}|c)} Q_{i\cdot j}(\mathbf{x}).\end{aligned}$$

For a given task pair, the only quantity that changes from one sequence to the next is $p(\mathbf{x}|c)$. Let $f^c_{i,j}(\mathbf{x}) = p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)/p(\mathbf{x}|c)$, and recall that, for a sequence of length k, $p(\mathbf{x}|c) = 1/k\sum_{i=1}^k p(\mathbf{x}|c_i)$. Therefore, on a new sequence (c,m):

$$\begin{aligned}f^{(c,m)}_{i,j}(\mathbf{x}) &= \frac{p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)}{(\sum_{i=1}^k p(\mathbf{x}|c_i) + p(\mathbf{x}|m))/(k+1)} \\ &= \frac{(k+1)p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)}{kp(\mathbf{x}|c) + p(\mathbf{x}|m)}.\end{aligned}$$

Taking the ratio of $f^{(c,m)}_{i,j}$ and $f^c_{i,j}$:

$$\begin{aligned}f^{(c,m)}_{i,j}(\mathbf{x})/f^c_{i,j}(\mathbf{x}) &= \frac{(k+1)p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)}{kp(\mathbf{x}|c) + p(\mathbf{x}|m)} \times \frac{p(\mathbf{x}|c)}{p(\mathbf{x}|c_i)p(\mathbf{x}|c_j)} \\ &= \frac{kp(\mathbf{x}|c) + p(\mathbf{x}|c)}{kp(\mathbf{x}|c) + p(\mathbf{x}|m)}. \qquad \text{(B.6)}\end{aligned}$$

If $p(\mathbf{x}|m)$ is larger (smaller) than $p(\mathbf{x}|c)$, this ratio is smaller (larger) than 1. It is largest when $p(\mathbf{x}|m) = 0$, namely $(k+1)/k$, and at its smallest it is

$$\lim_{p(\mathbf{x}|c)\downarrow 0} \frac{kp(\mathbf{x}|c) + p(\mathbf{x}|c)}{kp(\mathbf{x}|c) + p(\mathbf{x}|m)} = 0.$$

Since $Q_{i\cdot j}(\mathbf{x})$ is constant from one sequence to the next, this leads to the bounds as stated in the lemma. □

Note that especially the lower bound is quite loose, since usually $p(\mathbf{x}|c)$ will not be that close to 0. However, for our present purposes this is sufficient.

B.1.4 Proof of Theorem 2

We need to show that $|\rho_{k+1} - \rho_k| < |\rho_k - \rho_{k-1}|$ for any $k > 1$. The relevance of a given sequence consists of the sum of the elements of the covariance matrix for that sequence, where each element has weight $1/k^2$. As illustrated in Fig. 4.2, from one sequence c to a new sequence (c,m), the ratio of additional covariances formed by the new task m with the tasks already present in c is $(2k-1)/k^2$ and thus rapidly decreases with k. The same figure also shows that change in relevance is caused by two factors: the expansion of the covariance matrices as sequence length increases coupled with the change in sequence

probability, and change in the covariance between a given task pair from one sequence to the next. Suppose that the covariance of any task pair does not change from one sequence to the next. Then clearly, since the ratio of new covariance matrix elements changes with k as $(2k-1)/k^2$ and in addition the probability of all new sequences (c,m) formed from a given c sums up to the probability of c, $|\rho_{k+1}-\rho_k| \leq |\rho_k - \rho_{k-1}|$ for any $k > 1$.

Now suppose that covariances do change from one sequence to the next. As Lemma 1 and 2 show, the maximum change in covariance from any sequence c of length k to the next is $(k+1)\,\mathrm{Cov}(Q^c_{c_i}, Q^c_{c_j})/k$ for any i and j. This change also decreases with k, and therefore $|\rho_{k+1}-\rho_k| \leq |\rho_k - \rho_{k-1}|$ for any $k > 1$. If $|\rho_2 - \rho_1| = 0$, then by this property the difference must stay 0 and $|\rho_{k+1}-\rho_k| = 0$ for any k. In all other cases, the change in relevance is a strict contraction, $|\rho_{k+1}-\rho_k| < |\rho_k - \rho_{k-1}|$, by the above arguments. □

B.2 Theorem 3

Theorem 3. *If all tasks share the same distribution over state-action pairs $p(\mathbf{x}|m)$, then ρ_k, as defined in Theorem 2, is monotone.*

B.2.1 Ratio of Variances to Covariances

In the following lemma, for clarity we distinguish between variance and covariance by calling variance *type 1 covariance*, i.e. $\mathrm{Cov}(Q^c_{c_i}, Q^c_{c_j})$, $c_i = c_j$, and covariance *type 2 covariance*, i.e. $\mathrm{Cov}(Q^c_{c_i}, Q^c_{c_j})$, $c_i \neq c_j$. For $k = 1$, k-relevance consists solely of type 1 covariances.

Lemma 3. *The ratio of the number of type 1 covariances to the number of type 2 covariances decreases with k. For a given sequence c of length $k-1$, the ratio of new type 1 covariances in all new sequences of length k formed from c is*

$$\frac{2(k-1)+N}{N(2k-1)} \tag{B.7}$$

where $N = |\mathsf{M}|$.

Proof. Let c be any sequence on a domain with $N = |\mathsf{M}|$ tasks. Assume task $m \in \{1,2,\ldots,N\}$, occurs $o_m \in \{0,1,\ldots,k\}$ times in c. Then any c can be represented by an N-dimensional vector $\mathbf{o}$: $\mathbf{o} = (o_1, o_2, \ldots, o_N)$. Note that for a given sample size k, $\sum_i o_i = k$ for any c. Lastly, denote by σ the sum of elements in the last column and row of the covariance matrix – as shown in Fig. 4.2, these are the elements added from one sequence c to the next (c,m).

Now take any sequence c of length $k-1$, with task counts in vector $\mathbf{o}$. Form N new sequences of length k, where each sequence is formed by adding a task from M to c. To see how the ratio of covariances changes between $k-1$ and k, all that matters is the ratio in σ. For any new sequence formed by adding task m to sequence c, there will be $2o_m+1$ type 1 covariances in σ. Hence, in total, taken over the N new sequences formed from c, there will be $2(o_1+o_2+\ldots+o_N)+N = 2(k-1)+N$ new type 1 covariances. In total,

taken over the N new sequences, there are $N(2k-1)$ covariances. So the ratio of type 1 covariances in σ for a given sample size k is

$$\frac{2(k-1)+N}{N(2k-1)} \tag{B.8}$$

This ratio decreases with k. Therefore the ratio of type 2 covariances increases with k. □

B.2.2 Proof of Theorem 3

The assumption of a single distribution over state-action pairs implies that covariances do not change from one sequence to the next. This follows from Lemma 1 and Eq. B.6: since $p(\mathbf{x}|m) = p(\mathbf{x}|c)$, the ratio resolves to 1 and $\mathrm{E}\left[Q_{c_i}^{(c,m)} Q_{c_j}^{(c,m)}\right] = \mathrm{E}\left[Q_{c_i}^{c} Q_{c_j}^{c}\right]$.

The rest of the proof is by cases. Given $k = 1$, ρ_{k+1} can either be smaller than, greater than, or equal to ρ_k.

Case 1: $\rho_2 < \rho_1$.
Since $\rho_2 < \rho_1$, it follows that the expected value of a type 2 covariance is lower than that of a type 1 covariance: ρ_1 is made up of all possible type 1 covariances in the domain, while ρ_2 in addition consists of all possible type 2 covariances. Since covariances do not change from one k to the next, type 2 covariances must be lower on average. From Lemma 3, the ratio of type 2 covariances increases with k. Within the type 2 covariances, the frequency of a given task pair does not change, and the same holds for the type 1 covariances. Therefore, since covariances do not change with k, ρ_k must get ever lower with k, and ρ_k is monotonically decreasing with k.

Case 2: $\rho_2 > \rho_1$.
By a similar argument to that for case 1, ρ_k is a monotonically increasing function of k.

Case 3: $\rho_2 = \rho_1$.
Therefore $|\rho_2 - \rho_1| = 0$, and $|\rho_{k+1} - \rho_k|$ must stay 0 by Theorem 2, which shows that ρ_k is constant.

C

Full Test Results from Chapter 5

	CTRNN		DTRNN		ESN		DFFWD		FFWD		GRU		LSTM	
Ant	3214.6	(51.7)	3155.4	(64.3)	934.3	(36)	2307.7	(39.4)	2329.8	(21.7)	**3567.1**	**(52.3)**	1843.1	(35.8)
Halfcheetah	4382.4	(41.9)	**4888.9**	**(69.0)**	3105.1	(57.9)	3256.9	(8.9)	3692.0	(31.2)	3423.1	(45.1)	2510.1	(70.8)
Hopper	**2331.4**	**(64.8)**	**2149.7**	**(98.2)**	280.5	(8.7)	1665.3	(40.4)	1762.9	(23.4)	350.2	(6.8)	265.1	(3)
Swimmer	349.8	(0.3)	353.5	(0.2)	**355.5**	**(0.2)**	346.4	(0.2)	318.8	(0.8)	351.6	(0.2)	**355.3**	**(0.3)**
Walker	1409.2	(32.3)	1895.5	(21.1)	1717.6	(16.4)	**2075.1**	**(34.0)**	**2152.4**	**(43.8)**	1596.6	(79.5)	1148.1	(26.9)
Ant hill	1041.7	(47.3)	1482.9	(44.4)	196.9	(5.7)	1234.3	(61.8)	1872.0	(63.1)	**2245.3**	**(22.6)**	955.0	(12.8)
Halfcheetah hill	**3923.1**	**(22.6)**	3530.8	(55.1)	2396.0	(49.7)	2097.8	(19.3)	2613.0	(83.1)	1953.0	(15.9)	3110.6	(38.9)
Hopper hill	1157.6	(55)	420.8	(24.4)	265.8	(19)	1395.0	(56.9)	**1939.0**	**(24.4)**	84.1	(11.8)	136.8	(5.6)
Swimmer hill	231.9	(2.1)	**275.6**	**(1.4)**	240.0	(1.2)	113.7	(1.5)	177.0	(1.8)	137.2	(2.8)	225.3	(2.2)
Walker hill	1482.1	(70.6)	1195.6	(34.2)	1184.6	(29.4)	1400.2	(4.8)	**2555.8**	**(77.4)**	1115.8	(31.2)	974.4	(31.5)

Table C.1: Mean and stderr of 1000-step return over 20 randomized runs with fixed weights on flat terrain.

	CTRNN		DTRNN		ESN		DFFWD		FFWD		GRU		LSTM	
Ant	1309.3	(40.2)	1651.7	(58.1)	334.4	(5.7)	899.0	(49.7)	1393.4	(37.6)	682.0	(29.3)	782.4	(27)
Halfcheetah	3774.2	(121.8)	3261.5	(161.1)	2531.1	(85.9)	2790.4	(55.5)	1725.1	(46.7)	1785.1	(60.1)	1886.3	(70.8)
Hopper	151.7	(7.4)	112.2	(4.3)	152.8	(7.9)	151.8	(2.4)	142.5	(2.2)	86.8	(2.3)	148.8	(4.2)
Swimmer	176.8	(2)	150.6	(2.2)	197.7	(1.6)	213.9	(1.4)	173.3	(1.2)	69.3	(1.3)	51.9	(3.5)
Walker	1305.4	(30.7)	199.2	(3.6)	1302.5	(40.7)	1743.9	(69.8)	603.7	(41.3)	615.0	(30.8)	231.8	(20.5)
Ant hill	584.7	(25.1)	955.7	(30.6)	162.1	(5.5)	855.9	(39.8)	681.3	(37)	875.8	(33.1)	997.6	(15.9)
Halfcheetah hill	3559.6	(82)	3040.9	(38.6)	2359.0	(63.2)	1838.6	(63.2)	2569.7	(44.2)	1488.1	(40)	2911.0	(32.6)
Hopper hill	154.7	(6.5)	69.1	(3.5)	97.1	(6.8)	154.2	(3.7)	131.9	(2.4)	44.6	(6.2)	88.0	(3.8)
Swimmer hill	218.6	(2)	38.3	(2)	126.8	(1.7)	46.6	(1.2)	51.6	(1.9)	57.4	(1.5)	84.7	(2.4)
Walker hill	1378.8	(40.7)	671.9	(19.9)	640.0	(45.1)	319.9	(13.2)	857.2	(28.2)	673.6	(16.4)	586.6	(15.7)

Table C.2: Mean and stderr of 1000-step return over 20 randomized runs with fixed weights on noise perturbation.

	CTRNN		DTRNN		ESN		DFFWD		FFWD		GRU		LSTM	
Ant	1076.0	(63.4)	961.2	(41.7)	621.9	(45.2)	1071.8	(46.3)	1209.2	(44.1)	877.3	(43.2)	807.6	(28.3)
Halfcheetah	2290.8	(84.2)	1499.5	(42.5)	1375.4	(57.7)	1978.5	(65.9)	1639.7	(55.7)	1306.6	(30.7)	1359.7	(48.5)
Hopper	476.2	(14.7)	443.0	(10.1)	300.4	(10.3)	730.3	(36.9)	746.4	(15.5)	319.0	(10.5)	318.8	(11.9)
Swimmer	195.6	(5.7)	181.2	(6)	188.7	(5.7)	197.2	(6.2)	193.7	(4.7)	193.7	(5.2)	185.9	(7)
Walker	1444.7	(33.4)	1779.8	(36.2)	1817.0	(29.1)	1968.4	(41.9)	2063.0	(44.6)	1638.0	(60.8)	1212.7	(20.1)
Ant hill	832.9	(40.5)	1434.8	(60.1)	204.2	(11.9)	1403.2	(45.2)	1094.8	(42.1)	1603.4	(59.1)	835.9	(23.4)
Halfcheetah hill	3138.3	(56.3)	2268.9	(113.3)	1924.8	(69.6)	1471.3	(77.6)	2025.7	(75.8)	1304.1	(55.5)	2587.5	(77.1)
Hopper hill	354.4	(21.1)	350.7	(14)	338.8	(13.4)	612.1	(31.9)	599.5	(17.9)	80.7	(13.1)	200.6	(9.6)
Swimmer hill	43.2	(2.9)	47.4	(2.4)	28.9	(2.2)	19.7	(2.6)	4.8	(3.3)	8.5	(2.3)	29.3	(2.2)
Walker hill	1413.5	(58.7)	857.0	(33.2)	1608.8	(37.5)	1214.6	(44.8)	1632.7	(49.1)	982.2	(36.1)	829.8	(31.1)

Table C.3: Mean and stderr of 1000-step return over 20 randomized runs with fixed weights on hill perturbation.

Bibliography

[1] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *NIPS*, pages 1–8, 2006.

[2] A. Argyriou, T. Evgeniou, and M. Pontil. Convex multi-task feature learning. *Machine Learning*, 73 (3):243–272, 2008.

[3] J. Asmuth, M. Littman, and R. Zinkov. Potential-based shaping in model-based reinforcement learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 604–609. The AAAI Press, 2008.

[4] M. Babes, E. M. de Cote, and M. L. Littman. Social reward shaping in the prisoner's dilemma. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 1389–1392, 2008.

[5] B. Bakker. Reinforcement learning with long short-term memory. In *Advances in neural information processing systems*, pages 1475–1482, 2002.

[6] J. Baxter. A model of inductive bias learning. *J. Artif. Intell. Res. (JAIR)*, 12:149–198, 2000.

[7] R. D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3 (4):469–509, 1995.

[8] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[9] D. P. Bertsekas. *Dynamic programming and optimal control*. Athena, 1995.

[10] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in neural information processing systems*, pages 393–400, 1995.

[11] R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv:1606.01540*, 2016.

[13] T. Brys, A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, and A. Nowé. Reinforcement learning from demonstration through shaping. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3352–3358. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL `http://dl.acm.org/citation.cfm?id=2832581.2832716`.

[14] R. Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[15] R. Caruana. Inductive transfer retrospective and review. In *NIPS 2005 Workshop on Inductive Transfer: 10 Years Later*, 2005.

[16] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1179`.

[17] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[18] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In *NIPS*, pages 1017–1023, 1995.

[19] S. Devlin and D. Kudenko. Theoretical considerations of potential-based reward shaping for multi-agent systems. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 225–232, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9826571-5-3, 978-0-9826571-5-7.

[20] S. Devlin and D. Kudenko. Dynamic potential-based reward shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 433–440. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[21] S. Devlin, M. Grzes, and D. Kudenko. Multi-agent, reward shaping for robocup keepaway. In *AAMAS*, pages 1227–1228, 2011.

[22] C. Diuk, L. Li, and B. R. Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *ICML*, page 32, 2009.

[23] M. Dorigo and M. Colombetti. Robot shaping: developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[24] K. Doya. Bifurcations in the learning of recurrent neural networks. In *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, volume 6, pages 2777–2780. IEEE, 1992.

[25] K. Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245,

2000.

[26] K. Doya and S. Yoshizawa. Adaptive neural oscillator using continuous-time back-propagation learning. *Neural Networks*, 2(5):375–385, 1989.

[27] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continous control. In *International Conference on Machine Learning (ICML-16)*, 2016.

[28] S. Elfwing, E. Uchibe, K. Doya, and H. Christensen. Co-evolution of shaping: Rewards and meta-parameters in reinforcement learning. *Adaptive Behavior*, 16(6):400–412, 2008.

[29] S. Elfwing, E. Uchibe, K. Doya, and H. I. Christensen. Darwinian embodied evolution of the learning ability for survival. *Adaptive Behavior*, 19(2):101–120, 2011.

[30] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[31] G. Endo, J. Morimoto, T. Matsubara, J. Nakanishi, and G. Cheng. Learning CPG-based biped locomotion with a policy gradient method: Application to a humanoid robot. *The International Journal of Robotics Research*, 27(2):213–228, 2008.

[32] T. Erez and W. Smart. What does shaping mean for computational reinforcement learning? In *Development and Learning, 2008. ICDL 2008. 7th IEEE International Conference on*, pages 215 –219, aug. 2008. doi: 10.1109/DEVLRN.2008.4640832.

[33] E. Even-Dar and Y. Mansour. Convergence of optimistic and incremental q-learning. In *NIPS*, pages 1499–1506, 2001.

[34] K. Ferguson and S. Mahadevan. Proto-transfer learning in markov decision processes using spectral methods. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[35] E. Ferrante, A. Lazaric, and M. Restelli. Transfer of task representation in reinforcement learning using policy-based proto-value functions. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, pages 1329–1332, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-2-3.

[36] D. J. Foster and P. Dayan. Structure in the space of value functions. *Machine Learning*, 49(2-3):325–346, 2002.

[37] L. Frommberger. Task space tile coding: In-task and cross-task generalization in reinforcement learning. In *Proceedings of the 9th European Workshop on Reinforcement Learning (EWRL9)*, 2011.

[38] L. Frommberger and D. Wolter. Structural knowledge transfer by spatial abstraction for reinforcement learning agents. *Adaptive Behavior*, 18(6):507–525, 2010.

[39] J. Fu, K. Luo, and S. Levine. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248*, 2017.

[40] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[41] P. W. Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987.

[42] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[43] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1764–1772, 2014.

[44] E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

[45] S. Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1(1):17 – 61, 1988. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/0893-6080(88)90021-4. URL `http://www.sciencedirect.com/science/article/pii/0893608088900214`.

[46] M. Grześ and D. Kudenko. Learning shaping rewards in model-based reinforcement learning. In *Proc. AAMAS 2009 Workshop on Adaptive Learning Agents*, 2009.

[47] M. Grzes and D. Kudenko. Theoretical and empirical analysis of reward shaping in reinforcement learning. In *ICMLA*, pages 337–344, 2009.

[48] M. Grześ and D. Kudenko. Online learning of shaping rewards in reinforcement learning. *Neural Networks*, 23(4):541 – 550, 2010. ISSN 0893-6080. doi: 10.1016/j.neunet.2010.01.001.

[49] V. Gullapalli and A. G. Barto. Shaping as a method for accelerating reinforcement learning. In *Proc. IEEE International Symposium on Intelligent Control*, pages 554–559, 1992.

[50] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[51] H. Hachiya and M. Sugiyama. Feature selection for reinforcement learning: Evaluating implicit state-reward dependency via conditional mutual information. In *ECML/PKDD*, pages 474–489, 2010.

[52] A. Harutyunyan, S. Devlin, P. Vrancx, and A. Nowé. Expressing arbitrary reward functions as potential-based advice. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015.

[53] M. Hausknecht and P. Stone. Deep recurrent Q-learning for partially observable MDPs. In *2015 AAAI Fall Symposium Series*, 2015.

[54] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[55] D. Huh and E. Todorov. Real-time motor control using recurrent neural networks. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL'09. IEEE Symposium on*, pages 42–49, 2009.

[56] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. In *Reproducibility in Machine Learning Workshop, ICML 2017*, 2017.

[57] H. Jaeger. The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13, 2001.

[58] N. K. Jong and P. Stone. State abstraction discovery from irrelevant state variables. In *IJCAI-05*, 2005.

[59] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350, 2015.

[60] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research (JAIR)*, 4:237–285, 1996.

[61] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[62] S. M. Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems*, pages 1531–1538, 2002.

[63] S. M. Kakade. *On the Sample Complexity of Reinforcement Learning*. PhD thesis, University College London, 2003.

[64] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, pages 1–33, 2010. ISSN 0885-6125. URL `http://dx.doi.org/10.1007/s10994-010-5223-6`. 10.1007/s10994-010-5223-6.

[65] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[66] D. Koller and M. Sahami. Toward optimal feature selection. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning (ICML)*, pages 284–292. Morgan Kaufmann Publishers, 1996.

[67] J. Z. Kolter and A. Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *ICML*, page 66, 2009.

[68] G. Konidaris and A. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proc. 23rd International Conference on Machine Learning*, pages 489–496, 2006.

[69] G. Konidaris, I. Scheidwasser, and A. G. Barto. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, 13:1333–1371, 2012.

[70] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proc. IEEE Conference on Robotics and Automation*, pages 1398–1404, 1991.

[71] M. Kroon and S. Whiteson. Automatic feature selection for model-based reinforcement learning in factored MDPs. In *ICMLA 2009: Proceedings of the Eighth International Conference on Machine Learning and Applications*, pages 324–330, December 2009.

[72] A. Laud and G. DeJong. Reinforcement learning and shaping: Encouraging intended behaviors. In *Proc. 19th International Conference on Machine Learning*, pages 355–362, 2002.

[73] A. Laud and G. DeJong. The influence of reward on the speed of reinforcement learning: An analysis of shaping. In *ICML*, pages 440–447, 2003.

[74] A. Lazaric. *Knowledge Transfer in Reinforcement Learning*. PhD thesis, Politecnico di Milano, 2008.

[75] A. Lazaric and M. Ghavamzadeh. Bayesian multi-task reinforcement learning. In *ICML*, pages 599–606, 2010.

[76] A. Lazaric, M. Restelli, and A. Bonarini. Transfer of samples in batch reinforcement learning. In *ICML*, pages 544–551, 2008.

[77] L. Li, T. J. Walsh, and M. L. Littman. Towards a unified theory of state abstraction for mdps. In *Aritificial Intelligence and Mathematics*, 2006.

[78] L.-J. Lin and T. M. Mitchell. Reinforcement learning with hidden states. In *Proceedings of the second international conference on From animals to animats 2: simulation of adaptive behavior: simulation of*

adaptive behavior, pages 271–280. MIT Press, 1993.
[79] X. Lu, H. M. Schwartz, and S. N. Givigi. Policy invariance under reward transformations for general-sum stochastic games. *Journal of Artificial Intelligence Research (JAIR)*, 41:397–406, 2011.
[80] R. Maclin and J. W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22 (1-3):251–281, 1996.
[81] S. Mahadevan. Representation discovery in sequential decision making. In *AAAI*, 2010.
[82] P. Mannion, K. Mason, S. Devlin, J. Duggan, and E. Howley. Multi-objective dynamic dispatch optimisation using multi-agent reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 1345–1346. International Foundation for Autonomous Agents and Multiagent Systems, 2016.
[83] P. Mannion, J. Duggan, and E. Howley. A theoretical and empirical analysis of reward transformations in multi-objective stochastic games. In *Proceedings of the 2017 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2017.
[84] P. Manoonpong, F. Wörgötter, and J. Morimoto. Extraction of reward-related feature space using correlation-based and reward-based learning methods. In *ICONIP (1)*, pages 414–421, 2010.
[85] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal of Applied Mathematics*, 11:431–441, 1963.
[86] B. Marthi. Automatic shaping and decomposition of reward functions. In *Proc. 24th International Conference on Machine Learning*, pages 601–608, 2007.
[87] N. Marwan, M. C. Romano, M. Thiel, and J. Kurths. Recurrence plots for the analysis of complex systems. *Physics Reports*, 438:237–329, 2007.
[88] M. J. Matarić. Reward functions for accelerated learning. In *Proc. 11th International Conference on Machine Learning*, 1994.
[89] N. Mehta, S. Natarajan, P. Tadepalli, and A. Fern. Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, 73(3):289–312, 2008.
[90] M. Midtgaard, L. Vinther, J. R. Christiansen, A. M. Christensen, and Y. Zeng. Time-based reward shaping in real-time strategy games. In *Proceedings of the 6th international conference on Agents and data mining interaction*, ADMI'10, pages 115–125, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15419-0, 978-3-642-15419-5.
[91] D. K. Misra, J. Langford, and Y. Artzi. Mapping instructions and visual observations to actions with reinforcement learning. *arXiv preprint arXiv:1704.08795*, 2017.
[92] T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.
[93] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *J. Artif. Intell. Res. (JAIR)*, 11:241–276, 1999.
[94] A. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proc. 16th International Conference on Machine Learning*, 1999.
[95] A. Y. Ng, S. J. Russell, et al. Algorithms for inverse reinforcement learning. In *Proceedings of the Internal Conference on Machine Learning (ICML)*, pages 663–670, 2000.
[96] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.
[97] R. Parr, L. Li, G. Taylor, C. Painter-Wakefield, and M. L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *ICML*, pages 752–759, 2008.
[98] B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212–1228, 1995.
[99] B. A. Pearlmutter. Personal communication, 2017.
[100] J. Peters, S. Vijayakumar, and S. Schaal. Natural actor-critic. In *European Conference on Machine Learning*, pages 280–291. Springer, 2005.
[101] M. Petrik, G. Taylor, R. Parr, and S. Zilberstein. Feature selection using regularization in approximate linear programs for markov decision processes. In *ICML*, pages 871–878, 2010.
[102] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York, NY, 1994.
[103] A. Rajeswaran, K. Lowrey, E. Todorov, and S. Kakade. Towards generalization and simplicity in continuous control. *arXiv preprint arXiv:1703.02660*, 2017.
[104] J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proc. 15th International Conference on Machine Learning*, 1998.
[105] A. J. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with ap-

plications to game playing. In *Proceedings of the 11th Conference of the Cognitive Science Society*, 1989.

[106] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[107] G. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG-RT 116, Engineering Department, Cambridge University, 1994.

[108] L. M. Saksida, S. M. Raymond, and D. S. Touretzky. Shaping robot behavior using principles from instrumental conditioning. *Robotics and Autonomous Systems*, 22(3-4):231 – 249, 1997. ISSN 0921-8890. doi: 10.1016/S0921-8890(97)00041-9. ¡ce:title¿Robot Learning: The New Wave¡/ce:title¿.

[109] J. Schmidhuber. An on-line algorithm for dynamic reinforcement learning and planning in reactive environments. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 253–258. IEEE, 1990.

[110] J. Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In *Advances in neural information processing systems*, pages 500–506, 1991.

[111] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[112] J. Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. PhD thesis, University of California, Berkeley, 2016.

[113] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 1889–1897, 2015.

[114] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[115] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[116] O. Selfridge, R. S. Sutton, and A. G. Barto. Training and tracking in robotics. In *Proc. Ninth International Joint Conference on Artificial Intelligence*, 1985.

[117] A. A. Sherstov and P. Stone. Improving action selection in MDP's via knowledge transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, July 2005.

[118] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[119] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[120] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016. ISSN 0028-0836. URL `http://dx.doi.org/10.1038/nature16961`.

[121] S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22 (1):123–158, 1996.

[122] S. Singh, R. Lewis, and A. Barto. Where do rewards come from? In *Proc. 31st Annual Conference of the Cognitive Science Society*, pages 2601–2606, 2009.

[123] S. P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3):323–339, 1992.

[124] S. P. Singh, T. Jaakkola, and M. I. Jordan. Learning without state-estimation in partially observable markovian decision processes. In *ICML*, pages 284–292, 1994.

[125] B. F. Skinner. *The behavior of organisms: An experimental analysis*. Appleton-Century-Crofts, New York, 1938.

[126] M. Snel and S. Whiteson. Multi-task evolutionary shaping without pre-specified representations. In *Genetic and Evolutionary Computation Conference (GECCO'10)*, 2010.

[127] M. Snel and S. Whiteson. Multi-task reinforcement learning: Shaping and feature selection. In *Proceedings of the European Workshop on Reinforcement Learning (EWRL)*, 2011.

[128] M. Snel, S. Whiteson, and Y. Kuniyoshi. Robust central pattern generators for embodied hierarchical reinforcement learning. In *Development and Learning (ICDL), 2011 IEEE International Conference on*, volume 2, pages 1–6. IEEE, 2011.

[129] J. Sorg and S. Singh. Transfer via soft homomorphisms. In *Proc. 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 741–748, 2009.

[130] A. L. Strehl, C. Diuk, and M. L. Littman. Efficient structure learning in factored-state mdps. In *AAAI*,

pages 645–650, 2007.

[131] H. B. Suay, T. Brys, M. E. Taylor, and S. Chernova. Learning from demonstration for shaping through inverse reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 429–437, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-1-4503-4239-1. URL `http://dl.acm.org/citation.cfm?id=2936924.2936988`.

[132] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[133] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[134] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1983.

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

[136] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning (ICML)*, pages 216–224, 1990.

[137] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, pages 1057–1063, 1999.

[138] F. Tanaka and M. Yamamura. Multitask reinforcement learning on the distribution of mdps. In *Proc. 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA 2003)*, pages 1108–113, 2003.

[139] J. Taylor, D. Precup, and P. Panagaden. Bounding performance loss in approximate mdp homomorphisms. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1649–1656, 2009.

[140] M. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[141] M. E. Taylor, P. Stone, and Y. Liu. Value functions for rl-based behavior transfer: A comparative study. In *AAAI*, pages 880–885, 2005.

[142] M. E. Taylor, S. Whiteson, and P. Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *AAMAS*, page 37, 2007.

[143] S. Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing*, pages 640–646, 1995.

[144] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[145] L. Torrey, T. Walker, J. W. Shavlik, and R. Maclin. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *Proceedings of the Sixteenth European Conference on Machine Learning (ECML 2005)*, pages 412–424, 2005.

[146] L. Torrey, J. W. Shavlik, T. Walker, and R. Maclin. Transfer learning via advice taking. In *Advances in Machine Learning I*, pages 147–170. Springer, 2010.

[147] H. van Seijen, S. Whiteson, and L. Kester. Switching between representations in reinforcement learning. In *Interactive Collaborative Information Systems*, pages 65–84. 2010.

[148] N. Vlassis, M. L. Littman, and D. Barber. On the computational complexity of stochastic controller optimization in pomdps. *CoRR*, abs/1107.3090, 2011.

[149] T. J. Walsh, L. Li, and M. L. Littman. Transferring state abstractions between mdps. In *ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[150] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[151] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[152] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[153] S. D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings AAAI-91*, pages 607–613, 1991.

[154] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, 2006.

[155] S. Whiteson, B. Tanner, M. E. Taylor, and P. Stone. Protecting against evaluation overfitting in empirical reinforcement learning. In *ADPRL 2011: Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 120–127, April 2011.

[156] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber. Solving deep memory POMDPs with recurrent

policy gradients. In *Proc. Intl. Conf. on Artificial Neural Networks (ICANN)*, pages 697–706. Springer, 2007.

[157] E. Wiewiora. Potential-based shaping and q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.

[158] E. Wiewiora, G. Cottrell, and C. Elkan. Principled methods for advising reinforcement learning agents. In *Proc. 20th International Conference on Machine Learning*, pages 792–799, 2003.

[159] A. Wilson, A. Fern, S. Ray, and P. Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *ICML*, pages 1015–1022, 2007.

Samenvatting

Een reinforcement-learning *agent* leert door proefondervindelijk door interactie met zijn omgeving, het observeren van het effect van zijn acties, en de beloning die het ontvangt na elke actie. Over het algemeen is het doel van de agent om reeks(en) van acties te identificeren die leiden tot de maximale totale som aan beloningen, ook wel de maximale *return* genoemd. Hoewel reinforcement learning conceptueel eenvoudig is, is de methodiek opmerkelijk krachtig voor het oplossen van sequentiële beslissingsproblemen. Het heeft talloze succesvolle praktische toepassingen en blijft een belangrijk onderzoeksgebied binnen de machine learning gemeenschap.

Formeel wordt de *taak* die de agent moet oplossen, gemodelleerd als een *Markov beslissingsproces* (MDP). MDP's kunnen *gedeeltelijk waarneembaar* zijn, wat betekent dat de huidige ware toestand van de omgeving slechts gedeeltelijk bekend is. Voor praktische doeleinden is het handig als de agent taak oplossingen kan generaliseren naar nieuwe, vergelijkbare taken die het in de toekomst kan tegenkomen. Dit proefschrift onderzoekt twee strategieën voor generalisatie in reinforcement learning.

De eerste strategie richt zich op een *multi-task reinforcement learning* scenario, waarbij taken worden gesampled uit een *domein*, een kansverdeling over gerelateerde taken. Agenten beginnen met het oplossen van een of meer taken, gesampled uit het domein. Omdat taken in het domein gerelateerd zijn, wordt van agenten verwacht dat ze kennis over opgeloste taken behouden en gebruiken in nieuwe taken, ook gesampled uit het domein, teneinde ze sneller op te lossen. In dit proefschrift maken agenten expliciet gebruik van gemeenschappelijke structuur van taken in het domein, door het gebruik van *shaping functies*. Shaping functies bieden de agent extra informatieve kunstmatige beloning, bovenop de beloning die de MDP biedt. Deze functies kunnen vooraf ontworpen, of geleerd zijn; dit proefschrift richt zich op het automatisch leren ervan. We doen dit door een dataset te vormen die bestaat uit de combinatie van state-action-value-paren van waargenomen taken. Net als in een *supervised learning* probleem moeten twee belangrijke keuzes gemaakt worden: de *target* functie die de shaping functie moet benaderen, en de representatie voor de shaping functie, d.w.z. de *feature set*. We stellen drie verschillende target functies voor, en evaluaren deze op een aantal synthetische domeinen. We laten empirisch zien dat de beste target functie afhangt van het domein, het leeralgoritme en de leerparameters.

Shaping functie *representaties* kunnen ook vooraf worden ontworpen of geleerd, en in dit werk worden ze voor het eerst automatisch geleerd. Om dit te doen introduceren we FS-TEK (Feature Selection Through Extrapolation of k-relevance), een nieuw *feature selection* algoritme. Het is gebaseerd op de nieuwe notie van k-relevantie, de verwachte relevantie van een feature set op een reeks van k taken gesampled uit het domein. We bewijzen dat de k-relevantie asymptotisch de domeinrelevantie van de feature set benaderd. Deze eigenschap wordt gebruikt om FS-TEK af te leiden. Het belangrijkste inzicht achter FS-TEK is dat verandering in relevantie die wordt waargenomen in taaksequenties van toenemende lengte kan worden geëxtrapoleerd om de relevantie van het domein nauwkeuriger te voorspellen. We demonstreren empirisch het voordeel van FS-TEK op een aantal synthetische domeinen.

De tweede strategie voor generalisatie in reinforcement learning onderzoekt neurale controllers die een zekere mate van robuustheid vertonen ten opzichte van veranderingen

in taak. Dat wil zeggen dat het doel is de verwachte prestatiedaling op nieuwe taken te minimaliseren met betrekking tot de taak waarop zij getraind zijn, zonder in de nieuwe taak te leren. We trainen vijf *recurrent neural net* (RNN) architecturen en een diep en ondiep feedforward net (FNN) op een set gesimuleerde robot locomotie taken en onderwerpen ze aan twee soorten verstoringen tijdens de test: sensor ruis en een overgang van vlak naar heuvelachtig terrein. De FNN's leren het snelst, maar geen enkele architectuur is in alle locomotie taken het beste op het moment van testen. We laten echter zien dat de FNN's en een *continuous time RNN* (CTRNN) gemiddeld het meest robuust zijn voor taakveranderingen, waarbij de CTRNN aanzienlijk beter presteert dan de anderen onder sensor ruis. Bovendien laten we zien dat training op een heuveltaak de verwachte prestatiedaling verlaagt als gevolg van verstoringen voor de RNN's, maar niet de FNN's.

Summary

A reinforcement-learning agent learns through trial and error by interacting with the environment and observing the effect of its actions and the reward that it receives after each action. Generally, the goal of the agent is to identify the sequence(s) of actions that lead to the maximal sum of rewards, or maximal *return*. While conceptually simple, the reinforcement learning framework is a remarkably powerful one for solving sequential decision tasks. It has had numerous successful practical applications, and remains a major area of research within the machine learning community.

Formally, the *task* the agent needs to solve is modeled as a *Markov decision process* (MDP). MDPs can be *partially observable*, meaning that the current true state of the environment is only partially known. For practical purposes, it is useful if the agent can generalize from tasks it has solved to new, but similar, tasks it might encounter in the future. This thesis investigates two classes of strategies for generalization in reinforcement learning.

The first strategy focuses on a multi-task reinforcement learning setting, in which tasks are sampled from a *domain*, a distribution over tasks. Agents start by solving one or more tasks, sampled from the domain. Since tasks in the domain are related, agents are then expected to retain knowledge about solved tasks and transfer it to new tasks, also sampled from the domain, in order to solve them more quickly. In this thesis, agents explicitly leverage structure that is shared between tasks, through the use of *shaping functions*. Shaping functions provide the agent with additional informative artificial reward, on top of the reward provided by the MDP. They can be either pre-designed or learned; this thesis focuses on learning them automatically. We do so by forming a dataset that consists of the union of state-action-value pairs of observed tasks. Akin to a supervised learning setting, two key choices need to be made: the target function the shaping function should approximate, and the representation for the shaping function, i.e., the feature set. We propose three different target functions to approximate, and evaluate each on a number of artificial domains. We show empirically that which target function is best depends highly on the domain, learning algorithm, and learning parameters.

Shaping function *representations* can also be pre-designed or learned, and this thesis is the first to learn them. In order to do so, we introduce FS-TEK (Feature Selection Through Extrapolation of k-relevance), a novel feature selection algorithm. It is based on the new notion of k-relevance, the expected relevance of a feature set on a sequence of k tasks sampled from the domain. We prove that k-relevance converges asymptotically to the domain relevance of the feature set. This property is used to derive FS-TEK. The key insight behind FS-TEK is that change in relevance observed on task sequences of increasing length can be extrapolated to more accurately predict domain relevance. We demonstrate empirically the benefit of FS-TEK on a number of artificial domains.

The second strategy for generalization in reinforcement learning investigates neural controllers that exhibit a degree of robustness to changes in task. That is to say, while these controllers do not learn on the new task(s), the objective is to minimize degradation in performance with respect to the task they were trained on. We train five recurrent neural net (RNN) architectures and a deep and shallow feedforward net (FNN) on a set of simulated locomotion tasks, and subject them to two types of perturbations at test time: sensor noise, and a switch from flat to hilly terrain. While the FNNs learn fastest,

no single architecture is best at everything at test time. However, we show that the FNNs and a continuous-time RNN (CTRNN) are most robust to task changes on average, with the CTRNN significantly outperforming the others under noise perturbation. In addition, we show that training on a hill task decreases the expected drop in performance due to perturbation for the RNNs, but not the FNNs.

Acknowledgements

With the completion of this thesis, a long journey to scientific maturity (well – teenagehood perhaps) has come to an end. Before thanking the people who have helped me along the way in more or less chronological order, I'd like to express my gratitude to Shimon Whiteson, my supervisor at the University of Amsterdam. Shimon is really the person who, with his critical "question everything" view, high standards, and attention to detail ("the devil is always in the details") has had a lasting impact on the way I think and approach problems. This work would not have been possible without his guidance.

The journey began, then, with an MSc in Artificial Intelligence at the University of Edinburgh, where I got fascinated and inspired by the Reinforcement Learning course taught by Gillian Hayes. Although I had no plans for starting a PhD when I entered my Master studies, I found RL, and AI in general, so fascinating that I decided to take the plunge. Thanks to Gillian for inspiring me to study RL more deeply, taking me on as a PhD student, and giving me the freedom to pursue my own naive ideas.

Unfortunately, Gillian left the university after about two years to start her own company. I thank Michael Herrmann for taking me over from Gillian and briefly supervising me before I left for the University of Amsterdam, where Shimon took me on as a student under somewhat irregular circumstances (for which also thanks).

In this period, I also traveled to the University of Tokyo several times to work as visiting student in the lab of Prof. Yasuo Kuniyoshi, initially under a JSPS Fellowship and later directly for the lab itself. Thanks to Kuniyoshi-sensei for arranging the fellowship after just one meeting, and enabling me to have a fantastic experience in Japan and in the lab. I have especially fond memories of the karaoke nights with the lab, with Prof. Kuniyoshi among the most enthusiastic singers. Thanks also to Daniel Burfoot, Hassan Alirezaei, Alexandre Pitti, and Fumihiro Bessho for providing advice on Japanese culture, and for research discussions and social outings. Thanks to Tatsuya Kaneko for friendship and good parties.

At the time my contract with the university ended, I had about half a chapter of research work left, and in astounding naivety (still!) decided that I would finish this while taking on a full-time job as researcher at Optiver, my current employer. Now, several years and a half-year leave of absence later, I did finally manage to finish that chapter. Here, heartfelt thanks are in order to Wim Vink, my boss at the time I requested leave. I'd also like to thank Frans Groen for valuable advice and being my promotor for all this time, even past his retirement, and Ben Kröse for stepping in as promotor during the last stretch.

This work would also not have been possible without all the great work done by researchers who came before me. I'm grateful I've been given the chance to contribute my insignificant pebble to the edifice of science.

Thanks to all my friends, who have put up with my endless "have to work on the PhD" excuses without complaint (mostly). Sorry guys. Special thanks to Wing, for her general awesomeness and for designing the cover of this booklet.

Last but not least, my deepest thanks to my family, who have somehow managed to guide me while giving me total freedom, for always supporting me in whatever I do.