Poznan University of Technology
Institute of Computing Science

COEVOLUTIONARY SHAPING FOR REINFORCEMENT LEARNING

MARCIN G. SZUBERT

A dissertation submitted to
the Council of the Faculty of Computing
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Supervisor: Krzysztof Krawiec, Ph. D. Dr. Habil.
Co-supervisor: Wojciech Jaśkowski, Ph. D.

Poznań, Poland
2014

This dissertation is dedicated to my beloved wife, Michalina,
for her patience, support, and continuous encouragement.

ACKNOWLEDGMENTS

The work described here was carried out between October 2009 and May 2014 in the Laboratory of Intelligent Decision Support Systems at the Faculty of Computing at Poznan University of Technology. I would like to express my gratitude to all the people who have contributed to the completion of this dissertation.

First and foremost, this work would not have been possible without the enormous effort of my supervisor, Krzysztof Krawiec. I deeply thank him for his constant inspiration, encouragement and, most importantly, useful criticism. I am also very grateful for the advice and support of my second supervisor Wojciech Jaśkowski, who has shown a large interest in my work. His expertise and insightful ideas have greatly improved this thesis.

Furthermore, I would like to immensely thank my family for their understanding and tireless support. Their ongoing encouragement has kept me going throughout this work.

This research has been supported by the Polish National Science Centre grant no. DEC-2012/05/N/ST6/03152.

ABSTRACT

Shaping is an important animal training technique that originates from behavioral psychology. The main motivation behind this technique is to enable animals to perform tasks that are too difficult to be learned directly. Shaping typically consists in starting from related simpler tasks and progressively increasing their difficulty. As a result, the learner can be exposed to appropriate training experience and gradually refine its skills. By providing a pedagogical sequence of training tasks, shaping is expected to guide the learner towards the behavior of ultimate interest.

This thesis investigates the concept of shaping in reinforcement learning — a machine learning paradigm closely related to human and animal learning. In this paradigm, an agent learns a decision-making policy for a sequential decision task through repeated trial-and-error interactions with an environment. Although shaping has been already applied to improve the effectiveness of reinforcement learning, most of the existing approaches rely on manually designed training environments and thus require a substantial amount of domain knowledge and human intervention.

In this thesis we propose a unified shaping framework and introduce novel shaping approaches that avoid incorporating domain knowledge into the learning process. To this end, we rely mainly on competitive coevolutionary algorithms, which autonomously realize shaping by coevolving learners against their training environments. We investigate a hybrid of coevolution with self-play temporal difference learning and analyze this combination in the context of its generalization performance and scalability with respect to the search space size. Next, we design a novel measure of task difficulty and use it to devise a set of shaping methods that provide training tasks from a precomputed task pool according to either static or dynamic difficulty distribution. Finally, we formalize the problem of optimal shaping and design a coevolutionary method that optimizes training experience for a temporal difference learning algorithm.

The proposed shaping methods are experimentally verified in non-trivial sequential decision making domains, including the benchmark problem of cart pole balancing and the board games of Othello and small-board Go. We demonstrate that shaping can provide significant empirical benefits compared to conventional unshaped reinforcement learning, either by improving the final performance or by facilitating faster convergence.

PREFACE

Some ideas, figures and portions of text presented in this dissertation have appeared previously in the following publications:

[1] Krzysztof Krawiec and Marcin G. Szubert. Coevolutionary Temporal Difference Learning for Small-Board Go. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010*, pages 1–8, Barcelona, Spain, 2010. IEEE.

[2] Krzysztof Krawiec, Wojciech Jaśkowski, and Marcin G. Szubert. Evolving Small-board Go Players Using Coevolutionary Temporal Difference Learning with Archives. *International Journal of Applied Mathematics and Computer Science*, 21(4):717–731, 2011.

[3] Krzysztof Krawiec and Marcin G. Szubert. Learning N-tuple Networks for Othello by Coevolutionary Gradient Search. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 355–362, New York, NY, USA, 2011. ACM.

[4] Marcin G. Szubert and Krzysztof Krawiec. Autonomous Shaping via Coevolutionary Selection of Training Experience. In *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature - Volume Part II*, PPSN'12, pages 215–224, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Marcin G. Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. On Scalability, Generalization, and Hybridization of Coevolutionary Learning: A Case Study for Othello. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3):214–226, 2013.

[6] Marcin G. Szubert, Wojciech Jaśkowski, Paweł Liskowski, and Krzysztof Krawiec. Shaping Fitness Function for Evolutionary Learning of Game Strategies. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 1149–1156, New York, NY, USA, 2013. ACM.

CONTENTS

1 INTRODUCTION

1.1 PROBLEM SETTING AND MOTIVATION

Sequential decision problems

Many real-world problems concern sequential decision making in which a decision-making agent must perform a sequence of actions in an unknown environment. Since actions change the state of the environment, the agent must act dynamically to achieve its goals. Additionally, actions typically result in both immediate and delayed consequences that can be quantified as rewards for the agent. The goal of the agent is to select such actions that maximize the cumulative reward in a long-term perspective. Examples of such sequential decision problems include playing board games, driving a car, and dynamic task scheduling.

The machine learning approach

One way to develop intelligent agents capable of sequential decision making is to use machine learning. In such an approach, an agent is expected to learn automatically through the use of training experience and so improve its performance at a given task [35, 134]. To that aim, the agent employs a learning algorithm which processes the provided training experience and builds a specific knowledge representation (e.g., a neural network or a set of decision rules). Depending on the type and source of training experience, two machine learning paradigms can be used for sequential decision problems.

Supervised learning

In case of supervised learning paradigm, the experience is supplied by an external teacher (usually a human expert) in the form of labeled training examples. Most commonly, training experience would contain correct actions to be taken in selected states of the environment. An approach in which training examples are recorded during demonstrations performed by skilled human operator (e.g. a car driver [153] or an aircraft pilot [167]) has been termed learning from demonstrations [7] or behavioral cloning [11]. Importantly, it is the role of the teacher to select representative and informative training examples. This importance was emphasized already in the early work of Selfridge, Sutton and Barto:

> *The importance of good training experience is well recognised in pattern classification and inductive inference, where careful choice of rule exemplars and counter-exemplars clearly affects learning progress.* (Selfridge et al. [174], p. 670)

However, for many nontrivial problems, supervised learning is difficult to apply due to the lack of expert knowledge in a problem domain or a large cost of gathering such expertise.

Reinforcement learning

An alternative machine learning paradigm for sequential decision problems is reinforcement learning [189, 226]. In this paradigm, by contrast, there is no teacher meant as a provider of training experience. Instead, the experience is collected autonomously by the agent during interactions with the environment, which is given as a part of problem statement. The agent is not told how to respond to a given situation, but instead it must find out itself which actions yield the most reward by trying them. Through such trial and error search, with reward being the only training signal, the agent gathers experience about possible system states, actions, transitions and rewards. Essentially, very rarely the agent receives exact information about its performance directly after each action — usually such information is delayed, and thus, indirect. Since actions may affect all subsequent rewards, a problem arises known as temporal credit assignment [133] — the agent must determine which actions are to be credited with the eventual rewards.

The brief comparison of the two learning paradigms indicates that reinforcement learning, though typically requiring much less human intervention, is generally more challenging than supervised learning where a direct training signal is provided. Consequently, tackling complex tasks with reinforcement learning methods may be slow or infeasible. This is particularly notable when the agent starts learning from scratch, as a *tabula rasa*, without any (or with very little) knowledge about the environment. In their survey on reinforcement learning, Kaelbling, Littman and Moore concluded:

Tabula rasa attitude

> *There are a variety of reinforcement-learning techniques that work effectively on a variety of small problems. But very few of these techniques scale well to larger problems. This is not because researchers have done a bad job of inventing learning techniques, but because it is very difficult to solve arbitrary problems in the general case. In order to solve highly complex problems, we must give up tabula rasa learning techniques and begin to incorporate bias that will give leverage to the learning process.* (Kaelbling et al. [101], p. 274)

For this reason, a lot of research has been conducted towards improving the efficiency of reinforcement learning by incorporating some sort of domain knowledge [43, 45, 122, 190].

The idea of shaping

A possible means of aiding reinforcement learning is the concept of shaping, borrowed from behavioral psychology and originally applied by Skinner [180] to train animals. The main principle of shaping is to construct successive approximations of the original task that is too complex to learn directly, and use such approximations for learning. By starting from simpler tasks and progressively increasing their difficulty, the agent can gather useful training experience and gradually refine its skills. Shaping is supposed to provide such training tasks that guide the agent towards the behavior of ultimate interest.

The idea of shaping has been successfully applied to facilitate reinforcement learning of complex tasks. Although there are many computational renderings of shaping, most of them rely on training tasks that are derived from the target task by modifying some aspect of the problem [54]. For instance, training tasks may differ with respect to physical dynamics, the reward scheme, or the number of possible actions. One of the most appealing empirical results of shaping concerns learning to drive a bicycle by using additional training wheels and thus changing the physics of the problem [159]. An alternative shaping approach for the same task consists in providing agent with additional rewards for heading towards the goal state [160]. These studies convincingly justify the use of shaping as a powerful technique for accelerating reinforcement learning of nontrivial problems.

Promising shaping results

However, facilitating learning via shaping comes at a price. It typically requires giving up the *tabula rasa* view and employing a knowledgeable teacher responsible for providing training tasks. Although the training experience is still gathered autonomously by the agent, it is largely influenced by the choice of training tasks. For this reason, shaping can be regarded as a supervised variant of reinforcement learning [47, 54]. Besides involving substantial amount of domain knowledge, handcrafting training tasks that approximate desired behavior can also introduce unnecessary biases into the learning process. In this context, learning from scratch remains an attractive feature of basic reinforcement learning, conforming to the idea of autonomous intelligence — the primary aspiration of machine learning.

The price of shaping

1.2 AIMS AND SCOPE

Following the above discussion, in this thesis we focus on increasing the performance of solving sequential decision problems by the use of shaping techniques. Since there are many meanings of shaping in reinforcement learning [54], here we will apply the term 'shaping' to any method that affects the training environment, but at the same time leaves the learning algorithm unchanged. Therefore, instead of tuning the parameters of the algorithms, we put the emphasis on exposing the learner to the right training experience. In short, *what* to learn becomes here more important than *how* to learn.

The scope of shaping

Moreover, we attempt to avoid incorporating human knowledge into the shaping process and thus maintain the *tabula rasa* attitude. In particular, our purpose is to come up with useful training tasks without human supervision, in a knowledge-free way [124]. To this end, we employ competitive coevolutionary algorithms [154], which are believed to autonomously sustain a tractable learning gradient by coevolving learners and their learning environments. We expect that training experience provided by these algorithms will lead to both faster learning convergence and improved final performance.

Coevolutionary shaping

The overall goal of this thesis is thus to *propose and analyze coevolution-based methods aimed at improving the efficiency of reinforcement learning by implementing the general idea of shaping*. The specific objectives include:

Specific objectives

- To develop a unified shaping framework and identify the possible ways of knowledge-free shaping for reinforcement learning.
- To investigate competitive coevolution and self-play temporal difference learning, the two implicit variants of shaping widely applied for learning game-playing policies.
- To analyze the hybridization of evolutionary search and gradient-based learning in the context of scalability and generalization.
- To design a measure of task difficulty and devise difficulty-based shaping methods that provide training tasks according to a predefined or dynamically maintained difficulty distribution.
- To formalize the problem of optimal shaping and design a coevolutionary method which attempts to optimize the training experience for a temporal difference learning algorithm.
- To experimentally verify the proposed shaping methods on selected sequential decision problems and compare them to the reference unshaped approaches.

1.3 THESIS OUTLINE

This dissertation proposes several methods of shaping in reinforcement learning and describes them in separate chapters. Particular chapters present also the results of computational experiments that were conducted to validate the proposed methods and compare them to existing unshaped approaches. This implies certain organization of the text, where the description of the experimental domains precedes the presentation of particular methods. More specifically, the dissertation is organized as follows.

Chapter 2 provides a brief introduction into the field of reinforcement learning. We describe the conventional reinforcement learning framework based on the formalism of Markov Decision Processes and introduce the two approaches for sequential decision problems: *temporal difference learning* and *direct policy search* represented by evolutionary algorithms.

In **Chapter 3** we present shaping techniques applied in human and animal learning. Afterwards, we provide a brief literature review of existing shaping-related approaches in reinforcement learning.

Chapter 4 introduces a unified shaping framework which delineates the role of shaping in reinforcement learning. On this basis we demonstrate how coevolutionary algorithms fit into our understanding of shaping.

Chapter 5 presents three experimental domains used throughout this thesis to validate the proposed shaping methods: board games of Othello and small-board Go, and the control problem of cart pole balancing. For each domain we discuss the possible decision-making policy representations and the performance measures employed to evaluate the learning results.

Chapter 6 demonstrates the application of two popular reinforcement learning methods that can be regarded as forms of shaping. In particular, we employ single-population coevolution and self-play temporal difference learning to develop game-playing policies. Additionally, we present coevolutionary temporal difference learning, a hybrid method that combines elements of gradient-descent learning and population-based search. The considered methods are compared in terms of their scalability and generalization performance.

In **Chapter 7** we introduce the measure of task difficulty and the notion of difficulty distribution in the context of multi-task reinforcement learning domains. On this basis, we propose a set of shaping methods that provide training tasks according to either static or adaptively changing difficulty distribution. The most autonomous of the proposed methods relies on the two-population coevolution.

Chapter 8 formalizes the problem of designing an optimal shaping task sequence. To synthesize a useful task sequence, we suggest a coevolutionary algorithm that attempts to select the training experience on which temporal difference learning can successfully operate.

Chapter 9 summarizes the dissertation, reviews the main contributions and outlines the promising directions for future work.

2 REINFORCEMENT LEARNING

This chapter provides a brief introduction into the field of reinforcement learning (RL). Since RL "is defined not by characterizing learning methods, but by characterizing a learning problem" (Sutton and Barto [189], p. 4), we start by describing the reinforcement learning problem which is then formalized using the mathematical framework of Markov Decision Processes (MDPs) in Section 2.1. Next, we introduce the notion of value functions and define the optimal decision-making policy which constitutes a solution to an MDP. We also describe the dynamic programming methods, which in principle could be used to solve an MDP if its complete model is known in advance.

In Section 2.2 we introduce two distinct model-free approaches to solving reinforcement learning problems, namely, searching in the value function space and searching directly in the policy space. In this thesis we implement these two approaches in, respectively, temporal difference learning methods (Section 2.2.2) and evolutionary algorithms (Section 2.2.3). The reader interested in a more comprehensive treatment of reinforcement learning is referred to the works of Kaelbling et al. [101], Sutton and Barto [189], Moriarty et al. [138] and the recent book of Wiering and van Otterlo [226].

2.1 REINFORCEMENT LEARNING PROBLEM

The reinforcement learning problem can be regarded as a microcosm of artificial intelligence: an *agent* is placed in an *environment* and must learn how to act rationally by *interacting* with this environment [165, 187]. The agent (also called the *learner*) is an intelligent decision-making entity, which is able to observe the *state* of the environment. Depending on these observations, it makes a decision and takes an *action*. As a result, the state of the environment changes while the agent can receive a numerical *reward* for its actions. A sequence of such interactions between the agent and the environment (illustrated in Fig. 2.1) embodies a sequential decision making process. Importantly, through the interactions the agent gathers the experience which can be used to learn how to behave in the environment. The ultimate goal of an RL problem is to develop a decision-making *policy* which maximizes the expected sum of rewards.

Learning from interactions

Reinforcement learning, in contrast to supervised learning, does not rely on human supervision or any examples of correct behavior. The learner is not told how to respond to observed state, but instead it must discover which actions (and in which states) are the most

Trial-and-error learning

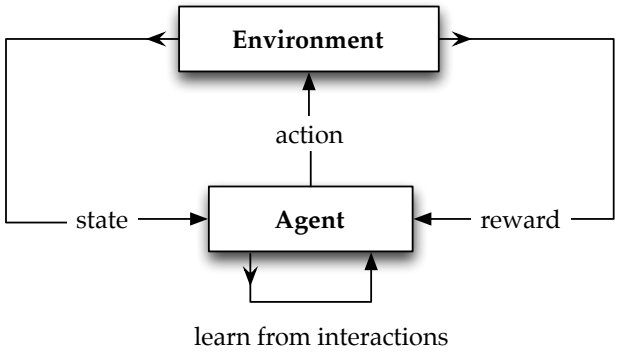


Figure 2.1: A general scheme of agent-environment interactions.

rewarding by experiencing them. Through such *trial-and-error* exploration of the environment, the agent gathers the training experience about the possible state transitions and rewards.

The main difficulty arises from the fact that rewards can be delayed in time. As a result, acting greedily is not always the best strategy and it is hard to determine which actions should be credited with the future rewards. This problem is known as *temporal credit assignment*:

Temporal credit assignment

> *In playing a complex game such as chess or checkers... one has a definite success criterion — the game is won or lost. But in the course of play, each ultimate success (or failure) is associated with a vast number of internal decisions. If the run is successful, how can we assign credit for the success among the multitude of decisions?* (Minsky [133], p. 20)

Exploration vs. exploitation

Another characteristic feature of an RL problem is the trade-off between *exploration* and *exploitation*. On the one hand, to receive higher reward, the agent should favor exploiting states and actions that it has tried in the past and already learned that they yield high rewards. On the other hand, the only way to discover highly-rewarded state-action combinations is to explore new states and actions. Thus, to learn successfully, the agent must maintain a proper balance between these two experimentation strategies.

Nondeterminism

RL in general does not assume the environment to be deterministic. The same action taken in the same state can result in different transitions and rewards. This makes it even harder to credit the actions with rewards and so renders RL problem challenging.

Real-world applications

Due to flexibility of the RL problem statement, applications of reinforcement learning are numerous[1]. In particular there are a few notable successes of RL in learning board game strategies, including the early work of Samuel [169] on checkers, and the famous backgammon program called TD-gammon implemented by Tesauro [198]. Other examples of interesting RL applications include job shop scheduling [233], helicopter flying [143] or controlling elevators [37].

1 http://www.ualberta.ca/~szepesva/RESEARCH/RLApplications.html

2.1.1 *Markov Decision Processes*

Reinforcement learning problems are conventionally modeled using the mathematical framework of Markov Decision Processes (MDP) [157]. An MDP is a discrete-time stochastic control process, defined as a 6-tuple $\langle S, A, T, R, I, \gamma \rangle$, in which: *MDP definition*

- S is a set of possible states of the environment and $s_t \in S$ is a state observed at time step t.
- A is a set of actions, where $A(s) \subseteq A$ denotes a set of actions available in state $s \in S$. The action taken by the agent at time step t is denoted as $a_t \in A(s_t)$.
- $T : S \times A \times S \to [0, 1]$ is a *transition function*, where $T(s, a, s') = \Pr(s' \mid s, a)$ denotes the probability of transition to state s' in result of taking action a in state s. If the MDP is deterministic, the transition function can be simplified to $T : S \times A \to S$.
- $R : S \times A \times S \to \mathbb{R}$ is a *reward function*, where $R(s, a, s')$ denotes the expected reward for taking action a in state s and causing a transition to state s'. The actual reward received by the agent after such transition is denoted as r_{t+1} and satisfies $\mathbb{E}\left[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right] = R(s, a, s')$. If the MPD is deterministic, reward function can be simplified to $R : S \times A \to \mathbb{R}$, and consequently $r_{t+1} = R(s_t, a_t)$.
- $I : S \to [0, 1]$ is an *initial state distribution* from which the initial states $s_0 \in S$ are drawn when the process is initialized.
- $\gamma \in [0, 1]$ is a *discount factor* which determines the *value* of future rewards — a reward received k time steps ahead is worth γ^{k-1} times as much as the same reward received immediately.

State transitions of an MDP, by definition, exhibit the so called *Markov property* — the conditional probability distribution of future states depends only on the current state of the process, and thus, is independent of the history. This can be expressed formally: *Markov property*

$$\begin{aligned}\Pr(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) \\ = \Pr(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t) \quad (2.1)\end{aligned}$$

The objective of an agent situated in an environment defined as an MDP is to maximize some function of the reward sequence, e.g., the expected cumulative discounted reward $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right]$. Clearly, the obtained rewards depend on the actions taken by the agent. Thus, to achieve the objective, the agent learns a decision-making *policy* $\pi : S \to A$, which specifies what action should be taken in the currently observed environmental state. *Policies*

Policy return

The *policy return* $J : \pi \to \mathbb{R}$ is the expected cumulative reward obtained by following a given policy starting from an initial state drawn from distribution I:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 \sim I \right], \tag{2.2}$$

Optimal policy

where $\mathbb{E}_\pi$ denotes the expected value when the agent makes actions according to π at any time step t, i.e., $a_t = \pi(s_t)$. The ultimate goal is to find an *optimal policy* π^* from the given policy space Π. Thus, π^* constitutes a solution to the MDP. Naturally, the optimal policy is the one that leads to gathering the maximal return:

$$\pi^* = \arg\max_{\pi \in \Pi} J(\pi). \tag{2.3}$$

Terminology

An MDP completely specifies the environment and together with the objective function it defines a sequential decision task to be solved — an instance of the reinforcement learning problem. Since, for the purpose of this thesis, we assume a fixed objective function (the expected cumulative discounted reward), the terms 'MDP', 'environment' and 'task' can be used interchangeably.

2.1.2 *Value Functions*

State value functions

According to the *value function hypothesis* formulated by Sutton [188], to find the optimal policy efficiently, *value function* needs to be computed as an intermediate step. A value function estimates policy's expected total future reward given the current state or state-action pair. The *state value function* $V^\pi : S \to \mathbb{R}$ describes the expected cumulative reward when the agent starts from state s at time step t and follows policy π to take actions:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right]. \tag{2.4}$$

Bellman equations

Importantly, the value $V^\pi(s_t)$ of the state observed at time step t can be divided into the reward r_{t+1} received immediately and the discounted sum of all the following rewards. The latter sum can be expressed as the value of the subsequent state $V^\pi(s_{t+1})$. The resulting recursive dependency, known as *Bellman Equation*, is crucial for dynamic programming methods (see Section 2.1.3):

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s \right] \\ &= \sum_{s' \in S} T(s, \pi(s), s') \left(R(s, \pi(s), s') + \gamma V^\pi(s') \right). \end{aligned} \tag{2.5}$$

In some cases, for instance when the transition function T is unknown, it may be more useful to calculate the *action value functions*.

For each policy, there exists an action value function $Q^\pi : S \times A \to \mathbb{R}$, which specifies the expected cumulative reward when following policy π and starting with taking action a in state s:

Action value functions

$$Q^\pi(s,a) \quad = \quad E_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right]. \tag{2.6}$$

By definition, following the optimal policy allows the agent to gather at least as much reward as it would receive by using any other policy, i.e., $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and all $\pi \in \Pi$. It can be shown that there exists at least one optimal policy for an MDP, but even if there is more of them, all optimal policies share the same *optimal value function* $V^* = V^{\pi^*}$. Moreover, the optimal value function is sufficient to act optimally:

Optimal value function

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V^*(s')). \tag{2.7}$$

This fact is exploited by many algorithms, which instead of directly searching for policies, aim for the optimal value function.

2.1.3 *Dynamic Programming*

If the complete model of the environment is available (i.e., its transition and reward functions), the optimal policy can be found with a *model-based* approach, which is implemented by *dynamic programming* (DP) methods. DP methods, proposed by Bellman [15], employ the aforementioned recursive dependency between the values of successive states (cf. Equation 2.5), to articulate *Bellman optimality equation*:

Bellman optimality equation

$$V^*(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V^*(s')). \tag{2.8}$$

The equation expresses the intuitive fact, that the optimal value of state s is equal to the expected cumulative discounted reward obtained after taking the best action available in that state. If the number of states is finite, then the system of such equations, one per each state $s \in S$, could be in principle solved explicitly by some method for solving systems of non-linear equations.

The two most popular DP methods, *value iteration* and *policy iteration* [89] approach the problem of computing the optimal value function by turning Bellman equations into iteratively applied update rules. The rules improve the estimate of the value of a given state on the basis of estimates of values of its successor states. This idea is known as *bootstrapping* [189] and is a characteristic feature not only of DP methods but also of temporal difference learning algorithms (cf. Section 2.2.2). Both policy iteration and value iteration start from arbitrary policies represented by value functions and are guaranteed to converge in the limit towards V^* [16].

Value iteration and policy iteration

Dynamic programming applicability

Although the DP methods are able to exactly solve an MDP, in practice they can be difficult to apply. The main problem is that they require the model of the MDP to be known and an exact (tabular) representation of the value function, with one entry per each state or state-action pair. In practice, even if we precisely know the environment's dynamics, the number of states can be so large that storing their values explicitly is technically infeasible. Such a problem exists for example in most non-trivial board games, where the environment's model is provided by game rules, but the huge number of possible game states makes application of DP impossible. The problem of large state spaces has been already identified by Bellman [15], who coined it the *curse of dimensionality*, by which he meant that the size of the state space grows exponentially with the number of state variables. Moreover, many problems are characterized by continuous state spaces, which also precludes the use of DP.

2.2 REINFORCEMENT LEARNING METHODS

Model-free approach to MDP

Reinforcement learning methods attempt to find a solution for an MDP but, in contrast to dynamic programming (cf. Section 2.1.3), do not take advantage of the environment's model given *a priori*, and thus are generally considered as *model-free* [226]. Instead of exploiting the knowledge about environment's dynamics, they essentially *learn* a policy from the samples of experience generated in simulated episodes of interactions between the agent and the environment. A single interaction consists of observing the current state of the environment s_t, choosing an action a_t, and receiving reward r_{t+1}. At the same time, the environmental state transitions to s_{t+1}. The quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$ can be considered as an elementary unit of *training experience* gathered in such an interaction.

Training experience

Online learning

The general scheme of model-free reinforcement learning is illustrated in Figure 2.2. Starting with some arbitrary policy, the agent is placed in the environment and takes actions accordingly. By observing state transitions and received rewards, it gathers training experience, which allows the learning algorithm to reason about the environment and adjust the *target policy* π developed so far. The phases of experience gathering and learning from it are typically alternated many times. In particular, the scenario in which learning occurs after every single interaction is called *online.*

Off-policy and on-policy learning

Depending on the *behavior policy* π_b employed to generate training experience, learning can be regarded as *on-policy* or *off-policy* [156, 189]. On-policy learning employs the target policy as the behavior policy, to take the actions in the environment, i.e. $\pi_b = \pi$. In the off-policy case, by contrast, the target policy is learned from the experience generated by following another behavior policy, typically a randomized policy derived from π.

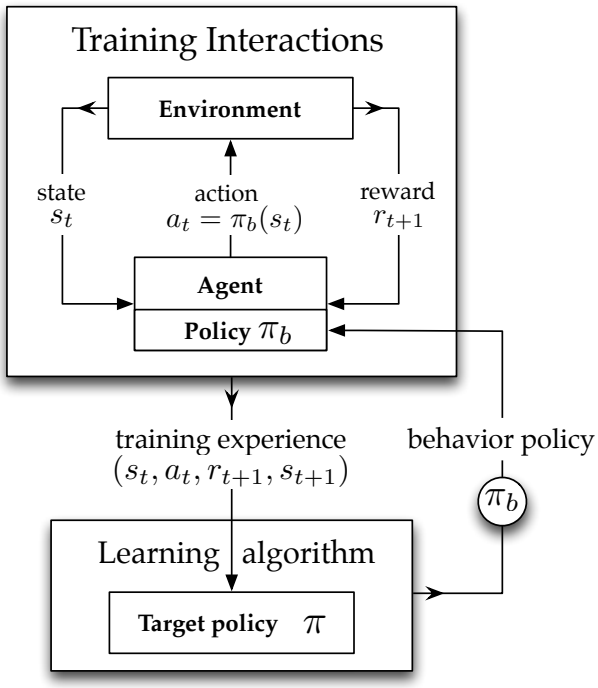


Figure 2.2: A general scheme of model-free reinforcement learning.

Exploration

In order to learn how to behave in an unknown environment, the crucial issue is to *explore* it during training interactions. If the agent always chooses the action specified by the target policy, it may repeat the same behavior and, in deterministic case, observe the same state transitions. Therefore, it is sometimes useful to follow another behavior policy in the hope of discovering more rewarding actions. The simplest way to ensure environment exploration is so called ϵ-*greedy* strategy [214]. This exploration strategy works by taking, with probability ϵ, a randomly chosen action instead of that specified by the target policy. Since using the right exploration strategy can lead to generating more informative samples of training experience, a lot of research has been devoted to efficient exploration in reinforcement learning [136, 202, 224].

Value function based methods

There are two types of model-free methods: those that rely on value functions and those that search the space of policies directly. In the former case the learning algorithm maintains policy implicitly in the form of a value function and updates the values of particular states (or state-action pairs) according to the training experience. The premise behind this approach is that any value function can be easily turned into policy by acting greedily and choosing actions leading to the most valuable successor states. Therefore, as already mentioned in Section 2.1.2, finding the optimal value function is equivalent to finding an optimal policy. Most of such methods are based on temporal difference learning described in Section 2.2.2.

Direct policy search

The second type of model-free methods, the direct policy search methods, represent policies explicitly and attempt to find the optimal one through a variety of search operators [138]. The objective function that steers the optimization process is typically calculated as the average policy return obtained in a series of training episodes in the given environment. Consequently, out of all training experience gathered by the agent, these methods utilize only the cumulative reward. These methods are represented by, among others, genetic algorithms — "although not often thought of in this way, genetic algorithms are, in a sense, inherently a reinforcement learning technique" (Whitley et al. [221]). Furthermore, since any method capable of solving RL problems can be considered as RL method, general purpose optimization techniques like simulated annealing or evolutionary computation may be treated as such too. In this thesis, among direct policy search methods, we are particularly concerned about evolutionary algorithms (see Section 2.2.3), which have become a widely used approach to reinforcement learning problems [73, 137, 217, 221].

2.2.1 *Function Approximation*

Policy as an action selector

Before we explore the particular learning algorithms we need to introduce the idea of *function approximation* [28], which allows to store policies and their value functions in case of high-dimensional or continuous state spaces. In practice, a policy is often represented as an *action selector* [196, 218], which realizes the same mapping as an action-value function. For this reason, we will limit the following discussion to the issue of efficiently representing value functions.

Look-up tables

In tasks with small and discrete state spaces, value function can be easily represented as a *look-up table* [201], where each value is stored individually. However, if the number of states grows, using an explicit value table becomes infeasible not only due to memory requirements but also because of the number of interactions required to visit all states and estimate their individual values accurately.

Generalization

The solution to these problems is to use a *function approximator* that adopts a class of parameterized functions to replace the look-up table. Employing function approximation allows to represent a value function in a much more compact way, because the number of parameters needed to specify the approximator is usually far less than the number of states. Additionally, approximation allows to *generalize* limited training experience across large state spaces, so updating the value of one state affects the values of many other states with related characteristics [100]. As a result there is no longer a need to explore every state in order to estimate its value, since a function generalizes from observed states to all other states, even those that were never experienced during interactions with the environment.

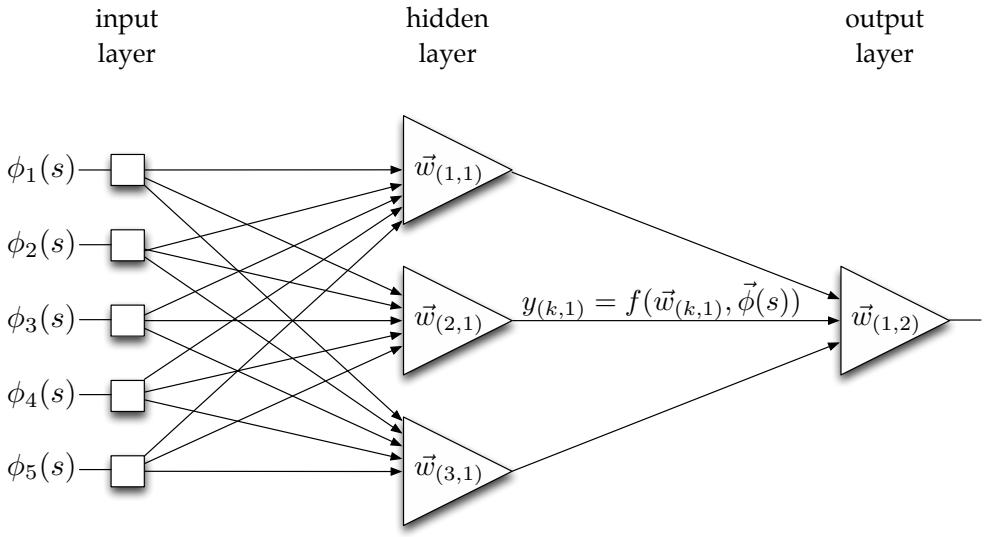


Figure 2.3: An illustrative multilayer perceptron with five inputs, one hidden layer consisting of three neurons and a single neuron in the output layer.

2.2.1.1 *Artificial Neural Networks*

The most common type of function approximators are *artificial neural networks* (ANNs), described thoroughly by Haykin [82]. Neural networks are bio-inspired general-purpose computational models for representing functions in a compositional manner. Typically, they are composed of many simple processing elements called *neurons*, which are interconnected and communicate with each other by sending signals. Remarkably, they are capable to uniformly approximate any differentiable function [38].

Multilayer perceptron

Among many architectures of ANNs, the particularly popular one is a feedforward layered network known as a *multilayer perceptron* (MLP). Figure 2.3 illustrates an example of three-layer[2] MLP. In this type of network, neurons are divided into a sequence of layers, where the neighboring layers are fully connected to each other. The first layer, called *input layer*, is responsible for preparing (e.g., normalizing) the input signals and propagating them to *hidden layers*. The signals propagate then through the hidden layers, and end up at the outputs of the neurons in the output layer, from where they can be fetched and interpreted in an application-specific manner. Each neuron implements a nonlinear activation function f and a modifiable vector of parameters (weights) $\vec{w}$, which both determine how the neuron aggregates all its inputs into a single output y. Typically, the aggregation involves a weighted sum of inputs passed through a sigmoid or a hyperbolic tangent function.

2 Some authors do not count the input layer while the others count only the number of hidden layers.

2.2.1.2 *Neural Networks for Value Function Approximation*

Neural networks such as MLPs are frequently employed in RL applications for representing both state value function $V(s)$ and action value function $Q(s,a)$. In such cases, the number of network inputs and outputs is task-specific while the number of hidden neurons is left to network designer.

State value approximation

When using ANNs to approximate a state-value function V, the inputs of the network are determined by a vector of features $\vec{\phi}(s)$ derived from the observed state s, and the single output of the network is supposed to approximate $V(s)$. Such function can be employed for selecting actions if the environment is deterministic and its transition model is known at least partially. For instance, in many board games it is not difficult to compute all possible board positions (called *afterstates* [189]) resulting from legal moves, although the opponent reply may not be known. By applying the ANN to estimate the value of every afterstate, the agent could take the move leading to the most valuable one. In this context the function approximated by the ANN can be considered as *afterstate value function*.

Action value approximation

To approximate the action value function $Q(s,a)$, which is generally more useful for selecting actions in nondeterministic or unknown environments, there are few possible ways of employing neural networks. In all of them, network inputs are determined by the features $\vec{\phi}(s)$ of state s, like in state value approximators. If the number of available actions is relatively small, it is possible to have a separate network output for each action $a \in A$. Alternatively, the values of particular actions can be approximated by a set of independent networks, each with a single output [208]. Clearly, neither of these approaches can be applied in tasks with numerous or continuous actions. A straightforward way to deal with such tasks is to use a single network with state features as inputs and a single output interpreted directly as an action. Yet another idea is to provide the action as an additional input for the network, to obtain its value at network's output.

2.2.2 *Temporal Difference Learning*

Temporal difference learning (TDL) is the most representative class of model-free reinforcement learning algorithms that rely on value functions (see Section 2.1.2). It was introduced by Sutton [186], but its origins date back to the famous checkers playing program designed by Samuel [169]. Like other value function based methods, TDL aims at learning the optimal value function (or its approximation), from which the optimal policy could be derived. Basically, TDL works by estimating the values of states or actions on the basis of other, hopefully more accurate estimates. This idea, known as bootstrapping, is featured also by the DP methods (cf. Section 2.1.3).

However, instead of exploiting the model-based Bellman equations (see Equation 2.8), the TDL algorithms are inherently model-free and estimate value functions from the training experience generated by interactions with the unknown environment. Moreover, TDL is typically applied in an incremental and online manner, in which the algorithm processes every single action taken by the agent in the environment.

2.2.2.1 *Value Prediction*

As an intermediate step towards improving current policy π, the TDL methods attempt to solve the *prediction* problem, i.e., compute its state value function V^π. Only then the policy can be adjusted by making it *greedy* with respect to the estimated value function (cf. Eq. 2.15). To estimate V^π, the TDL methods use the experience gathered by the agent following policy π in the given environment. Whenever the agent gathers a unit of training experience $(s_t, a_t, r_{t+1}, s_{t+1})$ resulting from a transition $s_t \to s_{t+1}$ and reception of r_{t+1} reward, it updates the current estimate of the value function V_t into V_{t+1}.

Prediction problem

In particular, the simplest TDL value prediction algorithm known as $TD(0)$ [186], updates the estimate of state value function with the following rule:

$TD(0)$ algorithm

$$V_{t+1}(s_t) = V_t(s_t) + \alpha(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)). \tag{2.9}$$

This rule attempts to minimize the difference between the current prediction of cumulative future reward $V_t(s_t)$ and the one-step-ahead prediction, where the latter involves the actual (received) reward r_{t+1} and is equal to $r_{t+1} + \gamma V_t(s_{t+1})$. Consequently, the *error* between the successive predictions $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$ is used to adjust the value of state s_t. Importantly, the *learning rate* $\alpha \in [0, 1]$ determines the size of correction.

The $TD(0)$ algorithm can be naturally extended by looking further than only one step ahead and using the subsequent actual rewards to update the value of the current state. In the extreme case, the algorithm would wait till the end of the training episode to know the exact discounted sum of rewards R_t (see Equation 2.11), compute the prediction error, and employ the update rule analogous to 2.9:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha(R_t - V_t(s_t)), \tag{2.10}$$

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.11}$$

This extreme variant of TD belongs to the class of *Monte Carlo* (MC) methods, which estimate the value function using the empirical (actual) sum of rewards of R_t rather than its estimation. Noteworthy, for deterministic environments, the update rule (2.10) can be said to implement supervised learning, because R_t is the correct (desired) value of $V(s_t)$.

Monte Carlo methods

2.2.2.2 *Value Prediction with Function Approximation*

Although both $TD(0)$ and Monte Carlo methods share the underlying idea of estimating state value function from samples of experience, they represent two extremes in implementing this process. MC-based methods need to wait until the end of an episode, when its exact outcome is known and can be back-propagated to correct the predictions made for the previously encountered states. $TD(0)$, to the contrary, waits only one time step, calculates the error between temporally successive predictions, and uses it to update the estimate of current state's value. $TD(\lambda)$ is an elegant umbrella that embraces the above special cases of $TD(0)$ and MC, which is equivalent to $TD(1)$. It allows to adjust the lookahead horizon by tuning the λ parameter, which makes the algorithm looks further in the future to compute temporal difference in estimations of state values.

$TD(\lambda)$ methods

In practice, the TDL algorithms are often combined with value function approximators (cf. Section 2.2.1) to allow generalization across large state spaces. In such situations, the value function is approximated by $V_{\vec{\theta}}$ — a differentiable function of the parameter vector $\vec{\theta}$ (i.e., for any $\vec{\theta} \in \mathbb{R}^d$, $V_{\vec{\theta}} : S \to \mathbb{R}$ is such that a gradient $\nabla_{\vec{\theta}} V_{\vec{\theta}}(s)$ exists for every $s \in S$). For example, if a neural network is employed to implement this function, the vector $\vec{\theta}$ would contain all network weights. Consequently, instead of adjusting the values of particular states (e.g., using equations 2.9 or 2.10), the learning algorithm operates on the vector of parameters. A variant of TDL that adjusts these parameters proportionally to the negative gradient of the squared prediction error is called *gradient descent* $TD(\lambda)$ [186, 189]. Its update rule is:

Gradient descent $TD(\lambda)$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t \sum_{k=1}^{t} (\gamma\lambda)^{t-k} \nabla_{\vec{\theta}} V_{\vec{\theta}_k}(s_k), \tag{2.12}$$

$$\delta_t = r_{t+1} + \gamma V_{\vec{\theta}_t}(s_{t+1}) - V_{\vec{\theta}_t}(s_t), \tag{2.13}$$

where the gradient $\nabla_{\vec{\theta}} V_{\vec{\theta}}$ is the vector of partial derivatives of value approximation for a given state with respect to each parameter. This rule illustrates that the trace decay parameter $\lambda \in [0, 1]$ determines the rate of 'aging' of the past gradients, i.e., the rate at which their impact on the current update decays when reaching deeper into the history. This general formulation of $TD(\lambda)$ takes into account the entire sequence of states and the corresponding predictions that appeared in a single episode up to time step t; in the case of $TD(0)$, the weight update is determined only by its effect on the most recent estimation:

Gradient descent $TD(0)$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t \nabla_{\vec{\theta}} V_{\vec{\theta}_t}(s_t). \tag{2.14}$$

Algorithm 2.1 Online gradient-descent $TD(\lambda)$ for learning policy. The approximate value function $V_{\vec{\theta}}$ is parametrized by $\vec{\theta}$.

Require: learning rate α, decay rate λ, number of training episodes n, exploration rate ϵ, deterministic MDP $\langle S, A, T, R, I, \gamma \rangle$

$\vec{\theta} \leftarrow$ INITIALIZE PARAMETERS
for $i = 1$ **to** n **do**
  $\vec{e} \leftarrow \vec{0}$
  $s \leftarrow$ INITIALIZE STATE(I)
  while $\neg$IS TERMINAL STATE(s) **do**
    with probability ϵ **do** $a \leftarrow$ RANDOM($A(s)$)
    else $a \leftarrow \arg\max_{a \in A(s)}(R(s,a) + \gamma V_{\vec{\theta}}(T(s,a)))$
    $r = R(s,a)$
    $s' = T(s,a)$
    if IS TERMINAL STATE(s') **then** $\delta = r - V_{\vec{\theta}}(s)$
    else $\delta = r + \gamma V_{\vec{\theta}}(s') - V_{\vec{\theta}}(s)$
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_\theta V_{\vec{\theta}}(s)$
    $\vec{\theta} = \vec{\theta} + \alpha\delta\vec{e}$
    $s \leftarrow s'$
  end while
end for

2.2.2.3 *Learning Policies*

Although in principle the $TD(\lambda)$ algorithms are used to solve the prediction problem (i.e., estimate V^π for a given policy π), if the model of the environment is available, they can be also employed to learn policies represented implicitly by value functions. Basically, the idea is to incrementally adjust the value function of a continually changing policy computed on-the-fly from the value function itself [226]. Such a scenario is illustrated in Algorithm 2.1, which demonstrates online learning of policy by gradient descent form of $TD(\lambda)$ algorithm for an approximated value function $V_{\vec{\theta}}$.

After initialization of the vector of parameters $\vec{\theta}$, the training experience is collected in n training episodes. Each episode starts from the initial state drawn from the distribution I and continues until a terminal state is reached. In the meantime, actions are taken according to the recent value function, which implicitly represents the policy which for each state s chooses the action leading to the most valuable successor state s' (the policy is greedy with respect to V):

From value function to policy

$$\pi(s) = \arg\max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V(s')). \tag{2.15}$$

However, such a greedy policy is not followed all the time, but to maintain sufficient environment exploration, with probability ϵ, a random action is taken instead (i.e., an ϵ-greedy policy is used).

After each state transition, the prediction error δ is calculated (see Equation 2.13), which is then used to adjust the value function parameters according to the gradient descent form of $TD(\lambda)$. Importantly, to avoid storing past gradients and calculating their sum at each time step (cf. Equation 2.12) the updates are calculated incrementally using the idea of *eligibility traces* [186, 189]:

Eligibility traces

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}} V_{\vec{\theta}_t}, \tag{2.16}$$

$$\vec{e}_0 = \vec{0}. \tag{2.17}$$

As a result, the parameters update rule can be formulated as:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t. \tag{2.18}$$

Since each update of the value function parameters aims to improve the prediction accuracy and the actions are taken greedily with respect to the most recent value function, the overall performance is likely to be improved, but the convergence cannot be guaranteed.

Applications in games

Learning policies implicitly represented by state value functions turns out to be particularly useful in the domain of games, in which the transition model is available and can be used to compute the resulting position for each move. Since the influential work of Tesauro [197, 198] and the success of his TD-Gammon player, variations of the $TD(\lambda)$ algorithm have become a well-known approach for elaborating game-playing strategies without human knowledge or expert strategies given a priori. A lot of research has been conducted with learning strategies for the games of Go [164, 172], Othello [207, 125] or Chess [14, 203].

Nevertheless, if the environment model is not known, taking actions on the basis of the state value function (cf. Equation 2.15) is impossible. Instead, an algorithm that learns control policies can rely on the action value function $Q(s, a)$ which allows for choosing actions just by comparing their values in the particular state:

$$\pi(s) = \arg\max_{a \in A(s)} Q(s, a). \tag{2.19}$$

Q-learning and SARSA

The two most recognized algorithms for model-free learning of $Q(s, a)$ functions are *Q-learning* [215] and *SARSA* [163]. Both algorithms represent the class of temporal difference learning and operate in the same incremental fashion as $TD(\lambda)$. For instance, after gathering a unit of training experience, the *Q-learning* algorithm updates the values of actions with the following rule:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a \in A(s_{t+1})} Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

Since this update rule essentially utilizes one-step lookahead to compute the prediction error, it can be considered as analogous to $TD(0)$ update rule expressed in Equation 2.9. Additionally, both Q function learning algorithms can be combined with function approximation and employ eligibility traces [150, 225].

2.2.3 *Evolutionary Algorithms*

As discussed in the previous section, one approach to solving reinforcement learning problems is temporal difference learning which relies on value functions to indirectly represent behavior policies. An alternative model-free approach is to search the space of policies Π directly in order to find a solution $\pi^* \in \Pi$ that maximizes the return $J(\pi)$ in a given MDP (see Equation 2.3). This idea can be realized by employing a general purpose optimization algorithm driven by the objective function $f : \Pi \to \mathbb{R}$. Typically, the value of the objective function $f(\pi)$ simply approximates the expected cumulative return by averaging the rewards obtained in multiple episodes in the given environment by policy π.

Direct policy search

Among many optimization techniques that could be applied to search for the optimal policy, *evolutionary algorithms* (EAs) form a class of stochastic algorithms that is particularly well-recognized for its successful applications in reinforcement learning [73, 146, 218, 221, 234]. EAs are bio-inspired optimization procedures that mimic the process of natural evolution including mechanisms such as genetic recombination, mutation, reproduction and selection [9, 51]. The most significant advantage of EAs lies in their flexibility — all they need to know about the optimization problem being solved is how to evaluate a quality (so called *fitness*) of a candidate solution (here: f).

Evolutionary algorithms

The many variants of EAs, including Genetic Algorithms [69], Evolution Strategies [17], and Genetic Programming [107], share a common set of underlying features:

- Unlike many conventional optimization methods, which iteratively process a single search point, EAs maintain a set of candidate solutions represented as a *population* of *individuals*. By sampling many points in the search space simultaneously, they implement a collective optimization process which allows them to avoid the problem of getting stuck in local minima.

  Implicit parallelism

- The *survival of the fittest* principle is used to iteratively refine the population of solutions. In each iteration (*generation*), all individuals in the population are evaluated and assigned fitness values. These values steer the selection process, which favors the better solutions and makes them more likely to contribute offspring to the next generation.

  Selection pressure

- The offspring of a population is generated in a randomized process that models the natural phenomena of *mutation* and *recombination*. These variation operators perform search in the space of candidate solutions and correspond to self-replication of slightly modified individuals and combining information from two or more selected individuals, respectively.

  Variation operators

Algorithm 2.2 General scheme of an evolutionary algorithm

```
P ← Create Random Population()
Evaluate Population(P)
while ¬Termination Condition() do
    S ← Select Parents(P)
    P ← Recombine And Mutate(S)
    Evaluate Population(P)
end while
return Get Fittest Individual(P)
```

A typical evolutionary algorithm

A scheme of a typical generational evolutionary algorithm is illustrated in Algorithm 2.2. In the first steps, the initial population of candidate solutions is created (typically in a randomized way) and evaluated according to the given fitness function. Afterwards, the algorithm proceeds iteratively in generations until the termination condition is met. In each iteration the fittest individuals are selected to act as parents and contribute offspring to the next generation. The offspring is created by copying parents or combining them using the crossover operator. Additionally, the newly-bred individuals can be subject to random mutations before they get evaluated. The termination condition typically depends on the number of such generations processed or on the quality of the best solution found so far. When fulfilled, evolution stops and the fittest individual is returned as the final solution.

This pseudocode is only intended to represent a general idea of how a typical evolutionary algorithm works. In particular, every single phase of the algorithm (including initialization, selection, recombination, evaluation etc.) can be implemented in various ways and a lot of research has been devoted to find which one is the best for particular problem classes or applications. The extensive presentation of evolutionary algorithms can be found in the book of Eiben and Smith [51], while their history and applications are surveyed by Bäck et al. [10].

2.2.3.1 *Evolutionary Reinforcement Learning*

Search space

Evolutionary algorithms can be used to solve reinforcement learning problems, as long as the latter can be framed as optimization problems. For this purpose, a space of candidate solutions and an objective function need to be defined. Since the goal is to find the optimal policy for an MDP with state space S and action space A, a candidate solution naturally corresponds to a control policy $\pi : S \to A$. However, there are many possible ways to express such a state-action mapping, and thus, depending on the problem being solved, a specific function representation needs to be chosen, e.g., neural networks.

Given the space of candidate solutions, the crucial issue is to specify the fitness function for evaluating their quality. Since in RL problems the objective is to maximize the policy return, EAs could calculate the fitness of a policy by simulating an interaction episode in the given MDP environment and summing the obtained rewards. However, the precise fitness evaluation of a policy may be non-trivial for at least two reasons: 1) stochastic transitions or rewards, and 2) many possible initial states of the environment. In such cases the policy return is a random variable and a single episode is not enough to evaluate the policy reliably.

Fitness function

In order to discuss how EAs fit into the general scheme of model-free learning from interactions, let us recall Figure 2.1. The first difference to mention, when compared to previously described temporal difference learning methods (cf. Section 2.2.2), is that such *evolutionary learning* is essentially based on the population of learners. The learning algorithm maintains a set policies and in each generation employs every policy to control the agent's behavior in the given environment. Training experience gathered in this way is used by the algorithm to compare the policies and select the best of them for further random adjustments. Importantly, from all the training experience the evolutionary algorithm cares only about the sum of rewards. Instead of investigating particular state transitions it treats the whole interaction episode just as a means of estimating the policy return. Therefore, it does not make use of all information that is available:

Evolutionary learning from interactions

> *Evolutionary methods ignore much of the useful structure of the reinforcement learning problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects.*
> (Sutton and Barto [189], p. 9)

Consequently, the evolutionary learning algorithm does not process the training experience *online*, after every single interaction, but waits until all episodes are completed. Only then the algorithm can reason about the fitness of particular policies. In such *offline*[3] mode of learning, policies remain unchanged for entire episodes of interactions.

Online vs. offline learning

To evaluate a policy in the above EA sense, one must strictly follow it during interaction trials. As it cannot change during a trial (or even during the entire evaluation act), the agent cannot explore the environment by trying new actions. This is in sharp contrast with temporal difference learning techniques, in which the agent intentionally deviates from its policy (using, e.g., ϵ-greedy exploration strategy) to discover potentially more rewarding alternative actions. Nevertheless, evolutionary learning algorithms are also capable of developing

3 Alternatively, the distinction between offline and online RL algorithms may depend on whether the algorithm use a simulator to generate training experience or gathers it directly from the real system.

Exploration in evolutionary learning

new policies but through exploration performed at the level of entire policies. In fact, both exploration (meant as discovering new regions of the policy space) and exploitation that concerns visiting neighborhoods of previously encountered candidate solutions are fundamental concepts for any search algorithm [209]. Evolutionary algorithms, which perform exploitation and exploration by means of selection and genetic operators (mutation and crossover), are well-recognized for keeping the balance between those two aspects of search. [130].

2.2.3.2 *Neuroevolution*

Policy representation

An important design choice in any direct policy search method (including EAs) concerns the representation of policies. Representing policies directly as state-action mappings (technically: tables) may be impossible particularly in MDPs with large or continuous state spaces. In analogy to temporal difference learning methods (see Section 2.2.2), this problem can be mitigated by employing function approximators like neural networks (see Section 2.2.1). Applying evolutionary algorithms to learn parameters of neural networks is known as *neuroevolution* [61, 231] and has been reported successful in many reinforcement learning domains [73, 90, 196].

Parametric policy space

Typically, neuroevolution assumes a fixed neural network topology and focuses solely on evolving its weights. In such cases, the space of considered policies can be seen as a *parametric policy space* [2, 83] $\Pi = \{\pi_{\vec{\theta}} \mid \vec{\theta} \in \mathbb{R}^d\}$, where each policy $\pi_{\vec{\theta}}$ is represented by a network parametrized with a d-dimensional weight vectors $\vec{\theta}$. Consequently, the learning algorithm performs a search in the parameter space $\Theta = \{\vec{\theta} \mid \vec{\theta} \in \mathbb{R}^d\}$ in order to find the optimal vector of parameters, i.e., such that maximizes the return of the corresponding policy:

$$\vec{\theta}^* = \arg\max_{\vec{\theta} \in \Theta} J(\pi_{\vec{\theta}}). \tag{2.20}$$

Evolution strategies

Evolution strategies (ES) is a variant of evolutionary algorithm that is particularly well suited for solving problems in which solutions are represented as real-valued vectors. Algorithm 2.3 demonstrates one of the flagships of ES, the $(\mu + \lambda)$-ES algorithm, applied to evolve weights $\vec{\theta}$ of a fixed-topology neural network in order to optimize the corresponding policy $\pi_{\vec{\theta}}$ for the given MDP.

$(\mu + \lambda)$-ES initialization

Algorithm 2.3 follows the general scheme of evolutionary algorithm presented in Algorithm 2.2. μ and λ are parameters that control the population size and the degree of elitism of the algorithm. At first, the initial population containing $\mu + \lambda$ individuals is created. Each individual is a d-dimensional vector drawn from the normal distribution with standard deviation σ_{init}. Next, the fitness of each individual is evaluated, which consists in simulating n interaction episodes in the given environment and calculating the average cumulative reward received when following the policy encoded by the

Algorithm 2.3 $(\mu + \lambda)$-*ES* for optimizing weights of a neural network with respect to its performance as an MDP policy.

Require: μ, λ, number of parameters d, number of generations g, number of training episodes n, MDP $\langle S, A, T, R, D, \gamma \rangle$

function EVOLUTION STRATEGY(μ, λ, g, MDP, n)
  $\mathcal{P} \leftarrow \{\vec{\theta}_i \sim \mathcal{N}_d(\vec{0}, \sigma_{init} \cdot \mathbf{I}_d),\ i = 1, ..., \mu + \lambda\}$
  $\mathcal{F} \leftarrow \{$EVALUATE INDIVIDUAL$(\mathcal{P}_i, MDP, n),\ i = 1, ..., \mu + \lambda\}$
  for $i = 1$ **to** g **do**
    $\mathcal{S} \leftarrow$ SELECT PARENTS ACCORDING TO FITNESS$(\mathcal{P}, \mathcal{F}, \mu)$
    $\mathcal{O} \leftarrow \{\mathcal{S}_{i \bmod \mu} + \mathcal{N}_d(\vec{0}, \sigma_{mut} \cdot \mathbf{I}_d),\ i = 1, ..., \lambda\}$
    $\mathcal{P} \leftarrow \mathcal{S} \cup \mathcal{O}$
    $\mathcal{F} \leftarrow \{$EVALUATE INDIVIDUAL$(\mathcal{P}_i, MDP, n),\ i = 1, ..., \mu + \lambda\}$
  end for
  return $\mathcal{P}_{\arg\max_i \mathcal{F}_i}$
end function

function EVALUATE INDIVIDUAL($\vec{\theta}, MDP, n$)
  $NN_{\vec{\theta}} \leftarrow$ INITIALIZE NEURAL NETWORK$(\vec{\theta})$
  for $i = 1$ **to** n **do**
    $s \leftarrow$ INITIALIZE STATE(MDP)
    while $\neg$IS TERMINAL STATE(s) **do**
      $a \leftarrow$ CHOOSE ACTION$(s, NN_{\vec{\theta}}, MDP)$
      $s, r \leftarrow$ TAKE ACTION(s, a, MDP)
      $fitness \leftarrow fitness + r$
    end while
  end for
  return $fitness/n$
end function

individual. It is worth to point out that exactly the same $(\mu + \lambda)$-*ES* algorithm could be used to solve any other optimization problem in which the solution can be expressed as a real vector — the only problem-specific part is the fitness evaluation.

The evolutionary loop

When the initial evaluation is completed, the algorithm repeats the evolutionary loop of selection, breeding, and evaluation for a g generations. Each generation starts from sorting the individuals according to fitness in descending order. The μ fittest individuals (which form the so called *elite* of the population) are chosen as the parents of the next generation. To obtain λ new individuals, each parent is cloned λ/μ times and its copies are mutated by adding random Gaussian noise of zero mean and standard deviation of σ_{mut}. Algorithm 2.3 illustrates the basic variant of ES, which does not involve recombination, thus the offspring is generated solely by the mutation operator. The next generation, created by merging the parents with

theirs newly-bred offspring, contains $\mu + \lambda$ individuals, which are then evaluated. After g generations, the individual with the highest fitness is returned as the result of the algorithm.

Neural network as a policy

Apart from choosing a specific evolutionary algorithm, an important technical issue is how to execute agent's policy, i.e., choose actions, using a given neural network (cf. line 18 of the Algorithm 2.3). Typically, neural networks are employed as a state or action evaluators. As a results, network's role resembles that of $V(s)$ and $Q(s, a)$ and thus its functioning could be based on the same principles as in approximating these function (see discussion in Section 2.2.1.2). The bottom line is that network's inputs are determined by the current state of the environment s, while its output evaluates this state or directly indicates which action is the best to take in this state.

Topology and Weight Evolving Neural Networks

Most neuroevolutionary methods do not change the architecture of the network in the course of evolution. For instance, when employing multilayer perceptrons, there is a predetermined number of neurons in each layer which are fully connected with each other layer-to-layer. However, relying on a fixed, manually predetermined architecture can significantly limit *evolvability* of the neural network. For this reason, several methods for discovering good network topologies during learning have been proposed. In such approaches, generally termed Topology and Weight Evolving Neural Networks (TWEANN), both weights and topology of the network are subject to optimization and can change along an evolutionary run. The most popular TWEANN methods include NeuroEvolution of Augmenting Topologies (NEAT) proposed by Stanley and Miikkulainen [184] and Symbiotic, Adaptive Neuro-Evolution (SANE) introduced by Moriarty and Miikkulainen [137]. Capability of evolving simultaneously both weights and structures of function approximators is one of the main strengths of evolutionary learning when compared with temporal difference learning techniques.

3

SHAPING BACKGROUND

In this chapter we present shaping techniques applied to teaching both living organisms and intelligent agents. We start by describing the idea of shaping that originates from the field of behavioral psychology and was applied in human and animal learning (Section 3.1). Afterwards, in Section 3.2, we provide a brief literature review of shaping-related approaches in computational reinforcement learning. In particular we analyze specific motivations for shaping and present some inspiring examples of shaping in robotics.

3.1 SHAPING IN ANIMAL AND HUMAN LEARNING

In order to fully understand the inspiration of our shaping approach we need to take a look at the early inspirations of the whole domain of reinforcement learning. Since this machine learning paradigm can be regarded as a computational counterpart of the trial-and-error learning process which occurs in nature, it was largely influenced by the research in the field of learning animals. The pioneering research in this field was conducted in the first half of the 20th century by, among others, two famous psychologists: Edward Lee Thorndike and Burrhus Frederic Skinner. Both of them have devised innovative animal training procedures to investigate the effects of consequences on developing new behaviors in living organisms. The observations they made allowed Skinner to define later the principles of *operant conditioning* — a type of learning in which reinforcing or punishing an action influences the future rate of repeating this action.

Operant conditioning

3.1.1 *The Law of Effect*

Although it is Skinner who is regarded as the father of operant conditioning, it was who first studied how behavior changes on the basis of its consequences Thorndike [199]. His most famous experiments involved home-made puzzle boxes in which he placed hungry cats. A cat could escape from a box to obtain food only by operating a series of latches and levers. The first successful escapes occurred by accident — thanks to the trial-and-error exploration of the box. Thorndike plotted a learning curve by measuring the amount of time needed for the cat to escape in successive trials. He discovered that after first few episodes, the escape time gradually declined since the cat was able to quickly recall how to behave to receive a reward.

Thorndike's puzzle boxes

From these experiments, he concluded that if an action brings a desirable consequence, it becomes 'stamped in' and will likely be repeated in the future, which explained how animals develop new habits in their behaviors. He described this phenomenon as the *Law of Effect:*

> *Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.* (Thorndike [200], p. 244)

The essence of trial-and-error learning

According to Sutton and Barto [189], Thorndike's procedures involved the essence of trial-and-error learning by combining both *search* and *memory*. Indeed, initially the cat was performing a random search by trying many different actions in an attempt to escape the box. Later, by comparing the results of such actions it could associate the rewarding behavior with a particular situation in order to memorize it and repeat it in the future.

3.1.2 *Discovery of Shaping*

Reinforcement and punishment

Following the work of Thorndike, Skinner [180] introduced the terms of *reinforcement* and *punishment* which formed the basis for his operant conditioning theory. Both terms refer to consequences of behavior which modify the organism's tendency to repeat that behavior in the future. Skinner believed that the ability to associate particular actions with the following reinforcements is a basic learning mechanism which allows animals to optimize their behavior in a given environment. This belief can be regarded as one of the main motivations for computational reinforcement learning paradigm.

The Skinner Box

The most of Skinner's studies on operant conditioning was based on his experiments with training rats in an improved version of the puzzle box, which he called an operant conditioning chamber (known also as the Skinner Box). In these experiments he observed a fundamental problem in Thorndike's trial-and-error learning procedures. The problem arose when the target behavior was too complex and it was never performed by the animal in the course of random environment exploration. In such situations there was nothing to be reinforced and thus learning could not take place. To deal with this problem, Skinner proposed a variant of operant conditioning that was later named *shaping*.

Shaping is defined as a method of successive approximations. Instead of reinforcing only the target behavior, which can be difficult to achieve by accident, any action that results in a progress towards behavior is rewarded. Shaping was discovered by Skinner during his work on Project Pigeon aimed at training pigeons to control the trajectory of flying bombs. The discovery is described by Peterson [151] as a day of great illumination. Skinner explained it as following:

A day of great illumination

> *We first give the bird food when it turns slightly in the direction of the spot from any part of the cage. This increases the frequency of such behavior. We then withhold reinforcement until a slight movement is made toward the spot. This again alters the general distribution of behavior without producing a new unit. We continue by reinforcing positions successively closer to the spot, then by reinforcing only when the head is moved slightly forward, and finally only when the beak actually makes contact with the spot...*
>
> *The original probability of the response in its final form is very low; in some cases it may even be zero. In this way we can build complicated operants which would never appear in the repertoire of the organism otherwise. By reinforcing a series of successive approximations, we bring a rare response to a very high probability in a short time... The total act of turning toward the spot from any point in the box, walking toward it, raising the head, and striking the spot may seem to be a functionally coherent unit of behavior; but it is constructed by a continual process of differential reinforcement from undifferentiated behavior, just as the sculptor shapes his figure from a lump of clay.*
> (Skinner [181], pp. 92-93)

Although reinforcement allows to control animal's behavior, some tasks are simply to difficult to be learned directly. Shaping can be considered as a developmental approach, in which the learner is trained on a pedagogical sequence of tasks of increasing difficulty.

3.1.3 *Scaffolding and the Zone of Proximal Development*

A similar problem to that encountered by Skinner in training animals was later observed in teaching humans. Sometimes solving a task requires a too high level of skills or knowledge to be approached by a student. In such situation, some sort of bias must be provided to allow the student accomplish a demanding task. In the *instructional scaffolding* proposed by Wood et al. [230], the bias is formed not by simplifying a task (as it was done in animal's behavior shaping) but through additional support and assistance of the teacher. Thanks to the new external source of information and experience, tailored to the needs of the student, learning progress can be sustained.

Instructional scaffolding

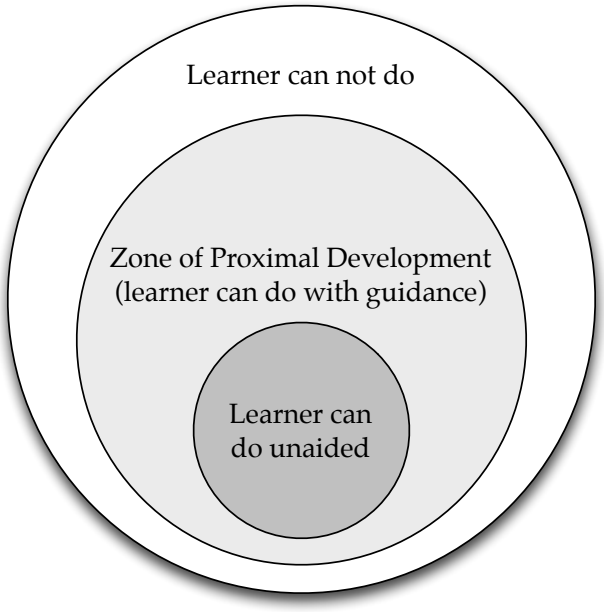


Figure 3.1: Graphical representation of the zone of proximal development. The middle circle illustrates the tasks that are too difficult to be learned without help, but that can be mastered with guidance of a knowledgable teacher.

The zone of proximal development

A closely related term is the *zone of proximal development*, defined by Vygotsky [213]. This term indicates how the acquisition of new knowledge depends on previous learning and the availability of teacher's guidance (cf. Fig. 3.1). In order to be solved, a task must be kept in this zone. Otherwise, the learner will be unable to gain competence in the task even if provided with the help of an experienced teacher. Vygotsky [212] describes the zone of proximal development as:

> *... the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers. For example, two 8 yr. old children may be able to complete a task that an average 8 yr. old can do. Next, more difficult tasks are presented with very little assistance from an adult. In the end, both children were able to complete the task.*
> (Vygotsky [212], pp. 92-93).

3.2 SHAPING IN COMPUTATIONAL REINFORCEMENT LEARNING

Despite the appealing inspirations of shaping techniques, they have not been widely adopted in computational reinforcement learning. This can be seen as quite surprising since reinforcement learning of intelligent agents faces very similar difficulties to those observed in learning of living organisms. In particular, there is a major problem caused by the fact that reinforcement is delayed and often too scarce. For example, in learning game-playing policies, the reward signal typically occurs only after finishing the game and is directly determined by its outcome. In the case of learning an extremely difficult game where chances of winning by making just random moves are practically negligible, an agent with zero knowledge never receives positive reinforcement and learning simply can not occur.

Delayed reinforcement ...

However, delayed rewards are not the only aspect of reinforcement learning tasks that make them difficult to solve. In this section we discuss other reasons of task difficulty that constitute direct motivations for using shaping to facilitate computational reinforcement learning (see Section 3.2.1). Afterwards, we provide a brief survey of the most interesting studies that attempted to provide an easier path to learning by the means of methods called shaping.

3.2.1 *Specific Motivations*

The general motivation for shaping concerns aiding an intelligent agent in mastering a difficult task. Here, we consider specific motivations by analyzing the reasons that make reinforcement learning tasks difficult to solve by conventional approach. In the following list we associate these reasons with particular elements of the underlying Markov Decision Process $\mathcal{M} = \langle S, A, T, R, I, \gamma \rangle$ (cf. Section 2.1.1):

... and other reasons of task difficulty

- *Large state space.* If the number of states is large, it may be very time-consuming to find a policy that generalizes well across the entire state space (cf. Section 2.2.1). Recall from Section 2.1.3 that the size of the state space grows exponentially with the number of variables describing the state (this phenomenon is known as the *curse of dimensionality* [15]).

- *Large action space.* The number of possible policies of the form $\pi : S \to A$ grows exponentially with the number of possible actions. As the size of the search space increases, it makes the problem of finding the optimal solution effectively harder.

- *Nondeterministic transition function.* When taking the same action in the given state may result in different transitions, decisions are made under uncertainty. For this reason, to acquire confidence in reasoning about the potential consequences, many

repeated exploration trials must be performed. Apart from state transitions, also the received rewards may be stochastic.

- *Delayed rewards.* Rewards are delayed in time and provide only minimal feedback about the agent's performance (e.g. limited to indicating final success or failure). Consequently, the agent is hardly able to reason about the quality of a given policy or its particular actions. This problem is known as *temporal credit assignment* and constitutes one of the main challenges in reinforcement learning.

Alleviating difficulties

Importantly, each of the above motivations does not exclude the others and thus a given task can be difficult for a number of reasons. Moreover, one aspect of the task can be influenced by changing the other ones, so difficulty can be reduced in different ways. For instance, to alleviate the problem of delayed rewards, we could modify the task by appropriately modulating the reward signal or by changing the initial state distribution to make an agent start near the highly rewarded states.

3.2.2 *Shaping Principles*

Task modification

Driven by the above motivations, most of the existing approaches to shaping in computational reinforcement learning consist in modifying the given MDP $\mathcal{M}$, which is assumed to be too hard to be learned directly by a traditional RL approach. The modification typically concerns only one of the elements of the MDP, and thus, a modified task $\mathcal{M}'$ resembles the original one but may have, e.g., a different reward function, i.e., $\mathcal{M}' = \langle S, A, T, R', I, \gamma \rangle$. Importantly, the ultimate goal remains unchanged, and $\mathcal{M}'$ is only expected to provide an easier path to learning the original task $\mathcal{M}$, while keeping the same overall computational cost. Put another way, by changing the training environment the agent is supposed to gather useful training experience that will facilitate learning of the original task.

The quintessence of shaping can be expressed by the following question:

> *When faced with a complex problem, is it better to tackle it directly with a standard RL algorithm, or to first solve a related simple problem and apply the experience gained to the complex problem?* (Madden and Howley [123], p. 391)

Sequence of task modifications

Moreover, according to Randløv [159], shaping can be essentially realized in two different ways. Although it is often limited to permanently changing the training environment to a single task modification $\mathcal{M}'$, it can also rely on a sequence of progressively more difficult task modifications culminating in the original (target) one.

Despite its appeal, shaping can be difficult to realize because a meaningful task modification usually requires substantial knowledge about the problem at hand. For this reason shaping is often regarded as a method of incorporating implicit domain knowledge into the learning process [127, 227]. The presence of an external supervisor that identifies useful task modifications for the shaping process is sometimes regarded as its inherent feature:

Shaping requirements

> *The essence of shaping is that of a supervised, iterative process, whereby the learned task is repeatedly modified in some meaningful way by an external trainer, so as to eventually bring the learning agent to perform the behavior of ultimate interest.* (Erez and Smart [54], p. 215)

3.2.3 *Inspiring Works in Robotics*

Most of early works on shaping concern learning robots to perform physical behaviors. This was a natural application for shaping as it roughly corresponds to teaching animals to master motor skills, a context in which shaping was historically applied for the first time (see Section 3.1).

To the best of our knowledge, the first work on computational shaping (albeit it does not refer to this particular term) is the study of Selfridge et al. [174], who demonstrated that solving a pole balancing task is easier if a learning system starts with mastering a simpler version of the task. Such *directed training* procedure can be implemented by modifying the parameters of the training environment that influence its transition function (e.g., by increasing the mass or shortening the length of the pole). Interestingly, the authors compared selecting a pedagogical sequence of training tasks to presenting the right training examples in supervised learning.

Directed training for pole balancing

One of the earliest works that explicitly uses the term 'shaping' is the study of Gullapalli and Barto [76] aimed at learning a robot hand to press keys on a simulated calculator keypad. They hypothesize that learning complex physical control behavior can be facilitated by providing the learner with some initial domain knowledge. Shaping is regarded as a natural way of introducing such knowledge through rewarding manually identified behavioral approximations of the target task. Although judging what constitutes a good approximation is in general not easy, even if domain knowledge is available, in the particular case of learning physical behavior, an approximation can be specified in terms of physical distances. Even though Gullapalli et al. successfully implemented shaping by manually designing a sequence of intuitive approximations of the target key-pressing task, this work clearly points to the need of more formalized and systematic shaping procedures.

Approximating physical distances

Supervised reinforcement learning

Dorigo and Colombetti [47] considered shaping as a *supervised* variant of reinforcement learning, which includes, apart from an agent and an environment, also a knowledgeable trainer responsible for guiding the learning process. It is the trainer who provides additional rewards for progressing towards the desired behavior, and thus, alleviates the problem of delayed reinforcement. Nevertheless, there is a need of specifying the *shaping policy*, i.e., trainer's strategy for providing reinforcements. In principle, the role of trainer could be played by a human expert. Otherwise, a dedicated *reinforcement program* should be implemented to automatically steer the learning process. However, programming a trainer for, e.g., a complex problem within the robotics domain can be as hard as solving the original problem.

Learning from easy missions

Asada et al. [8] focused on scaling a theoretical reinforcement learning method to larger and more complex robot learning problems. They also applied a form of shaping to cope with the delayed reinforcement problem in the learning vision-based robot to shoot a ball into the goal. In their *learning from easy missions* they began training by placing a robot and the ball near the goal. Only after mastering such a simplified task, they started to gradually moving the robot further from the goal:

> *This situation resembles a case in which the small child tries to shoot a ball into the goal, but he (or she) cannot imagine which direction and how far the goal is because a reward is received only after the ball has entered into the goal. Further, he (or she) does not know how to choose an action from several action commands. This is the famous delayed reinforcement problem due to no explicit teacher signal that indicates the correct output at each time step. Then, we construct the learning schedule such that the robot can learn in easy situations at early stages and learn in more difficult situations at later stages.*
> (Asada et al. [8], p. 285)

3.2.4 *Reward Shaping*

Artificial reward signal

The above examples of early shaping approaches demonstrate different ways in which original tasks can be modified to facilitate learning. In particular, the presented studies involve modification of reward function R, transition function T and initial state distribution I. More recent applications of shaping focus mainly on changing the first of these elements, which is typically realized by adding an artificial reward signal for training purposes. Such approach is commonly referred to as *reward shaping* [111, 127, 142] and is the closest to the original shaping procedure employed by Skinner (cf. Section 3.1.2).

Ideally, a reward function R should provide useful feedback about performed actions soon after they were taken. In many MDP tasks, however, reinforcement occurs only upon reaching the goal state. In order to mitigate the problem of temporal credit assignment, reward shaping creates an additional reward signal F that is expected to provide an intermediate feedback about the progress towards the goal. Consequently, the agent is trained in a more supportive environment with enhanced reward function, i.e., $\mathcal{M}' = \langle S, A, T, R + F, I, \gamma \rangle$. The crucial question is whether the optimal policy learned in such modified task is equivalent to the optimal policy for the original task. Importantly, Ng et al. [142] showed conditions that must be fulfilled by the additional reward function F in order to preserve the optimal policy between the tasks.

Intermediate feedback

3.2.5 *Related Approaches*

Most of the shaping approaches considered so far have been motivated by the problem of delayed reinforcement. However, the vast size of the space space has also stimulated a lot of research towards scaling and accelerating reinforcement learning [100, 136, 195]. One possible approach is *hierarchical reinforcement learning* [12, 43, 84] which boils to decomposing the original MDP into a hierarchy of smaller MDPs. Training experience gathered on such subtasks can be used to solve the original task more effectively.

Hierarchical RL

Another notable example of a shaping-related work is that of Madden and Howley [123]. The authors considered a sequence of logical tasks of progressive difficulty that were characterized by increasing size of the maze. They introduced the approach called *progressive RL* which alternates the cycles of experimentation and introspection. In the experimentation phase, the agent gathers the experience and learns a policy for a given task. Then, in the introspection phase, symbolic knowledge is extracted from the elaborated policy and applied to a new state space to create an initial policy for a next task in the sequence. The progressive RL provided a significant speed-up with respect to standard RL methods.

Progressive RL

Finally, in the particular case of evolutionary learning methods, the approach based on employing several fitness functions corresponding to progressively harder versions of the given task is referred to as *incremental evolution* [72, 140, 206]. We discuss this approach in more detail in Section 7.1.2.

Incremental evolution

4

COEVOLUTIONARY SHAPING

In the previous chapter we presented the existing shaping approaches applied in computational reinforcement learning. Here, we attempt to place these approaches in a unified shaping framework that delineates the role of shaping in a standard reinforcement learning scenario (see Section 4.1). We discuss the potential improvements of the learning process that can be achieved by the means of shaping. Afterwards, in Section 4.2 we describe coevolutionary algorithms and discuss how they are capable of implementing the shaping process autonomously.

4.1 UNIFIED SHAPING FRAMEWORK

The examples of existing shaping approaches in reinforcement learning indicate common need of good training experience. In a typical reinforcement learning setup (see Figure 2.2), it is implicitly assumed that an agent gathers training experience by interacting with the same environment that constitutes the goal of learning. Put another way, the original (target) MDP task for which the optimal policy is to be found, is also used for training purposes, i.e., to generate the samples of experience and to get the feedback concerning the quality of the current policy. Although such setting may seem straightforward and natural, it may be beneficial to diverge from it and employ different tasks (environments) for training. By using purpose-built training environments, the agent can be exposed to a more informative training experience that allows a specific learning algorithm to progress faster and to find better solutions in the assumed search space.

Sources of training experience

Throughout this thesis, we will use the term 'shaping' for any method that affects the training environment, but at the same time does not influence the reinforcement learning algorithm. Therefore, instead of tuning the parameters of the algorithms, we put the emphasis on finding useful source of training experience. Most of the existing shaping approaches discussed in Section 3.2 are compatible with this notion of shaping, which is presented graphically in Fig. 4.1. The grey box in the figure encapsulates the conventional RL scenario discussed earlier (see Section 2.1). Although this may remain transparent for the learning algorithm or the agent, in case of shaping the training interactions take place in purpose-built training environments. The figure emphasizes the distinction between these training environments and the target ones, in which the agent should ultimately perform well.

General notion of shaping

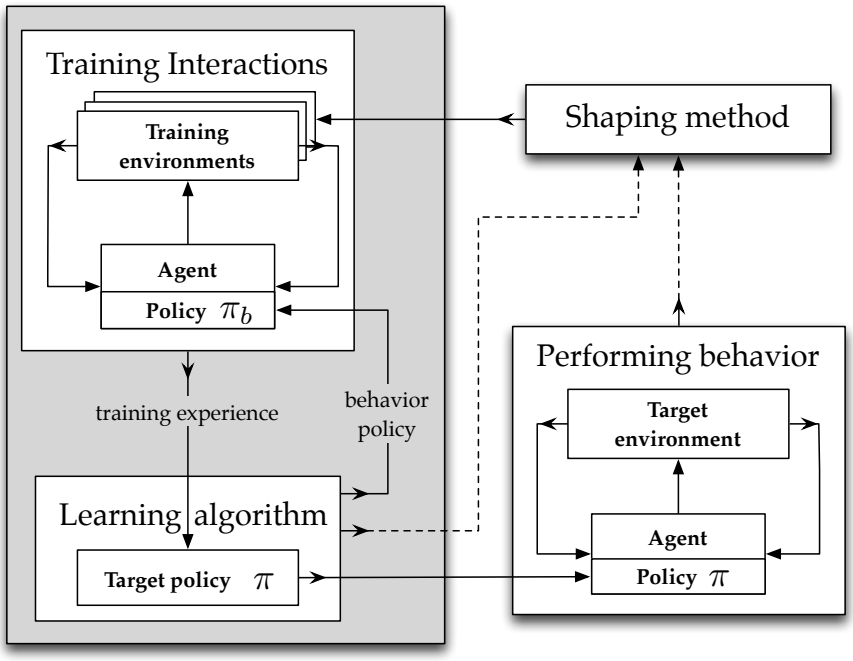


Figure 4.1: The place of shaping in computational reinforcement learning

The role of shaping

Let us explain Fig. 4.1 in detail. The role of the *Shaping method* is to provide such *Training environments* that can facilitate progress of a given *Learning algorithm*. To provide useful source of training experience, the shaping method must take into account the goal of learning (*Target environment*). Consequently, the training environments are typically closely related to the target one and differ only with respect to some element of the MDP definition (cf. Section 3.2.2). By constraining the scope of variations of the target task, we can assume that the role shaping method is limited to choosing the right tasks from a specific space (domain) of tasks. Additionally, the shaping method may work adaptively by exploiting the feedback from the learning algorithm (illustrated with a dashed line). For instance, it can analyze the results of training interactions and verify how the agent copes with the tasks provided so far. Preferably, on this basis it should adjust the training environment to construct a more 'pedagogical' learning gradient.

Transferring the policy

Importantly, in this thesis, we assume that a policy $\pi : S \to A$ learned in the training environments can be directly applied in the target one. This means in practice that the considered training and target environments share a common state space S and action space A[1] and can differ only with respect to transition function T, reward function R or initial state distribution I. However, in general, it is possible to employ training environments that vary also with respect to S and A. In such cases, an additional step may be needed which

1 Technically, to reuse a policy in a new task, it may be enough if the state and action description variables remain the same.

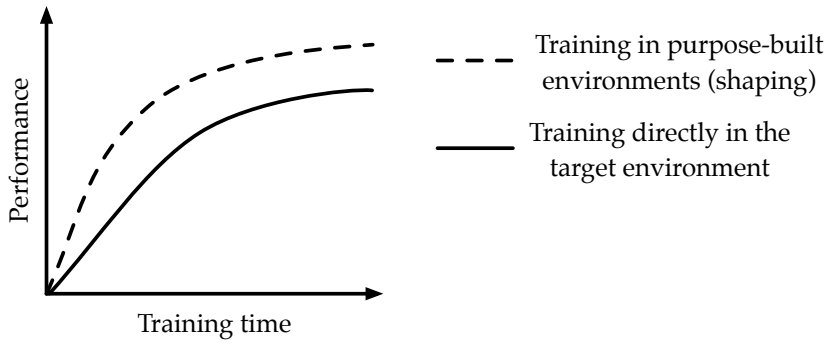


Figure 4.2: Potential benefits of shaping. The figure is adapted from Torrey and Shavlik [205].

converts the learned policy so that it is applicable in the target environments. This step is conventionally realized by *transfer learning* methods [195, 205] which attempt to transfer the knowledge gained in one task to another.

It is worth to note that the proposed shaping framework is not limited to the shaping techniques developed in past work. One particular approach that fits into the framework and is used together with evolutionary learning algorithms is known as *fitness modeling* or *fitness approximation* [95]. The main motivation behind this approach is the complexity of task dynamics, which causes large computational cost of computing state transitions and simulating interactions with the environment. The idea is to employ *surrogate* training tasks which approximate the target one but are simpler to simulate. Training tasks act as a proxy of the true goal which allows to effectively compute fitness of evolving individuals [105].

Fitness approximation

Moreover, although all the previously discussed works on shaping rely on manually selected training tasks, a shaping method can in principle work without human supervision by trying to autonomously identify useful task in the assumed space of task variations. Investigating the ideas of such knowledge-free shaping constitute the main purpose of this thesis. In particular, we employ competitive coevolutionary algorithms [154]. We expect that training experience provided by these algorithms will lead to both faster learning convergence and improved final performance. These two measures of learning effectiveness, commonly adopted in transfer learning [205], correspond to the dashed (shaped) learning curve in Fig. 4.2 being, respectively, steeper and higher than the one for unshaped (target) environment.

Knowledge-free shaping

Measures of learning effectiveness

In the following chapters we investigate particular shaping methods in the context of specific learning algorithms, namely, evolutionary algorithms (Chapters 6 and 7) and temporal difference learning algorithms (Chapters 6 and 8).

4.2 COEVOLUTIONARY SHAPING

Coevolutionary algorithms have been introduced into the field of computational intelligence as an alternative to conventional evolutionary algorithms described in Section 2.2.3. There are two main types of coevolutionary algorithms, namely, *competitive coevolutionary algorithms* and *cooperative coevolutionary algorithms*. The difference between them concerns the character of relationships between coevolving individuals (symbiotic cooperation or competition). Since we focus on the competitive form of interactions in this thesis, whenever the term 'coevolution' is used, it shall be interpreted as competitive coevolution. The reader interested in cooperative coevolutionary algorithms and its applications is referred to the works of Potter and De Jong [155], Gomez et al. [70], Krawiec and Bhanu [108].

Competitive and cooperative coevolution

In this section we provide a brief description of coevolutionary algorithms and we formalize the class of test-based problems for which they are typically applied. Afterwards, we discuss how coevolutionary methods can be generally applied in reinforcement learning. In particular we frame reinforcement learning problem as a test-based problem, which allows us to draw a close analogy between coevolution and shaping.

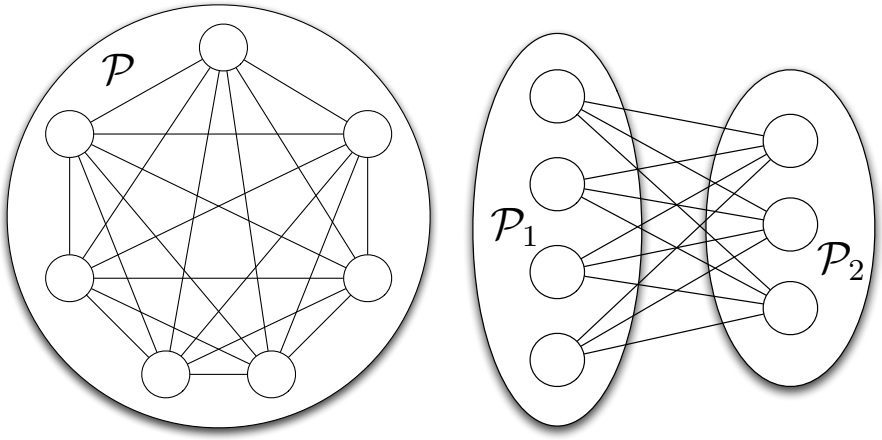
4.2.1 *Coevolutionary Algorithms*

Fitness function is an indispensable component of evolutionary algorithms that drives the search process by assigning fitness values to candidate solutions. It is also the fitness function that constitutes the major difference between *evolutionary* and *coevolutionary* algorithms. In evolutionary algorithms this function is usually expected to be *static* and reflect individual's absolute performance, which is assumed to be independent of other individuals in the population. Coevolutionary algorithms, by contrast, employ *dynamic* fitness functions that assess the relative performance of individuals with respect to other evolving individuals. As a result, fitness evaluation is *context-sensitive*: a candidate solution appearing well in one population may turn out to be poor when transferred to another.

Context-sensitive fitness evaluation

Coevolutionary algorithms are particularly well-suited to *interactive domains*, in which there is no intrinsic objective function given or such a function is costly to compute. Instead, in order to evaluate candidate solutions, coevolution relies on the outcomes of interactions between coevolving individuals. The abstract notion of *interaction* denotes here a procedure that reveals information about a pair of candidate solutions. The formal definition of interactive domains can be found in the work of Popovici et al. [154]. A canonical example of interactive domains are classic two-player board games like chess, backgammon

Interactive domains



(a) Single-population coevolution. (b) Two-population coevolution.

Figure 4.3: Round-robin interaction patterns.

or Othello. In these domains the interaction consists simply in playing a game between two individuals.

The simplest coevolutionary algorithm employs a single, homogeneous population of individuals, which interact directly with each other. Using such *single-population* coevolution is limited to symmetric domains, like the game of Othello, in which the roles in interactions are interchangeable. Although single-population coevolution has been intensely exploited in the context of games [5, 63, 115, 152], some results suggest that it can be generally more useful to maintain simultaneously two populations of individuals — a population of *solutions* (learners) and a population of *tests* (teachers) [24, 56, 85, 98, 162]. In such *two-population coevolution,* the interactions occur only between individuals that belong to different populations. When applied to evolving game-playing policies, each population contains the opponents used for evaluating players in the other population.

Single- and two-population coevolution

An important aspect of fitness evaluation in coevolution is the interaction scheme that determines which individuals should be confronted with each other [147]. The most common interaction scheme, known as *round-robin tournament* is illustrated in Fig. 4.3. In this scheme, every member of each population interacts with every other individual which can serve as a partner. Depending on the number of populations employed by the algorithm and their roles, the set of appropriate partners is different. Typically, in the single-population coevolution all other members of the population are used as opponents, while in the two-population coevolution — all members of the opposite population.

Interaction scheme

4.2.2 *Test-Based Problems*

Test-based problems [41, 91] belong to the class of *co-optimization problems* posed in interactive domains [154] and are conventionally approached by coevolutionary algorithms. In test-based problems, the quality of a candidate solution can be determined by performing interactions with a number of tests. Formally, a test-based problem can be defined [92] as a tuple $\langle \mathcal{S}, \mathcal{T}, \mathcal{G} \rangle$, in which:

- $\mathcal{S}$ is a set of *candidate solutions,*
- $\mathcal{T}$ is a set of *tests,*
- $\mathcal{G} : \mathcal{S} \times \mathcal{T} \to \mathbb{R}$ is an *interaction function.*

The objective in test-based problems is defined by a *solution concept* [56], which describes a subset of candidate solutions that constitute *solutions* to the problem. One of the most commonly used solution concepts is *maximization of expected utility* [40], i.e., maximization of the expected result of interaction with a randomly selected test. This solution concept can be used, for instance, to identify the best scoring white player strategy in the game of chess. To fully define such a test-based problem, S would include all possible white player strategies, the set T would contain all black player strategies, while the interaction G would simply correspond to a single game of chess.

Solution concepts

Since typically the set of all possible tests is very large , it is computationally infeasible to explicitly evaluate individuals on all of them. Therefore, approaching test-based problems with conventional evolutionary algorithms requires defining a computationally cheaper fitness function. The easiest way to reduce the computational complexity is to limit the number of tests used for fitness evaluation purposes. One approach, which was recently reported successful [33, 94], is to randomly sample the tests whenever fitness need to be evaluated. Another approach is based on a heuristic handcrafted fitness function. Designing such a function typically requires intimate knowledge of problem domain and its precise articulation. For instance, to evaluate chess-playing strategies, the function would need characterizing various aspects of expert play and observing these characteristics in the behavior of evaluated player.

EAs for test-based problems

Coevolutionary algorithms (see Section 4.2.1) attempt to select the tests in an adaptive, dynamic manner, and thus can in principle avoid potential biases resulting from the use of a fixed or manual selection of tests. Typically they maintain a population of candidate solutions that are rewarded for solving tests and a population of tests rewarded for challenging the solutions. Consequently, coevolution is believed to encourage an *arms race, "in which the two populations reciprocally drive one another to increasing levels of performance and complexity"* [162]. In other words, coevolution can be seen as method that simultaneously searches the space of solutions and evolves a fitness function:

Coevolutionary arms race

> *Coevolutionary algorithms progress from a simple intuition: evolve the fitness function together with the evolving individuals. By adjusting the challenge put to evolving individuals, we hope algorithms might tune the fitness function to push individuals into continually increasing their capabilities.*
> (Bucci and Pollack [25], p. 221)

4.2.3 *Coevolution for Reinforcement Learning*

Coevolutionary algorithms have been frequently applied in machine learning to evolve behaviors of autonomous agents. In particular, a lot of problems addressed by coevolution originates from the areas of robotics [60, 132, 144, 178] and game playing [19, 162, 152, 63]. Such problems can be usually framed not only as test-based problems but also as reinforcement learning problems, which inherently involve interactions between agents (policies) and their environments (tasks). Consequently, we can define a general *test-based reinforcement learning problem*, over a common space of states S and actions A:

Test-based reinforcement learning problem

- $\mathcal{S}$ — a set of policies of the form $\pi : S \to A$ which constitute candidate solutions to the given problem.
- $\mathcal{T}$ — a set of MDP tasks which play the role of tests. All tasks in $\mathcal{T}$ share the same state space S and action space A, so we can use the same policy $\pi : S \to A$ across the entire set $\mathcal{T}$. However, each task $\tau \in \mathcal{T}$ fully specifies its individual transition function T_τ, reward function R_τ and initial state distribution I_τ.
- $J : \Pi \times \mathcal{T} \to \mathbb{R}$ — an MDP interaction function, where $J(\pi, \tau)$ is the expected return obtained by the agent following policy π in task $\tau = \langle S, A, T_\tau, R_\tau, I_\tau, \gamma \rangle$:

$$J(\pi, \tau) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \mid R = R_\tau,\ T = T_\tau,\ s_0 \sim I_\tau \right].$$

Following the notion of solution concepts [56], we can define the goal of learning in such a problem. For the most studied solution concept of maximization of expected utility, this would boil down to specifying the solution π^* that satisfies:

$$\pi^* = \arg\max_\pi \mathbb{E}\left[J(\pi, \tau) \mid \tau \in \mathcal{T} \right].$$

Coevolutionary approach to such problems involves a population of policies and a population of tasks[2]. The interactions between individuals in both populations are used for fitness evaluation that

2 Sometimes a single entity can play both interaction roles, e.g. a game-playing policy can also specify the opponent in the training environment. In such *symmetric* cases, it is possible to apply a single-population coevolution.

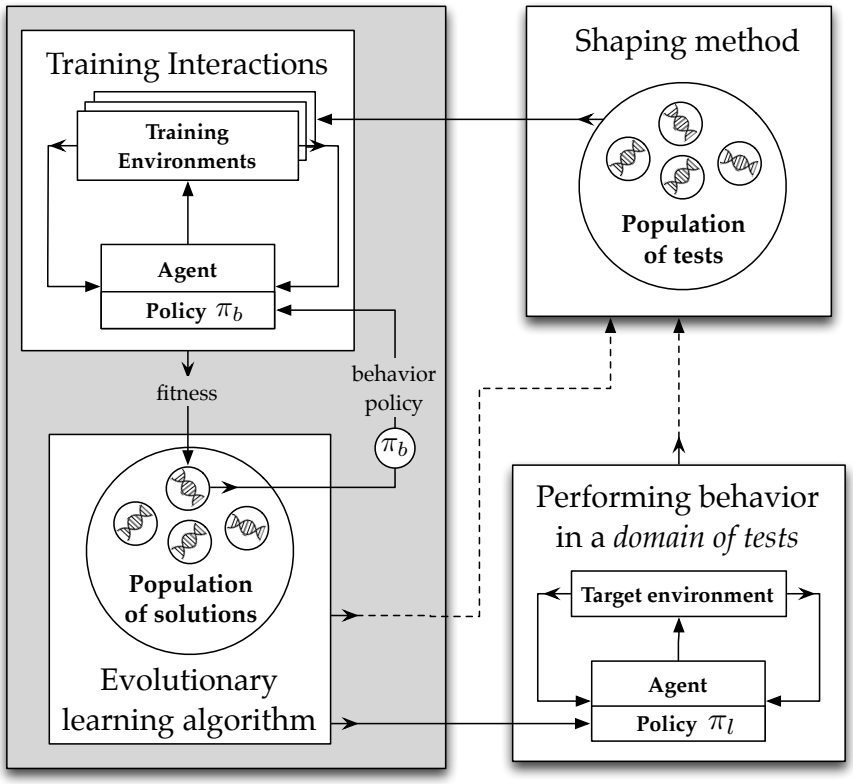


Figure 4.4: Two-population coevolution as a shaping method.

drives the entire learning process. However, while the policies are evaluated for their performance within the population of tasks, the tasks should be rather rewarded for informing about the capabilities of evolving policies ("performing is not the same as informing" [24]). In this context, the population of tests plays the role of the *teacher* [19, 53, 98, 99] which ideally should pose tasks that are consistently neither too 'difficult' nor too 'easy', but feature the level of difficulty that provides a tractable learning gradient for the coevolving learners (policies) [210].

Population of tests as a teacher

The role of the teacher (played by the population of tests) can be expressed as the hypothesis that

> *The best way for adaptive agents to learn is to be exposed to problems that are just a little more difficult than those they already know how to solve* (Juillé [97], p. 127).

This belief resembles the concept of the zone of proximal development (see Section 3.1). Also, it pertains to the observation that "you can only learn what you almost already know" [57].

Furthermore, we can draw an analogy between coevolution and shaping. Figure 4.4 illustrates how two-population coevolution naturally fits into the introduced shaping framework. In this context, the population of tests together with its selection and variations mechanisms acts as a shaping method and provides training tasks that

Population of tests as a shaping method

are expected to facilitate the learning process. As already discussed in Section 4.1, although most reinforcement learning scenarios focus on developing the optimal policy for a single target environment, it can be beneficial to consider a set of related training environments that can provide an easier path to learning. Coevolution is a nature-inspired method of searching such a space of training environments and adaptively constructing a useful learning gradient for the population of learners.

Similar analogies between shaping (or staged/incremental learning) have been considered since the early works on coevolution:

> *The success of a machine learning system depends very much on the learning environment in which it is placed. After it has extracted all the accessible information from its original environment, it may need to be put in to a new, more challenging, environment in order to progress. 'Curricular' or 'staged' learning occurs when a learner is placed in to a pre-designed series of environments one after the other, as it progresses. However, designing an appropriate series of environments may be very difficult. This difficulty would be avoided if there were some way for the learner and its environment to co-evolve with each other, so that the one would always be appropriate for the other.*
> (Blair and Pollack [19], p. 166)

In particular, it has been emphasized that coevolution is capable of realizing shaping-like process automatically:

> *Shaping . . . requires a human experimenter to design a schedule of changes to the task, whereas coevolution itself can be seen as an automatic shaping method: the fitness depends on the behavior of the opposing population, which is gradually becoming more proficient in a coevolutionary arms-race.*
> (Dziuk and Miikkulainen [49], p. 1078)

This statement closely resonates with this thesis, which revolves around coevolution meant as an 'unsupervised' shaping approach.

5 EXPERIMENTAL DOMAINS

In this chapter we introduce three testbed domains that are used throughout this thesis. Each domain provides a common basis for one or more sequential decision tasks, which can be learned with reinforcement learning methods described in Section 2.2. In the subsequent chapters we report the experiments conducted in these domains which allow us to compare the performance of the proposed shaping methods with that of conventional learning techniques.

Motivation for using board games

The first two domains are board games, which have always been a popular area of machine learning research [65]. In fact, creating a game-playing program capable of beating human masters has been one of the earliest goals of the artificial intelligence field [171]. Furthermore, since the seminal studies of Arthur Lee Samuel [169, 170] and his checkers-playing program, games have been regarded as particularly useful testbeds for developing and evaluating new learning techniques:

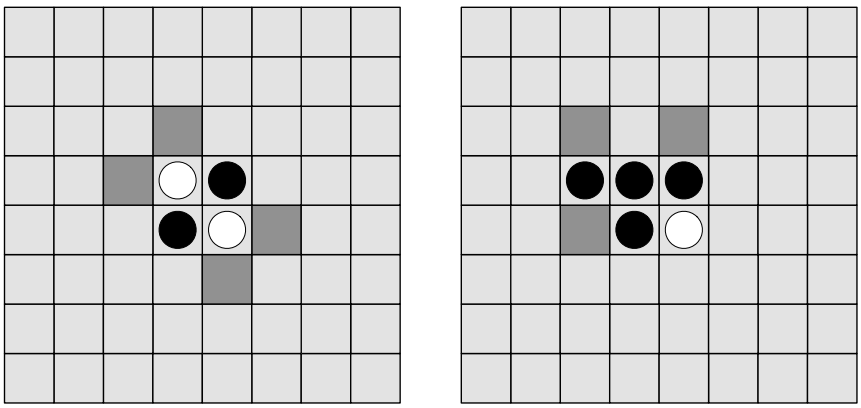
> *For some years the writer has devoted his spare time to the subject of machine learning and has concentrated on the development of learning procedures as applied to games. A game provides a convenient vehicle for such study as contrasted with a problem taken from life, since many of the complications of detail are removed* (Samuel [169], p. 211).

Importantly, Sutton and Barto [189] refer to Samuel's program as to the first example of what can be called a temporal difference learning technique. Following this influential work, a lot of research consisted in applying reinforcement learning methods for elaborating board game policies [67], with the backgammon player called TD-Gammon [198] being the most famous example.

Implemented domains

Most experiments in this thesis concern the board game of Othello which is described in Section 5.1. The second considered game is a simplified version of Go played on a 5×5 board. Besides the games, the third domain discussed in Section 5.3, is a standard reinforcement learning problem of cart pole balancing (also known as the inverted pendulum task).

In the following sections we describe the particular domains in detail. For each of them we discuss the possible ways of representing decision-making policies and present the performance measures employed in the experiments to evaluate the learning results and compare particular learning methods. Moreover, we also provide a brief survey of the previous research conducted in each of the considered domains.



(a) Othello initial board state. Black to move.

(b) Board state after black's move. White to move.

Figure 5.1: Othello boards with legal moves marked as shaded locations.

5.1 OTHELLO

A minute to learn... a lifetime to master is the motto of the game of Othello. Indeed, despite its apparent simplicity, it is one of the most challenging board games with numerous tournaments and regular world championship matches. The exact origin of the game is unknown but rumors say that it arose from an old Chinese game called *Fan Mian* [78]. Contemporarily, Othello was proposed in 19th century by Lewis Waterman, who marketed it under the name *Reversi* [149]. Both names 'Othello' and 'Reversi' are often used interchangeably today. The modern rules of Othello, which are standardized around the world now, have been proposed and popularized in Japan by Goro Hasegawa [80]. The name of the game refers to William Shakespeare's drama "Othello, the Moor of Venice" [175], to illustrate that the game is full of dramatic reversals caused by the rapid changes in dominance on the board.

History of Othello

5.1.1 *Othello Game Rules*

The game of Othello is a deterministic, perfect information, zero-sum board game played by two players on an 8×8 board. Typically, pieces are disks with a white and black face, each face representing one player. Figure 5.1a shows the initial state of the board; each player starts with two pieces in the middle of the grid. The black player moves first, placing a piece, black face up, on one of four shaded locations. Figure 5.1b illustrates the board state resulting from one of possible moves of the black player. Players make moves alternately by placing their pieces on the board until neither of them is able to make a legal move.

Initial game setup

A legal move consists in placing a piece on an empty square and flipping the appropriate pieces. To place a new piece, two conditions must be fulfilled. Firstly, the position of the piece must be adjacent to an opponent's piece. Secondly, the new piece and some other piece of the player must form a vertical, horizontal, or diagonal line with a contiguous sequence of opponent's pieces in between. After placing the piece, all such opponent's pieces are flipped; if multiple lines exist, flipping affects all of them. This makes the game particularly dramatic — a single move may gain the player a large number of pieces and swap players' chances of winning. A legal move requires flipping at least one of the opponent's pieces. Making a move in each turn is mandatory, unless there are no legal moves. The game ends when both players have no legal moves. The objective of the game is to have the majority of pieces on the board at the end of the game. If both players have the same number of disks, the game ends in a draw.

Making legal moves

5.1.2 *Policy Representations*

Since the number of states in the game of Othello is too large to represent the policy directly as a state-action mapping, we need to employ an indirect representation of policy. For this aim, we can approximate either state value function or action value function, as already discussed in Section 2.2.1.2.

Function approximation

In the context of games, state value function plays the role of *position evaluation function* [176] which, given a state (a board game position), returns a scalar value indicating how beneficial the state is for the player. If it is possible to compute the resulting afterstates for each possible action (what we assume here), we can evaluate them with the position evaluation function and choose the most favorable move. Since typically an evaluation function is more accurate near the end of a game, it can be combined with a deeper minimax game tree search to flexibly balance between computation time and evaluation accuracy.

Position evaluation function

Although action value functions have also been used in practice [62], most recent works on learning Othello strategies have focused on state value functions [120, 126], and we follow that trend in this thesis. Moreover, we focus our research on comparison between learning methods rather than developing efficient tree search algorithms or designing new representation of policies. For this reason, to select a move, we evaluate all states at 1-ply — when a player is to make a move, it expands the current game state to all possible direct afterstates and evaluates each of them using the position evaluation function f. The state gauged as the most valuable determines the move to be made. Ties are resolved at random.

1-ply search

Evaluation function architecture	Number of weights	References
Weighted Piece Counter (WPC)	64	[120, 232]
Multilayer Perceptron (MLP)	2 144 (flexible)	[18, 102]
Spatial Neural Network	5 900	[29, 30]
N-Tuple Network	8 748 (flexible)	[117, 126]

Table 5.1: Evaluation function architectures supported by the Othello Position Evaluation Function League.

5.1.2.1 *Popular Position Evaluation Functions and the Othello League*

A good overview of different function architectures used to evaluate Othello positions is provided by Lucas and Runarsson in their Othello Position Evaluation Function League[1]. Table 5.1 shows the architectures acceptable in the league and the number of parameters (weights) employed by each of them. If the number of weights is flexible, like for MLPs or *n*-tuple networks, we refer to the specific settings from the papers listed in the table.

Othello League player architectures

To represent game-playing policies, in our experiments we employ both Weighted Piece Counters (WPCs) and *n*-tuple networks, which are precisely described in Sections 5.1.2.2 and 5.1.2.4, respectively. WPC is the simplest possible architecture, which may be viewed as an artificial neural network comprising a single linear neuron with inputs connected to all 64 board locations. It assigns a single weight to each location and calculates the utility of a given board state by multiplying the weights by color-based values of the pieces occupying corresponding locations. Regarding the league results, all the best players submitted to the competition are based on more complex architectures than WPC, such as *n*-tuple networks. Such functions operate in a highly non-linear fashion and typically involve much larger number of parameters.

Trial league vs. heuristic player

As a side note regarding the Othello League, apart from the competitions held among submitted evaluation functions [118], there is an online trial league[2] based on the score obtained in 100 games played at 1-ply (in which 10% of moves are forced to be random) against the standard WPC heuristic (SWH) player. The weights of the SWH player handcrafted by Yoshioka et al. [232] are illustrated in Table 5.2 and visualized in Fig. 5.2a. We adopt the score obtained against this benchmark player as one of the performance measures used in our experiments to compare policies developed by different learning methods (see Section 5.1.3). Additionally, to make the comparison more informative, we employed also the best players submitted to the league as opponents in round robin tournaments.

1 http://algoval.essex.ac.uk:8080/othello/html/Othello.html
2 http://algoval.essex.ac.uk:8080/othello/League.jsp

1.00	−0.25	0.10	0.05	0.05	0.10	−0.25	1.00
−0.25	−0.25	0.01	0.01	0.01	0.01	−0.25	−0.25
0.10	0.01	0.05	0.02	0.02	0.05	0.01	0.10
0.05	0.01	0.02	0.01	0.01	0.02	0.01	0.05
0.05	0.01	0.02	0.01	0.01	0.02	0.01	0.05
0.10	0.01	0.05	0.02	0.02	0.05	0.01	0.10
−0.25	−0.25	0.01	0.01	0.01	0.01	−0.25	−0.25
1.00	−0.25	0.10	0.05	0.05	0.10	−0.25	1.00

Table 5.2: The weight matrix of the SWH player.

5.1.2.2 *The Weighted Piece Counter Architecture*

The Weighted Piece Counter (WPC) architecture relies on a heuristic assumption that to judge the utility of a particular board state it is enough to *independently* consider the occupancy of every board location. For this reason, WPC assigns a weight w_i to each board location i and uses scalar product to calculate the value of position evaluation function f for a given board state $\mathbf{b}$:

WPC operation

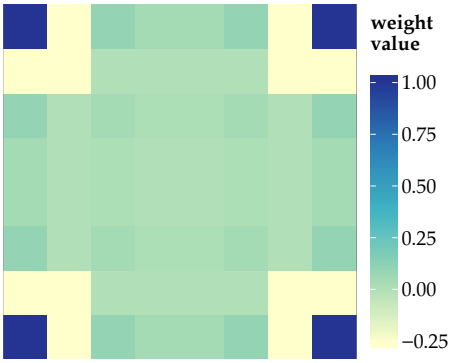
$$f(\mathbf{b}) = \sum_{i=1}^{8\times 8} w_i b_i, \tag{5.1}$$

where b_i is $+1$, -1, or 0 if, respectively, location i is occupied by a black piece, white piece, or empty. The players interpret the values of f in a complementary manner: the black player prefers moves leading to states with larger values while the smaller values are favored by the white player.

The main advantage of WPC is its simplicity, resulting in very fast board evaluation. Moreover, a policy represented by a WPC can be easily interpreted just by inspecting the weight values. For instance, Table 5.2 presents the weight matrix of the SWH player which clearly focuses at taking possession of the corners because they feature the highest values. Additionally, WPC weights can be also presented in a more illustrative way by means of a weight-proportional coloring as in Fig. 5.2a (darker squares denote larger weights, i.e., more desirable locations on the board).

Advantages of WPC

The handcrafted SWH weights (Fig. 5.2a) illustrate the importance of the corners of the Othello board. These are the only squares that once occupied by a player, can never be captured by its opponent. While the corners are the most desirable, their immediate neighbors have very low weights. Such configuration indicates that placing disks on neighboring locations should be avoided to prevent the opponent from capturing a corner. This feature of the game is confirmed in practice: placing a stone in a corner can gain a player a large number of points and revert the previously anticipated game outcome.



(a) Othello board colored proportionally to corresponding SWH player weights.

A	B	C	D	D	C	B	A
B	E	F	G	G	F	E	B
C	F	H	I	I	H	F	C
D	G	I	J	J	I	G	D
D	G	I	J	J	I	G	D
C	F	H	I	I	H	F	C
B	E	F	G	G	F	E	B
A	B	C	D	D	C	B	A

(b) Othello board symmetries. Squares that are equivalent under symmetry are marked with the same letter.

Figure 5.2: Othello board illustrations that indicate SWH player weights and board symmetries.

5.1.2.3 *The Shared Weighted Piece Counter Architecture*

Othello board symmetries

Although the WPC architecture contains 64 independent weights associated with particular board locations, some weights play very similar role due to symmetries in Othello board. For instance, the weights of the handcrafted SWH player illustrated in Fig. 5.2a are symmetric under reflection and rotation. The Othello board symmetries are visualized in Fig. 5.2b which illustrates 10 types of locations that play unique roles in the Othello board. Squares marked with the same letter can be seen equivalent if symmetries are taken into account.

Sharing WPC weights

Lucas [117] reported that enforcing symmetry of WPC weights and thus reducing the number of parameters to only 10 weights can increase the learning speed. Sharing weights across symmetric locations have been found successful also in other works [207]. We implement this idea and term the WPC with symmetric weight matrix Shared Weighted Piece Counter (SWPC). Symmetries of the Othello board are exploited also by the n-tuple networks described below.

5.1.2.4 *The N-tuple Network Architecture*

Origins of n-tuple networks

The idea of n-tuple networks was originated by Bledsoe and Browning [20] for character recognition. Since then it has been successfully applied to both classification [161] and function approximation tasks [104]. Their main advantages include conceptual simplicity, speed of operation, and capability of realizing non-linear mappings of combinatorial characteristics. Following significant research on using n-tuple classifiers for hand-written digits [119] and face recognition problems [116], Lucas proposed employing the n-tuple architecture also for game-playing purposes [117].

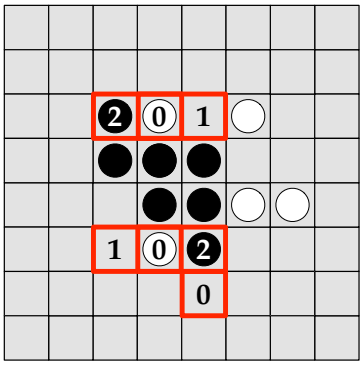


Figure 5.3: Two sample n-tuples superimposed on the Othello board. Each input location represents a ternary digit. Multiplying them by successive powers of 3 leads to decimal values of $2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0 = 19$ and $1 \cdot 3^3 + 0 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0 = 33$, which are used as indexes in the associated look-up tables.

Network operation

An n-tuple network expects as input some compound entity (matrix, tensor, image) $\mathbf{x}$, which elements (usually scalar variables) can be retrieved using some form of coordinates. An n-tuple network operates by sampling that input object with m n-tuples. An n-tuple $t_i, i = 1, \ldots, m$, is a sequence of n variables $a_{ij}, j = 0 \ldots n-1$, each corresponding to predetermined coordinates in the input. Assuming that each variable in $\mathbf{x}$ takes on one of v possible values, an n-tuple can be viewed as a template for an n-digit number in base-v numeral system. When a specific input $\mathbf{x}$ is given, it assumes one of v^n possible values. The number represented by the n-tuple t_i is used as an index in an associated look-up table LUT_i, which contains parameters analogous to weights in standard neural networks. For a given input $\mathbf{x}$, the output of the n-tuple network can be calculated as:

$$f(\mathbf{x}) = \sum_{i=0}^{m-1} f_i(\mathbf{x}) = \sum_{i=0}^{m-1} LUT_i \left[\sum_{j=0}^{n-1} \mathbf{x}(a_{ij}) v^j \right], \tag{5.2}$$

where $\mathbf{x}(a_{ij})$ denotes the element located at position a_{ij} in $\mathbf{x}$.

N-tuple network for Othello

In the context of Othello, an n-tuple network acts as a state evaluation function. It takes a board state as an input $\mathbf{x}$ and returns its utility. Input variables are identified with coordinates on the board, and the value retrieved from a single location is 0, 2, or 1 if, respectively, it is occupied by a white piece, black piece, or is empty. Consequently, an n-tuple represents a ternary number which is used as an index for the associated look-up table containing 3^n entries (see Fig. 5.3). Additionally, board symmetries are incorporated (though not shown in this figure) — a single n-tuple is employed eight times, once for each possible board reflection and rotation. LUT values indexed by all such equivalents are summed together to form the output of the particular n-tuple. The final value of a board is simply the sum of all n-tuple outputs (Equation 5.2).

Reward scheme	Reward value		
	win	draw	loss
Three points for a win [44]	3	1	0
Othello League scoring [118]	1	0.5	0

Table 5.3: Reward schemes employed in Othello tasks.

Input assignment

The number of possible n-tuple instances is exponential in function of the size of $\mathbf{x}$ and n, so assigning the initial input variables (board locations) to n-tuples is an important design choice. Typically, in pattern recognition the simplest approach of random selection of coordinates is commonly used. However, in the context of games, the spatial neighborhood of chosen locations is intuitively relevant. For this reason, and particularly for Othello, connected sets of locations like a straight line or a rectangle area are usually chosen. In our implementation we allowed for more flexible assignments in the form of *snake* shapes, proposed by Lucas [117]. For each n-tuple, we choose a random square on the board from which a random walk of $n-1$ steps in any of the maximum eight possible directions is taken.

5.1.3 *Performance Measures*

Othello game as an MDP task

To monitor the progress of learning in the Othello domain, we employ several performance measures for evaluating policies. Given a policy π, each measure is based on calculating the expected cumulative reward, i.e., the policy return $J(\pi)$, in one or more MDP tasks. In these tasks, the state space contains all possible Othello board positions while the actions are legal moves. Reward is granted only at the end of the game and is determined by the game outcome. Specific values of rewards are defined by the reward schemes presented in Table 5.3.

In the performance measures described below, each game-playing task is posed by a certain opponent which is treated as a part of the environment as it determines the transition function. Importantly, when a policy is evaluated in a given task it can be used either by a black or white player. Technically, we simulate the same number of games for both these scenarios.

5.1.3.1 *Standard WPC Heuristic Player*

To assess how well a policy copes with a moderately strong opponent, we evaluate it in a single task posed by the standard WPC heuristic (SWH) player. This human-designed opponent is used in the online Othello League (see Section 5.1.2.1) as well as in many previous works on Othello [18, 207, 120, 168]. The weights of the WPC policy used by this player are shown in Table 5.2 and in Fig. 5.2a.

Since the game itself is deterministic, we force both players to make random moves with probability $\epsilon = 0.1$ to diversify their behaviors and we use the average policy return in a number of games as a performance measure. Following Lucas and Runarsson [120], we assume that the ability of playing such a randomized game is highly correlated with the ability of playing the original Othello.

Move randomization

5.1.3.2 *Round-Robin Tournament*

A handcrafted heuristic policy like SWH, even when randomized, cannot be expected to represent in full the richness of possible opponent policies. In order to compare different learning methods in a wider context, in some of the experiments conducted in this thesis we employ a *relative* performance measure based on a round-robin tournament between the policies developed by particular methods. For this purpose we recruit teams of diverse opponents composed of the policies representing these methods and play a tournament, where each team member is confronted with all members from the opponent teams. The final performance of a team (representing a learning method) is determined as the sum of rewards obtained by its policies in the tournament.

Relative performance measure

Let us notice that the round-robin tournament offers yet another advantage: there is no need to randomize moves (as it was the case when playing against a single external player), since the presence of multiple policies in the opponent team naturally provides behavioral variability.

5.1.3.3 *Othello League Contestants*

Thanks to the courtesy of the Othello League organizers (see Section 5.1.2.1), we were provided with the strategies submitted to the league by anonymous contestants. From the several hundreds submitted to the league, we selected the top 14 strategies to form a pool of opponents. To evaluate a given policy we compute its cumulative return in games with every opponent in the pool. It is worth noticing that this performance measure, by being based on highly skilled players, should be considered as the most demanding one in comparison to the remaining measures.

External pool of opponents

5.1.3.4 *Expected Utility*

A fully objective assessment of the policy would consist in playing against all possible Othello opponents. However, the huge number of opponent policies makes the explicit computation of such *expected utility* measure impossible. We can only estimate it by calculating the score obtained in a limited number of games against randomly generated WPC policies. Clearly, this measure evaluates how well a policy generalized across the entire domain of Othello tasks.

Generalization performance

The expected utility measure corresponds to the *maximization of expected utility solution concept* [56] in co-optimization (cf. Section 4.2.2) and is also referred to as *generalization performance* [31, 33].

5.1.4 *Previous Research on Computer Othello*

The game of Othello has been a subject of artificial intelligence research for more than 20 years. The significant interest in this game may be explained by its simple rules, large state space cardinality (around 10^{28}) and high divergence rate[3]. All these reasons combined cause Othello to stay *unsolved* — a perfect Othello player has not been created yet (as opposed to, e.g., checkers). Thus, Othello remains an excellent benchmark for learning algorithms and player architectures.

Traditional approach

Conventional Othello-playing programs are based on thorough human analysis of the game, materialized in sophisticated handcrafted evaluation functions. They often incorporate supervised learning techniques that use large expert-labeled game databases and efficient look-ahead game tree search. One of the first examples representing such approach was BILL [112]. Besides using pre-computed tables of board patterns, it employed Bayesian learning to build in certain features into an evaluation function.

Logistello

Today, one of the most known Othello programs is Logistello [27], which makes use of advanced search techniques and applies several methods to learn from previous games. Its evaluation function is based on a pre-defined pattern set including horizontal, vertical and diagonal lines as well as special patterns covering edges and corners of the board. Pattern configurations correspond to binary features and have associated values. Evaluating a board state consists in summing up the values of features occurring in it, and thus, is practically equivalent to calculating the value of an n-tuple network.

Recent trends

Recently, the mainstream research on Othello has moved towards better understanding of what types of learning algorithms and player architectures work best. The series of CEC Othello Competitions [118] pursued this direction by limiting the ply depth to one, effectively disqualifying the algorithms that employ a brute-force game tree search.

Knowledge-free approach

The most challenging scenario of elaborating a game-playing policy is learning without any support of human knowledge or expert opponents known in advance. This task formulation is addressed by, among others, self-play temporal difference learning and coevolutionary learning, which were applied to Othello by Lucas and Runarsson [120]. Other examples of using self-learning approaches for Othello include coevolution of spatially aware MLPs [30], TD-leaf learning of structured neural networks [207], and Nash Memory applied for coevolved n-tuple networks [126].

3 Divergence rate describes the average difference between board positions that can be reached with a single move from a given state [67].

5.2 SMALL-BOARD GO

The game of Go is believed to have originated about 4000 years ago in Central Asia, which makes it one of the oldest known board games. Although the game itself is very difficult to master, its rules are relatively simple and comprehensible. For this reason the famous chess player, Edward Lasker [110], summarized Go in the following way:

> *While the Baroque rules of Chess could only have been created by humans, the rules of Go are so elegant, organic, and rigorously logical that if intelligent life forms exist elsewhere in the universe, they almost certainly play Go.*

5.2.1 *Original Game Rules*

Go is played by two players, black and white, typically on an 19×19 board. Players make moves alternately, blacks first, by placing their stones on unoccupied intersections of the grid formed by the board. At any time the player who is about to move may pass his turn. The game ends if both players pass consecutively.

Game objective

In a broad sense, the objective of the game is to control more territory than the opponent at the end of the game. This can be achieved by forming connected stone *groups* enclosing as many vacant points and opponent's stones as possible. A stone group is a set of stones of the same color adjacent to each other; empty intersections adjacent to a group constitute its *liberties*. When a group loses its last liberty, i.e., becomes completely surrounded by opponent's stones or edges of the board, then it is captured and removed.

Legal moves

A legal move consists in placing a piece on an empty intersection and capturing enemy groups which are left without liberties (if any). Additional restrictions on making moves concern *suicides* and so called *ko rule*. A suicide is a potential move that would reduce the number of liberties of player's own group to zero. Moves leading to suicides are illegal. Ko rule states that a move that recreates a previous board state (i.e., the arrangement of stones on the board) is not allowed either.

Scoring systems

The winner is the player who scores more points at the end of the game. The scores are determined using a scoring system agreed upon before the game; the two popular systems include *area counting* (Chinese) and *territory counting* (Japanese). Both ways of calculating the score of a player involves the number of empty intersections surrounded by the player (player's *territory*). This figure is augmented by the number of player's stones on the board in the area counting system, or by the number of captured stones (*prisoners*) in the territory counting system. In this thesis we use the Chinese scoring scheme with no *komi* (i.e., no points given in advance to one of the players).

The reader interested in more detailed description of Go rules is referred to the book by Bozulich [23].

5.2.2 *Adopted Computer Go Rules*

There are a few noteworthy issues about the rules of Go that make developing computer players particularly difficult. For this reason, we restrict our research a simplified version of Go, described in following and well adopted in the computational intelligence community.

Small-board Go

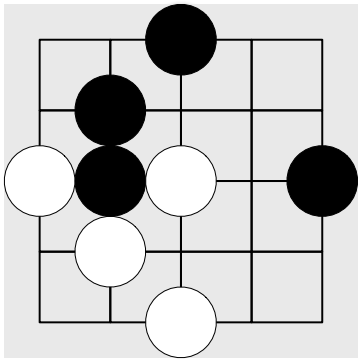
First of all, the immense cardinality of the state space and the large branching factor render full-board Go intractable for many algorithms. Fortunately, the game rules are flexible enough to be easily adapted to smaller boards without loss of the underlying 'spirit' of the game, so in a great part of studies on computer Go the board is downgraded to 9×9 or 5×5. Following Runarsson and Lucas [164] and Lubberts and Miikkulainen [115], we consider playing Go on a 5×5 board (see Fig. 5.4).

Controlled intersections

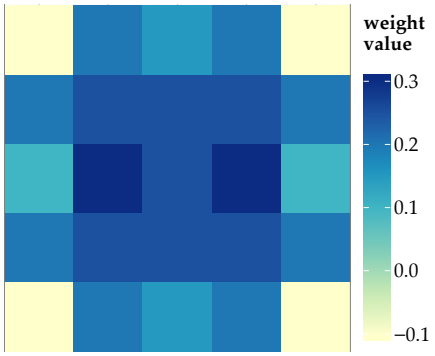
A more subtle difficulty in adopting Go rules to computer programs concerns the fact that game termination criterion is in a sense subjective here, i.e., human players end a game after they agree that they can gain no further advantages. In such situations they use a substantial amount of knowledge to recognize particular intersections as *implicitly controlled*. According to the game rules, if it is impossible to prevent a group from being captured, it is not necessary to capture it explicitly in order to gain its territory. Such a group is considered as *dead* and it is removed at the end of the game when both players decide which groups would inevitably be captured. Because determining which stones are dead is nontrivial for computer Go players, we assume that all groups on the board are alive and that capturing is the only way to remove an opponent's group. As a consequence, games are usually continued until all intersections are *explicitly controlled* and, thus, are much longer than those played by humans.

Super-ko rule

Finally, in some Go rule sets (including the Chinese rules that we employ), the *ko* rule is superseded by *super-ko* that forbids repetition of board states during a game. Recurrently appearing states imply cycling and, theoretically, an infinite game. However, strict implementation of super-ko requires storing all previous board configurations and comparing each of them to the current state. Since most of possible cycles are not longer than 3, we use a reasonable approach in which we remember just two previous board configurations. However, longer cycles can still occur, so to ensure that the game ends, an upper limit of 125 on the total number of moves is additionally imposed. Exceeding this limit results in declaring game's result as a draw.



(a) An example of game position.



(b) Go board colored proportionally to heuristic WPC weights.

Figure 5.4: Small-board version of Go.

5.2.3 *Policy Representations*

Due to huge number of possible small-board states, Go policies, similarly to Othello, can not be represented explicitly. For this reason, we employ weighted piece counters (WPCs, see Section 5.1.2.2) to implement the position evaluation function f. Using WPCs implies the simplest form of board representation (known as *Koten* [128]), in which exactly one input for each intersection is provided to the evaluation function f. Although such an input signal does not carry information about neighboring intersections, which seems to be essential in Go, it has been frequently used in related studies [164, 172].

WPC and direct board representation

The fact that WPC weighs occupancy of each board intersection independently makes it probably the least sophisticated policy representation for the game of Go. One can discuss whether WPC is appropriate for the game that exhibits such strong spatial coherence [177]. In objective terms therefore, we do not anticipate the policies represented in this way to beat the top-ranked computer players. However, in the context of this thesis this should not be perceived as a hindrance, as our primary goal is to investigate the performance of learning and shaping methods. Still, we postulate that at least some of the conclusions can be generalized to other, more sophisticated policy representations.

Lack of spatial context

On the other hand, WPC's simplicity and positional character bring substantial advantages, fast board evaluation being the most prominent one. As in Othello, WPC policies can be also easily visualized by coloring board locations according to corresponding weights, as illustrated in Fig. 5.4b (darker squares denote larger weights, i.e., more desirable locations on the board). The figure presents a hand-crafted WPC policy based on a best player found by Runarsson and Lucas [164]. Table 5.4 presents the weight matrix of this player, which clearly aims at occupying the center of the board while avoiding the corners.

Hand-crafted WPC policy

−0.10	0.20	0.15	0.20	−0.10
0.20	0.25	0.25	0.25	0.20
0.10	0.30	0.25	0.30	0.10
0.20	0.25	0.25	0.25	0.20
−0.10	0.20	0.15	0.20	−0.10

Table 5.4: The weight matrix of the heuristics player.

5.2.4 *Performance Measures*

To evaluate a policy for the game of small-board Go, we rely on similar performance measures to those used for Othello (cf. Section 5.1.3). In particular, we employ the round-robin tournament and two human-designed opponents described below.

5.2.4.1 *Heuristic WPC Player*

Analogously to the SWH player used in the Othello domain (see Section 5.1.3.1), we hand-crafted a heuristic player based on the WPC policy representation to evaluate policies for the game of small-board Go. Table 5.4 illustrates the weight matrix of this player, which is based on the strategy developed by Runarsson and Lucas [164].

All WPC-based policies are deterministic and so is the game of Go. Thus, in order to estimate player's probability of winning against the WPC heuristic player, we force both players to make random moves with probability $\epsilon = 0.1$; this allows us to take into account a richer repertoire of players' behaviors and make the resulting estimates more continuous and robust.

5.2.4.2 *Liberty Player*

To provide another, qualitatively different benchmark for the developed methods we created a simple game-specific heuristic strategy based on the concept of liberties (described in Section 5.2.1). This strategy, called here *Liberty Player* looks 1-ply ahead and evaluates a position by subtracting the number of opponent liberties from the number of its own liberties. Ties are resolved randomly. As with the heuristic WPC player, both players are forced to make random moves with probability $\epsilon = 0.1$.

5.2.5 *Previous Research on Computer Go*

The game of Go has been a subject of computational intelligence research for more than 40 years and it is increasingly recognized as a great challenge because the best computer players can still be beaten by professional human players in full-board games [96, 129]. This is

a result of a huge combinatorial complexity of this game, which is much higher than for other popular two-player deterministic board games. The state space cardinality in Go is around 10^{170} and the game tree has an average branching factor of around 200. For this reason a lot of research on computer Go focuses on the versions with smaller boards, like 9×9 or even 5×5.

Hand-coded Go programs

Many Go-playing programs are precisely tuned expert systems based on a thorough human analysis of the game. In such programs, knowledge of professional Go players formulated as a multitude of rules and guidelines is implemented in a game-playing algorithm in order to recognize particular board patterns and react to them. However, this knowledge-based approach is constrained by the extent and quality of the available knowledge and the possibility of its articulation in the source code of the playing program.

Monte Carlo techniques

An appealing alternative to using hand-coded expert rules is an approach based on Monte Carlo Tree Search techniques [22], which by contrast, requires very little domain knowledge. This method chooses a move on the basis of statistics collected during thousands of random playouts starting from the current board state. One of the strongest Go programs using this approach is *Fuego* [52], the champion of 2009 Computer Olympiad in 9×9 Go, which has recently defeated 9-dan professional player on the same board size. Another major program, *Many Faces of Go* [64], champion of 2008, uses an interesting combination of Monte Carlo Tree Search with an older knowledge-based approach.

Self-learning approach

Nevertheless, in Go as in Othello, the approach that is most inspiring from the AI perspective is learning policies without any reference to human knowledge or game strategy given *a priori*. This approach is represented by, e.g., self-play temporal difference learning and co-evolutionary learning, which were investigated and compared in the context of small-board Go by Runarsson and Lucas [164]. There are more examples of using self-learning approaches for Go including the studies by Lubberts and Miikkulainen [115] and Schraudolph et al. [172]. A comprehensive review of all AI methods applied to computer Go can be found in a survey by Bouzy and Cazenave [21].

5.3 CART POLE BALANCING

Alternative names

Cart pole balancing is a standard benchmark problem in the field of control theory and for over 50 years has been widely used to demonstrate the performance of machine learning methods [4, 66]. The problem is also known by the name of *inverted pendulum problem* [3], while in the early work of Widrow and Smith [222] it is originally described as a *broom-balancing machine*:

> *The dynamic system is a motorized cart carrying an inverted pendulum. The controller for the system is required to keep the pendulum balanced and keep the cart within certain bounds by applying a horizontal force to the cart. . . . This is similar to a person trying to balance a broom on his finger.*
> (Widrow and Smith [222], p. 312)

Reasons of popularity

There are several reasons that explain the popularity of this problem in reinforcement learning community. Apart from being easy to understand and visualize (see Fig. 5.5), it involves the hallmark challenge of temporal credit assignment — the feedback signal occurs only after a long sequence of actions, when the pole falls or the cart hits the track boundary. Moreover, as noted by Wieland [223]:

> *The problem is of interest because it describes an inherently unstable system and is representative of a wide class of problems.*
> (Wieland [223], p. 667)

For instance, the inverted pendulum problem is related to rocket and missile guidance [74]. Besides, it was also used as a testbed for learning strategies for satellite attitude control [166].

Problem variations

Finally, although the standard version of the problem is relatively easy and can be solved with a single linear neuron [222] and a random search in its weight space [66, 75], there are many extensions of the problem that make it more challenging, even for modern learning methods. The most commonly used variations include adding a second pole, using a jointed pole [223], or restricting the amount of available state information. The last variation makes the problem non-Markovian and requires to employ short term memory [73].

5.3.1 *Physical Model*

The canonical version of the single pole balancing problem, which is described in the influential studies of Michie and Chambers [131] and Barto et al. [13], is schematically illustrated in Fig. 5.5. A rigid pole (pendulum) of length $2l$ and mass m is mounted on a wheeled cart of mass M. The movement of the cart is restricted to one-dimensional bounded track and the pole can only swing in a vertical plane defined by the track. The objective is to balance the pole by exerting horizontal

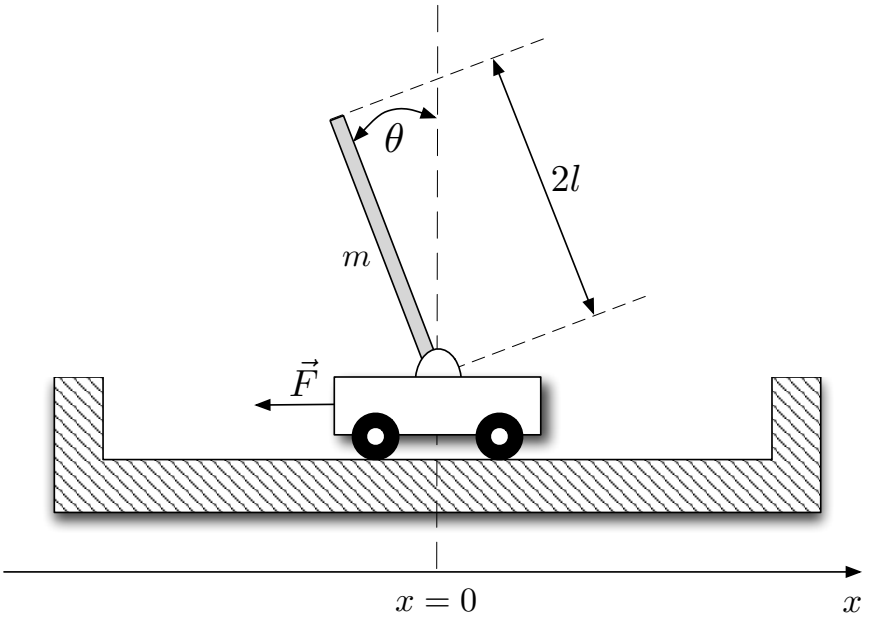


Figure 5.5: Single pole balancing problem.

force F on the cart, either from the left or from the right side. The balancing attempt fails when the pendulum inclination θ exceeds the given limit or when the cart reaches the track boundaries.

The state of the system is described by four real-valued variables:

- x — the position of the cart measured as the distance from the center of the track,
- $\dot{x}$ — the velocity of the cart,
- θ — the angle of the pole,
- $\dot{\theta}$— the angular velocity of the pole.

The dynamics of the system is modeled by the following differential equations [223]:

$$\ddot{x} = \frac{F + ml\dot{\theta}^2 \sin\theta + \frac{3}{4}m\cos\theta\left(\frac{\mu_p\dot{\theta}}{ml} + g\sin\theta\right) - \mu_c\,\mathrm{sgn}(\dot{x})}{M + m\left(1 - \frac{3}{4}\cos^2\theta\right)} \tag{5.3}$$

$$\ddot{\theta} = -\frac{3}{4l}\left(\ddot{x}\cos\theta + g\sin\theta + \frac{\mu_p\dot{\theta}}{ml}\right), \tag{5.4}$$

where g denotes the gravitational acceleration, μ_c is the coefficient of friction between the cart and the track and μ_p is the friction coefficient of the hinge between the cart and the pole.

5.3.2 *Pole Balancing as an MDP Task*

States, actions and transitions

Controlling the cart can be naturally formulated as an undiscounted episodic MDP task, in which states are represented by tuples of the form $(x, \dot{x}, \theta, \dot{\theta})$, actions are the forces applied to the cart, and the transition function is defined by the motion equations (5.3 and 5.4). In the initial state, the cart is placed in the centre of the track, i.e. $x = 0$, and the pole inclination θ is set to nonzero value θ_0, so the system cannot be balanced just by applying no force at all.

Reward scheme

At each discrete time step t (every T seconds of the simulated time), the control policy must provide the force as a signed real value from a certain range. A positive reward equal to 1 is awarded for each time step before failure. Consequently, the return of the policy (i.e. the total reward) is equal to the number of steps for which the pole was kept in balance. We adopt this reward definition throughout this thesis; an alternative reward scheme consists in providing only a single reward equal to -1 directly after a failure [13].

Continuous space

In contrast to game-playing domains, in pole balancing tasks both the state space and the action space are continuous. For such tasks, artificial neural networks (ANNs) are particularly well suited to represent controller policies. Moreover, in many previous studies ANNs have been successfully applied to different variants of the pole balancing problem [3, 46, 75, 90]. In our experiments we rely on MLPs (see Section 2.2.1.1) with four inputs corresponding to state variables and a single output specifying the amount of force used to push the cart. Typically, the state variables are scaled to the same range before being input to the network.

Standard parameter setting

The most common choices of parameters and limits for the canonical variant of the pole balancing task [13] are presented in Table 5.5. Note that, to provide more realistic environment, the force is clamped, so an agent cannot push the cart arbitrarily strongly. Moreover, the pole's inclination is also limited, because falling outside the given interval is considered as a failure.

5.3.3 *Performance Measure*

Since the fully observable single pole balancing task can be easily solved by a single neuron and random weight guessing [71], it can not be used to reliably evaluate and compare capabilities of learning methods. To make the problem more challenging, in our experiments we attempt to learn a policy that is able to generalize across many tasks from the pole balancing domain.

Expected utility

Consequently, to evaluate the policy we simulate a series of runs in many tasks and calculate the average policy return which is an estimate of its expected utility. A similar performance measure was employed by Whitley et al. [221], who measured the generality of pole

Symbol	Description	Value
x	position of the cart on the track	$[-2.4\,m, 2.4\,m]$
θ	inclination of the pole	$[-12°, 12°]$
F	force applied to the cart	$[-10\,N, 10\,N]$
l	half length of the pole	$0.5\,m$
m	mass of the pole	$0.1\,kg$
M	mass of the cart	$1.0\,kg$
μ_c	track friction coefficient	0.0005
μ_p	pole's hinge friction coefficient	0.000002
θ_0	initial pole inclination	1°
T	simulation time step	$0.02\,s$
g	gravitational acceleration	$-9.81\ m/s^2$

Table 5.5: Standard parameter setting of cart pole balancing problem.

balancing controllers by using a fixed set of 625 initial states. Here, instead of modifying the initial state, we consider tasks with different physical properties such as the length of the pole or the mass of the cart. Importantly, these parameters affect the transition function of the original MDP task. Detailed experimental settings concerning the multi-task pole balancing domain are discussed in Chapter 7.

Simulation details

Each simulated run starts from the state $(0, 0, 1°, 0)$ and the simulation ends when balancing fails or after predetermined number of time steps. The dynamics of the environment is implemented using the fourth order Runge-Kutta integration [48] with a step size of $0.01\,s$. The implementation of the environment is based on the freely available source code by Faustino Gomez [71].

5.3.4 *Previous Research on Pole Balancing*

Many learning algorithms have been applied to the pole balancing problem and its variations. In one of the earliest works in this domain, Widrow and Smith [222] trained a single linear neuron using supervised learning. For this purpose, they required prior knowledge of the correct control policy so they computed it by linearizing the dynamics of the system and applying conventional control theory methods. On this basis they were able to calculate the error signal at each time step and teach the neuron accordingly.

Supervised learning approach

Early reinforcement learning approach

The pole balancing task described by Widrow and Smith [222] was later adapted by Michie and Chambers [131]. They proposed the algorithm called BOXES which can be regarded as one of the first examples of reinforcement learning methods. The algorithm relied on discretizing the space of states into regions called "boxes". For

each box, the algorithm maintained an action indicating the force to be applied when the given state was observed. The idea of state space quantization was reused by Barto et al. [13], who proposed an adaptive critic element (ACE) to train the neural network controller. Importantly, all these early studies employed the so called "bang-bang" control, in which only two actions are possible — the forces of full magnitude in either direction, e.g. either $10\,N$ or $-10\,N$.

Single agent and evolutionary methods

According to Gomez et al. [75], the recent works on pole balancing problem can be divided into those that rely on a single agent (value function based methods) and those relying on evolutionary (direct policy search) techniques. The authors provide also empirical comparisons of these approaches and show that evolutionary methods are generally more efficient, in particular for the more complex variants of the problem. This is one of the reasons why the single agent methods have not been very popular for solving this task. Let us just mention that this group of methods include temporal difference learning (see Section 2.2.2) such as *Q-learning*, which was applied to pole balancing by Lin and Mitchell [114].

Neuroevolution

The evolutionary approach, on the other hand, have been commonly applied especially in combination with neural networks. The conventional neuroevolutionary approach (where each individual represents a complete network) was employed by Wieland [223], who considered different variants of the problem including a jointed pole and two poles of different lengths. The author employed recurrent neural networks to solve both Markovian (with velocities given as the part of the state description) and non-Markovian (with partially observable state) versions of the problems.

Cooperative coevolution

In the context of pole balancing, the conventional neuroevolution was outperformed by cooperative coevolutionary approach proposed by Moriarty and Miikkulainen [137]. Their method called Symbiotic Adaptive Neuro-Evolution (SANE) evolves single neurons instead of complete networks. The fitness of a neuron depends on the performance of the network in which it participates. This work was extended by Gomez and Miikkulainen [73], who proposed Enforced Sub-populations (ESP) which allowed to evolve recurrent neural networks. As a result, the method could be applied to solve the non-Markovian variant of the double pole balancing task. Interestingly, the authors observed that the hardest variant of the task is the one with two poles of similar length. For this reason, they attempted to facilitate learning by starting from the relatively easier tasks and gradually increasing the length of the shorter pole. Such incremental approach can be regarded as a form of supervised shaping.

CMA-ES

An alternative neuroevolutionary method was applied by Igel [90], who evolved neural network weights by CMA-ES, an efficient variant of evolution strategies with self-adaptation of mutation distribution [77]. The author considered a few pole balancing scenarios with one

or two poles and in both Markovian and non-Markovian versions. When compared to other mentioned methods [75], this approach was one of the most efficient.

6

COEVOLUTIONARY TEMPORAL DIFFERENCE LEARNING

The previous chapter provided a description of experimental domains including two board games: Othello and small-board Go. This chapter demonstrates how to learn to play these games with reinforcement learning methods. In particular, we employ single-population coevolution and self-play temporal difference learning, which both can be seen as forms of shaping. Additionally, we introduce *coevolutionary temporal difference learning*, a hybrid method that combines elements of gradient-descent learning and population-based search.

Section 6.2 describes how to apply these methods to learn game-playing policies represented by function approximators. Specifically, we investigate empirical results of learning n-tuple networks for Othello (Section 6.3) and weighted piece counters for small-board Go (Section 6.4). We employ several performance measures to compare both the effectiveness of learning methods and their scalability with the size of policy representation. The main finding is that hybridization of the learning techniques improves the final performance and allows to cope with growing dimensionality of the search space.

6.1 INTRODUCTION

Most board games inherently involve sequential decision making and thus constitute natural test-beds for reinforcement learning methods. One approach to learning game-playing policies is temporal difference learning (TDL, 2.2.2) which has become particularly popular after the influential work of Tesauro [197] and the success of his TD-Gammon player. Importantly, TD-Gammon was able to learn to play Backgammon at expert level solely by playing against itself. By using such self-play training paradigm, it did not require any human supervision or expert strategies given *a priori*.

Self-play TDL

An alternative approach to autonomously elaborating game policies is single-population coevolutionary learning (CEL). The most promising examples of this approach include successfully learning to play Checkers [29, 63] and Backgammon [19]. Although CEL follows the idea of self-teaching and breeds a population of policies by letting them play against each other, the learning process is diametrically different than that realized by gradient-descent TDL. In particular, CEL does not exploit all the training experience available from training games — it uses only the final game outcome (reward), while ignoring the entire course of interactions.

Single-population CEL

Samuel's legacy

Despite significant differences, these two learning methods can be seen as conceptually derived from the same classic work of Samuel [169] on the checkers playing program. The program was trained by playing against a stable copy of itself and thus it has been considered as the world's first example of self-play training paradigm. The learning procedure employed by Samuel is often referred to as an early precursor of temporal difference learning:

> *Samuel was one of the first to make effective use of heuristic search methods and of what we would now call temporal difference learning.* (Sutton and Barto [189], p. 267)

On the other hand, Bucci [24] argues that Samuel's learning algorithm can be also framed as a specific form of two-population coevolution:

> *To elaborate the analogy with evolutionary computation, Samuel's procedure can be called a coevolutionary algorithm with two populations of size 1, asynchronous population updates, and domain-specific, deterministic variation operators.*
> (Bucci [24], p. 2)

Shaping perspective

From another perspective, both TDL and CEL can be regarded as forms of shaping, although they do not fit exactly into our general shaping framework (cf. Section 4.1). In fact, they are rather implicit forms of shaping without separate mechanism responsible for providing training environments (such as, e.g., a second coevolving population). Essentially however, they both conduct learning in a dynamic environment which is expected to get more challenging while the competence of learners increases. The analogies between coevolution and shaping have been already discussed in Section 4.2.3. It's worth pointing out, howeveer, that self-play TDL has been also recognized in the past as a successful example of shaping:

> *Gerald Tesauro's Backgammon playing agent achieved master level play through self-play ... Self-play is a sort of shaping, since at first the agent plays against a nearly random opponent and thereby solves an easy task. The complexity of the task then grows as the agent gets better at playing.*
> (Randløv and Alstrøm [160], p. 466)

TDL vs. CEL

Several papers directly compare the effectiveness of TDL and CEL applied to games like Othello [120, 192, 207], small-board Go [164], Backgammon [39] or the card game of Rummy [106]. Generally speaking, the reported results show that TDL and CEL exhibit complementary features. Typically, TDL learns much faster but then got stuck, and, no matter how many training games it plays, does not improve its performance. CEL, by contrast, progresses slower but, if properly tuned, for some domains and specific policy representations can eventually outperform TDL.

In this chapter, we ask whether it is possible to combine the advantages of these two implicit forms of shaping in a single algorithm that would develop better solutions than any of these methods on its own. To this aim, we propose a hybrid method referred to as *coevolutionary temporal difference learning* (CTDL, [192, 193]) that works by combining elements of gradient-descent learning and population-based search. To verify the potential synergistic effect of this combination, we apply CTDL to learning n-tuple networks for Othello (see Section 6.3) and WPCs for small-board Go (Section 6.4). Before we present the results of conducted experiments, in the next section we discuss general characteristics of the employed learning algorithms.

CTDL

6.2 LEARNING GAME-PLAYING POLICIES

For most nontrivial games, it is impossible to represent a policy directly as a mapping from states to actions, due to huge number of states. Thus, typically, a more concise way of representing policies needs to be employed (cf. Section 2.2.1). Importantly, the choice of policy representation determines the size and characteristics of the hypothesis space of the learning problem.

In this section we discuss TDL, CEL and CTDL in the context of learning policies for perfect information deterministic zero-sum board games. We assume that policies are represented as position evaluation functions (see Section 5.1.2) approximated by some sort of neural networks. In such case, learning game-playing policies can be naturally viewed as searching through a space of parameters of such function approximators (a parametric policy space).

Search perspective

6.2.1 *Temporal Difference Learning*

The use of temporal difference learning for elaborating game-playing policies stems from modeling board games as MDP tasks, where the goal is to maximize the expected reward in the long run. In such tasks, the state space contains all possible board positions while the actions represent legal moves. The essential feature of this scenario is that the actual reward is determined by the game outcome and thus it is not known before the end of the game.

In this chapter we employ TDL in the form of gradient-descent $TD(\lambda)$ illustrated in Algorithm 2.1. Apart from setting parameters like α, λ and ϵ, the algorithm requires a specification of the training MDP environment. Following the success of TD-Gammon, we conduct learning in a self-play environment. Consequently, a policy being learned is used to select actions for both players, alternately. The rewards occur only in terminal states and equal to $+1$ if the winner is black, -1 if white, and 0 when the game ends in a draw.

Self-play MDP environment

Technical details

Technically, we use Algorithm 2.1 to adjust the weight vector $\vec{\theta}$ of a neural network that is used to calculate the value of position evaluation function f. However, in principle, the algorithm attempts to learn a state value function $V_{\vec{\theta}}$, which is closely related to f albeit not necessarily identical. Unlike f which aims at relative ordering of particular game states, $V_{\vec{\theta}}$ predicts the expected return from the given state till the end of the game. Under the assumed reward scheme, this return is limited to the range $[-1, 1]$. Thus, to employ f as a return predictor in the course of learning its values are subsequently squeezed to this range using hyperbolic tangent. The same setting was used in the previous works [120, 164]. Importantly, squeezing the values of f does not change the policy it represents, as far as the squeezing function is monotonically increasing.

6.2.2 *Evolutionary and Coevolutionary Learning*

Complementary alternative for TDL

In the context of searching the parametric policy space, TDL can be seen as a single-point search with a gradient-based operator and as such may not be able to escape from local optima [197]. Evolutionary algorithms, described briefly in Section 2.2.3, have complementary characteristics — they maintain a population of policies, but have no means for calculating individually adjusted corrections for each policy weight. That lessens the problem of local optima by its implicit parallelism and random modification of candidate solutions. Consequently, evolutionary computation seems to be an attractive complementary alternative for TDL for learning game-playing policies.

Objective fitness

The main search driver in evolutionary search is fitness function. However, one faces substantial difficulty when designing an absolute fitness function for the task of learning game policies. A truly *objective* assessment of individual's utility in case of games can be done only by playing against *all* possible opponent policies. For the majority of games this is computationally intractable. An alternative is to consider only a limited number of opponents, and thus, lessen the computational burden. In this case the sample of opponents used for evaluation could be formed by a predefined expert player(s) or a sample of random opponents [33].

Relative fitness

In this light, coevolution is an appealing alternative that offers a natural way of designing fitness function. In a single-population coevolutionary algorithm (see Section 4.2.1), *relative* performance of individuals is calculated on the basis of the results of their interactions with other population members. In learning game policies, an interaction consists in playing a game and increasing the fitness of the winner while decreasing the fitness of the loser. In the evaluation scheme adopted here, individuals play games with each other in a round-robin fashion and the outcomes of these interactions determine their fitness values.

Algorithm 6.1 Coevolutionary temporal difference learning.

```
1: P ← CREATE RANDOM POPULATION()
2: EVALUATE POPULATION(P)
3: while ¬TERMINATION CONDITION() do
4:     S ← SELECT PARENTS(P)
5:     P ← RECOMBINE AND MUTATE(S)
6:     for all θ⃗ ∈ P do
7:         TDL(θ⃗)
8:     end for
9:     EVALUATE POPULATION(P)
10: end while
11: return GET FITTEST INDIVIDUAL(P)
```

EL and CEL for games

Evolutionary learning (EL) and coevolutionary learning (CEL) of game policies used in this chapter follow the above presented ideas. They typically start with generating a random initial population of player individuals (policies). Individuals are evaluated with an objective or relative fitness function for EL or CEL, respectively. The best performing policies are selected, undergo genetic modifications such as mutation and crossover, and their offspring replace some of (or all) former individuals. In practice, this generic scheme is usually supplemented with various details which causes EL and CEL to embrace a broad class of algorithms that have been successfully applied to many two-person games, including Backgammon [152], Chess [81], Checkers [29, 63], NERO [185], Pong [135], and AntWars [93]. In particular, Lucas and Runarsson used $(1 + \lambda)$ and $(1, \lambda)$ evolution strategies (cf. Algorithm 2.3) to learn policies for the games of small-board Go [164] and Othello [120].

6.2.3 *Coevolutionary Temporal Difference Learning*

Combining TDL and CEL

In order to benefit from the complementary features of TDL and CEL, it sounds reasonable to combine these approaches into a single hybrid algorithm. Following this motivation, we proposed a method termed coevolutionary temporal difference learning (CTDL, [192, 193]). CTDL is a straightforward hybrid that exploits different characteristics of the search process performed by each constituent method. It maintains a population of policies and alternately performs CEL and TDL. In the CEL phase, individuals are evaluated on the basis of a round-robin tournament and a new generation is obtained using standard selection and variation operators. Then, in the TDL phase, each policy is subject to a number of $TD(\lambda)$ self-play training games. This succession of CEL and TDL repeats in cycles. The pseudocode of CTDL is presented in Algorithm 6.1.

Related work

Other hybrids of TDL and CEL have been occasionally applied for learning game-playing policies. Kim et al. [102] trained a population of neural networks with $TD(0)$ and used the resulting policies as an initial population for the standard genetic algorithm with mutation as the only variation operator. Singer [179] combined coevolution with temporal difference learning which, as the author suggests, "may be superior to random mutation as an engine for discovery of useful substructures". A similar approach with TDL used as a weight mutation operator in a coevolutionary algorithm was recently investigated by Manning [126]. Contrary to CTDL, which uses straightforward coevolution with no long-term memory mechanism, the author employed the Nash Memory algorithm [59] with bounded archives.

Lamarckian inheritance

It is worth noticing that hybridization of evolutionary learning and temporal difference learning can be considered as a form of memetic algorithm. Memetic algorithms [139] are hybrid approaches coupling a population-based global search method with some form of local improvement. Since these algorithms usually employ evolutionary search, they are often referred to as Lamarckian Evolution, to commemorate Jean-Baptiste Lamarck who hypothesized, incorrectly in the view of today's neo-Darwinism, that the traits acquired by an individual during its lifetime can be passed on to its offspring.

6.3 LEARNING N-TUPLE NETWORKS FOR OTHELLO

In this section we present the experimental results of learning policies represented by n-tuple networks for the game of Othello (see Section 5.1.2.4). We conducted several experiments to determine how the considered learning methods fare for different sizes of networks with respect to selected performance measures. In particular, we aimed to answer the following questions: How do the algorithms scale with the size of policy representation? Is the performance against a heuristic player a good predictor of player's likelihood to beat other opponents? What is the ability of the policies trained with particular methods to play against previously unseen opponents?

6.3.1 *Experimental Setup*

All algorithms were implemented within our coevolutionary algorithms library called cECJ [191] built upon Evolutionary Computation in Java (ECJ) framework [121]. Our unit of computational effort is a single game and the computing time of other stages of evolutionary cycle is neglected. To provide fair comparison, all runs were stopped when the number of games played reached 3 000 000. Each experiment was repeated 24 times.

6.3.1.1 *Policy Representation*

We rely on n-tuple networks (see Section 5.1.2.4) because of its appealing potential demonstrated in recent studies [117, 126] and promising results in the Othello League (cf. Section 5.1.2.1). We start from small networks formed by seven 4-tuples (7×4), which include 567 weights. Later, we move to 9×5 networks (2187 weights on aggregate) to end up with the largest 12×6 architecture (8748 weights), which has been recently successfully applied to Othello by Manning [126]. This progression enables us to observe how particular learning methods cope with the growing dimensionality of the search space. A candidate solution is represented as a concatenation of lookup table weights associated with its n-tuples

N-tuple networks

We decided to employ the input assignment procedure that results in randomly placed snake-shaped tuples (see Section 5.1.2.4). Regarding the look-up table weights, their initial values depend on the particular learning algorithm. As previous research has shown [193], TDL learns faster when its weights are initialized with zeroes. Evolutionary methods, on the other hand, assume that the population is randomly dispersed in the search space. For this reason, in the purely evolutionary and coevolutionary algorithms (i.e., without TDL) we start with weights initialized randomly in the $[-1, 1]$ range.

Initialization

6.3.1.2 *Search Operators*

The considered learning algorithms perform search in two spaces — a discrete network topology space and a continuous weight space. Dimensions of the topology space are: the number of tuples, their length and input connections. Dimensionality of the weight space depends directly on the number of weights and grows with the number of n-tuples and their lengths. We search both spaces in parallel as it gives us more flexibility than searching only one of them. However, to avoid excessive complexity, we limit topology changes just to input assignment — the number of n-tuples and their length stay the same throughout learning. Although the majority of methods applied to train neural networks are based on a fixed structure and search only the weight space, there are some exceptions which explore topology space too [183, 204].

Weight space and topology space

In accordance with the twofold nature of the policy space, we employ two types of operators: genetic (CEL) and gradient-based (TDL). The former group includes:

- *weight mutation* — each weight (LUT entry) undergoes Gaussian mutation ($\sigma = 0.25$) with probability $p_{mw} = 0.05$,
- *topology mutation* — each input (board location) is replaced, with probability $p_{mt} = 0.01$, by another input from its neighborhood,

- *topology crossover* — sexual reproduction with probability $p_x = 1.00$ — two individuals mate and exchange genes, i.e., entire tuples with look-up tables.

The only gradient-based operator works in the weight space and consists in running a single self-play game incorporating $TD(0)$ algorithm (see Section 6.2.1). We use learning rate $\alpha = 0.001$ and force random moves with probability $\epsilon = 0.1$.

6.3.1.3 *Learning Algorithms*

TEMPORAL DIFFERENCE LEARNING TDL (Section 6.2.1) operates solely in the weight space using a single network and self-play $TD(0)$ as the only search operator.

EVOLUTIONARY LEARNING EL (Section 6.2.2) is a generational evolutionary algorithm with population of 50 individuals. It operates in a loop of: 1) evaluation – the fitness of each individual is calculated as a sum of points obtained in 50 randomized games against the SWH player (cf. Section 5.1.3.1), 2) selection – evaluated individuals are subject to tournament selection with tournament size 5, 3) recombination – individuals undergo topology crossover, and 4) mutation – individuals are modified by weight and topology mutation.

COEVOLUTIONARY LEARNING CEL (Section 6.2.2) is a coevolutionary algorithm with population of 50 individuals. The algorithm operates in a similar fashion to EL, except for the evaluation phase, where a round-robin tournament is played between all individuals, with wins, draws, and losses rewarded by 3, 1, and 0 points, respectively. The total number of awarded points becomes individual's *competitive fitness*. For each pair of individuals, two games are played, with players swapping the roles of the black and the white player.

EVOLUTIONARY TEMPORAL DIFFERENCE LEARNING ETDL combines EL and TDL. Similarly to EL, it uses topology mutation and topology crossover, but instead of weight mutation, it employs the self-play TDL training. By default, in each TDL phase, a budget of 5000 training games is allocated to the players in the population and thus each individual plays 100 games.

COEVOLUTIONARY TEMPORAL DIFFERENCE LEARNING CTDL is a hybrid of CEL and TDL (see Section 6.2.3). The algorithm operates as ETDL but uses competitive fitness like CEL. Notice that CTDL extends CEL in the same way as ETDL extends EL. Moreover, CEL and CTDL (also: EL and ETDL) differ only in the way they search the weight space (weight mutation vs. TDL).

Furthermore, where possible, the parameters for the above algorithms were taken directly from related works [117, 120, 126]. In some cases the parameters were determined by preliminary experiments. This includes the value of σ for weight mutation and the number of TDL games in a single phase of hybrid algorithms.

6.3.1.4 *Performance Measures*

To monitor the progress of learning, 50 times per run (approximately every 60 000 games), we appoint the individual with the highest fitness as the *best-of-generation individual* (for TDL, the single policy maintained by the method is the best-of-generation by definition). By the *best-of-run individual* we mean the best-of-generation individual of the last generation. We identify method's performance with the performance of its best-of-run players. In particular experiments, the performance is calculated using the following measures (see Section 5.1.3 for details):

1. Performance against a heuristic player, i.e., percentage score against the SWH player (see Section 5.1.3.1).
2. The number of points in a round-robin tournament between the teams of best-of-generation players produced by particular algorithms (see Section 5.1.3.2).
3. The place taken in a round-robin tournament involving the best entries from the online Othello League (Section 5.1.3.3).

In the following sections (6.3.2–6.3.4) we employ, respectively, these three performance measures to evaluate the learning methods.

It should be emphasized that the outcomes of performance assessments are unavailable to learning algorithms and thus do not influence the learning process. In a machine learning perspective, the opponents used in the above measures form a testing set and are intended to verify the generalization capability. The only exception to this rule are EL and ETDL, where fitness assessment uses the same opponent as the first performance measure.

6.3.2 *Performance Against a Heuristic Player*

The first performance measure is the percentage of points (1.0 point for a win, 0.5 for a draw, calculated with respect to the maximum possible total score) obtained in 1 000 randomized games (500 as black and 500 as white) against the SWH player (see Section 5.1.3.1). Since all policies in our experiments are deterministic we force both players to make random moves with probability $\varepsilon = 0.1$.

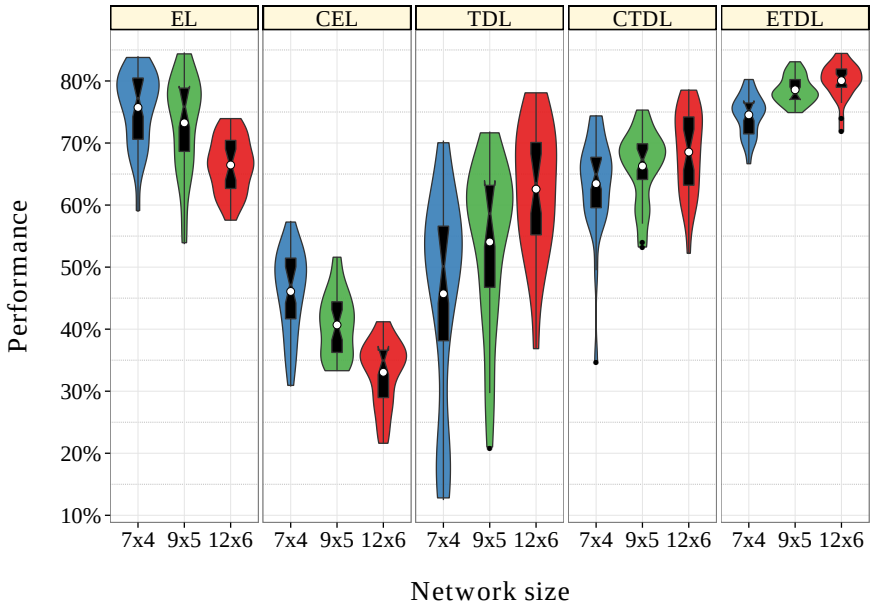


Figure 6.1: Comparison of learning methods for three network sizes. The performance measured against the SWH player is shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Narrowings of the box around the median indicate 95% confidence interval. White circles denote the mean performance.

6.3.2.1 *Scalability*

In the first experiments we focus on the scalability with respect to the policy representation size. Figure 6.1 illustrates how methods' performance against the heuristic player changes when moving from 7×4 to 9×5 and to 12×6 n-tuple networks. The figure shows violin plots [87] that summarize the distribution of final performance obtained with particular methods.

Increasing the network size

Interestingly, increasing the network size is not necessarily beneficial for all tested methods. Only the TDL-based methods are able to improve their performance by utilizing the possibilities offered by larger networks. On the contrary, EL and CEL perform even worse with larger networks than with the smaller ones. We hypothesize that the weight mutation operator is not efficient enough to elaborate fast progress in the larger (higher-dimensional) weight search space. This hypothesis is supported by all plots — only the methods involving weight mutation (EL, CEL) have such problems.

To make sure that this is not due to possibly unfavorable settings of weight mutation, we performed another experiment with different standard deviations (σ) of weight mutation. Results for EL presented in Fig. 6.2 show that our choice ($\sigma = 0.25$) is among the best values of deviation. Importantly, no matter what value of σ is used, the

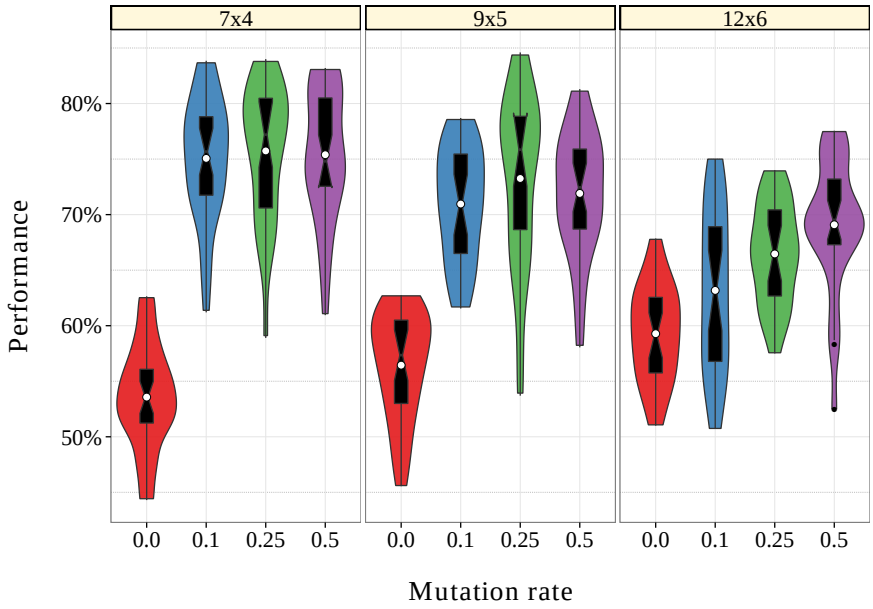


Figure 6.2: EL with different σ of weight mutation for three network sizes. The performance measured against the SWH player is shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Narrowings of the box around the median indicate 95% confidence interval. White circles denote the mean performance.

Verifying mutation rate

performance is lower with the larger networks. Conversely, when no weight mutation is used ($\sigma = 0$), larger networks allow for achieving better results. In this case the weights remain unchanged, and the evolutionary process modifies only the topologies of networks. Although this implies that weights remain fixed for an entire evolutionary run and therefore the total number of strategies that can be represented by individuals is more limited, the resulting search problem is easier and evolution eventually benefits from the larger network size.

Hybridization benefits

Finally, the hybrid methods perform either comparably (CTDL) or better (ETDL) with larger networks. Apparently, using TDL to search the weight space of n-tuple networks is more advantageous than applying random mutations, especially when the search space is larger. Hybridization allows evolutionary components to focus entirely on searching the topology space while leaving the continuous weight space to a dedicated gradient-based algorithm which works well on this problem when applied separately. Gradient-based update proceeds for each weight separately and remains effective no matter how many of them have to be optimized.

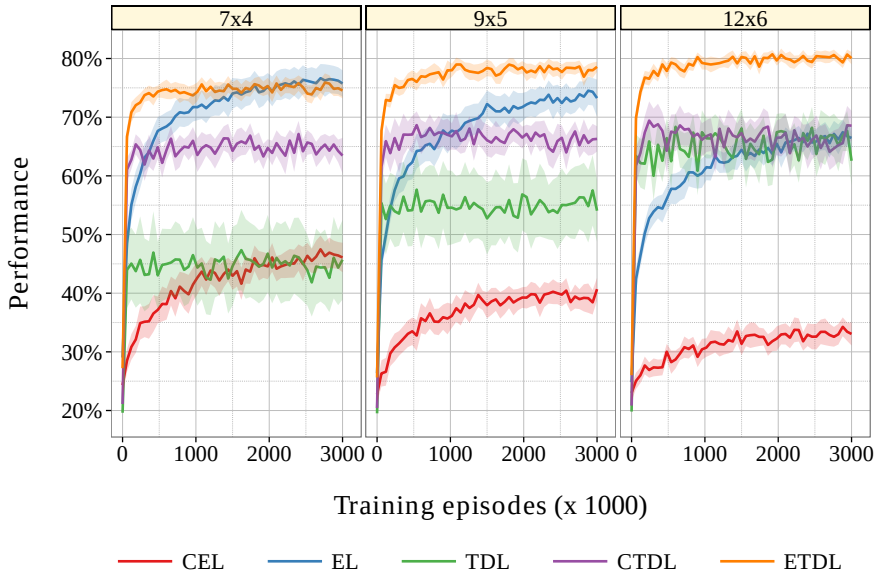


Figure 6.3: Comparison of learning methods for three network sizes. The average performance of the best-of-generation individuals against the SWH player shown as a function of the number of training episodes (games). Semi-transparent ribbons around the curves indicate 95% confidence intervals for the mean.

6.3.2.2 *Method Comparison*

Smallest networks

After answering the question whether larger representations pay off, we ask which method works best. Figure 6.3 compares all the methods for the three considered representation sizes. The results for the smallest 7×4 network demonstrate that the CTDL hybrid in the long run significantly outperforms the non-hybrid algorithms: TDL and CEL. Moreover, as also observed in previous research [120], TDL learns rapidly, whereas CEL advances slower but eventually reaches a similar or even slightly higher performance level.

Larger networks

However, while the superiority of CTDL over its constituents is still observable for 9×5 networks, for the 12×6 ones there is no difference between TDL and CTDL, which both score between 65% and 70%. This level is similar to that reported by Manning [126] for 12×6 networks trained with the Nash Memory approach (between 66% and 68%). This indicates a *ceiling effect* [211] in evaluation of self-learning methods with the WPC-heuristic performance measure. We hypothesize that the randomized SWH player does not offer sufficiently diversified challenge to differentiate the policies produced by these algorithms. To verify this claim and to differentiate the algorithms in terms of their performance, we conducted a series of performance assessments on a pool of opponents, detailed in Sections 6.3.3 and 6.3.4.

	CTDL	ETDL	TDL	CEL	EL	overall
CTDL		64.4%	63.5%	91.9%	95.7%	**78.9**%
ETDL	34.6%		53.4%	92.7%	95.4%	**69.0**%
TDL	35.6%	46.8%		89.7%	93.9%	**66.5**%
CEL	7.8%	6.2%	9.6%		72.9%	**24.1**%
EL	3.8%	4.0%	5.9%	26.2%		**10.0**%

Table 6.1: Results of the round-robin tournament between the teams of individuals from the last generations. Each number represents the percentage of obtained points; the percentages may not total 100% since there were 3 points for win and 1 for draw.

Importantly, when the performance is measured by playing against the SWH player, the evolutionary algorithms (EL and ETDL) perform better or not worse than the other ones. This is however not surprising, given that these methods have been guided by the fitness function using the very same opponent (SWH). As we will demonstrate in subsequent sections, these observations tell us very little about the performance of the trained players on another, more sophisticated, sample of opponents.

Evolutionary overfitting?

Last but not least, let us notice that ETDL performs better that EL for larger policy representations. This observation is another evidence supporting our claim that TDL mutation is much more efficient than weight mutation.

6.3.3 *Round Robin Tournament*

To widen the range of assessment opponents, we let the considered methods generate opponents for each other. Technically, every 60 000 training games we create teams that embrace all the best-of-generation strategies found by all 24 runs of particular methods. Next, we play a round-robin tournament between the teams (see Section 5.1.3.2). The score of a team is the overall sum of points obtained by its players (according to the *three points for a win* reward scheme, cf. Table 5.3). As the tournaments are played multiple times along evolutionary runs, we call this method *generational round-robin tournament*.

Generational round-robin tournament

Notice that this assessment scheme is *relative*: gain for one team implies loss for its opponent team. A team can be judged good due to its virtues, but also due to the weaknesses of other teams. Note also that generational round-robin tournament allows us to drop randomization of moves, since the presence of multiple opponents provides enough behavioral variability.

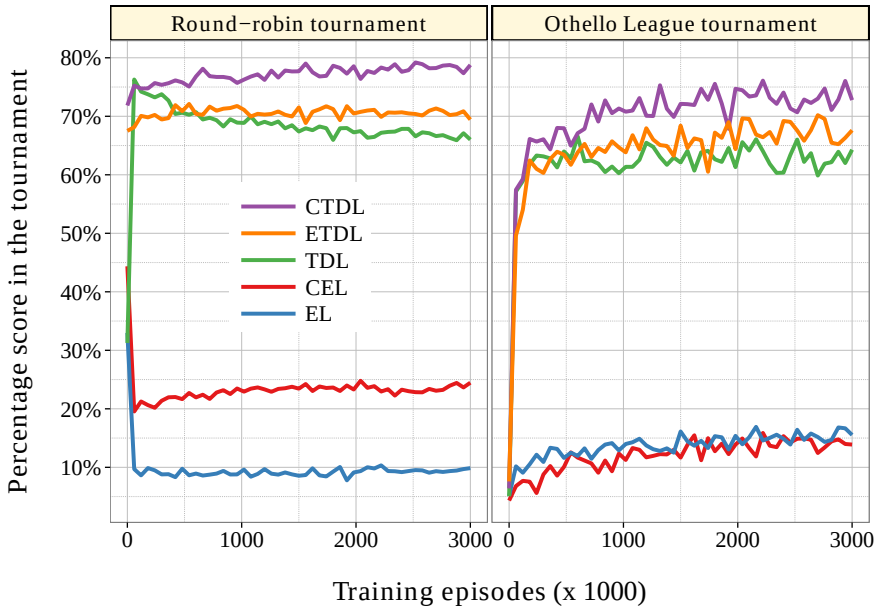


Figure 6.4: Generational round robin tournament and generational Othello League tournament for all methods using 12×6 networks. Percentage score is the score of a team of best-of-generation individuals normalized by the maximum possible score.

Relative performance analysis

Figure 6.4 plots the relative performance of all the algorithms using the 12×6 network. Noteworthy, the policies developed by TDL, CTDL and EL, which played at the same level against the SWH player (cf. Fig. 6.3) reveal varying levels of skills when evaluated against a different pool of opponents. As these opponents uncover previously unobserved differences between the methods and have been trained using different algorithms, we hypothesize that they are more behaviorally diversified. Let us also emphasize that the teams confronted here are composed of *the same* best-of-generation individuals that produced the results reported in Fig. 6.3, i.e., we assess here the outcomes of *the same* runs of learning algorithms.

Overfitting

In the tournament confrontation, the CTDL hybrid is clearly the winner and beats its constituent methods, TDL and CEL. Also, its advantage over the competitors increases over time. Interestingly, CTDL defeats both evolutionary methods. The superiority of CTDL is clearly demonstrated in Table 6.1, which shows the detailed results of the last round-robin tournament. This finding supports our intuition expressed in the previous section that ETDL and EL tend to overfit: they perform best against the SWH player, but fail when faced with another set of players that is likely to be more diversified. On the other hand, ETDL is still quite good, and in particular slightly better than TDL. We cannot say the same about EL, which wins only around 10% of games. Apparently, it is the self-learning TDL component of ETDL that reduces the negative effects of overfitting.

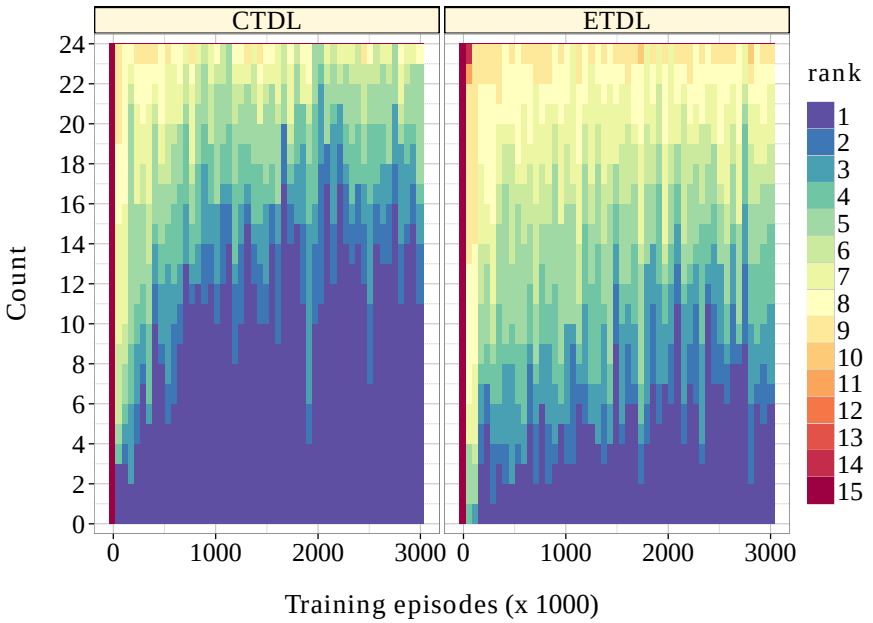


Figure 6.5: Frequency distribution of ranks obtained by ETDL and CTDL best-of-generation individuals in a round-robin competition with Othello League players. Bar height indicates how many times a particular rank was obtained by the 24 evolved players.

6.3.4 *Othello League Tournament*

One of our goals was to create a policy that would win in a direct confrontation with the best entries in the Othello League (see Section 5.1.2.1). To verify if this goal was attained, we employed a pool of top 14 Othello League contestants (cf. Section 5.1.3.3). Each of our best-of-generation players was assessed by playing against every opponent in the pool with wins, draws, and losses rewarded by 3, 1, and 0 points, respectively. Note that each player faces every other player twice — once as black and once as white.

Overall points in the league tournament

Figure 6.4 shows the performance of the trained policies expressed as a percentage of the maximum score possible to attain when confronted with the league pool. Each method (represented by 24 best-of-generation players) could score up to $14 \times 24 \times 3 \times 2 = 2016$ points (100%). Note that this assessment ranks the methods roughly in the same order as in the generational round robin tournament. Once again we can observe that ETDL turns out inferior to CTDL when the opponents are different from the strategy it was taught with.

Frequency distribution of league ranks

The total number of points is a valuable relative performance measure, but it does not inform us about the absolute places taken in the tournament by our best-of-generation players. Figure 6.5 shows how many times the 24 players produced by, respectively, ETDL and CTDL occupy particular ranks in the league. Clearly, CTDL leads to winning the tournament much more often than ETDL.

Name	Size	Played	Won	Drawn	Lost
epTDLmpx_12x6	12×6	100	89	1	10
prb_nt30_001	30×6	100	84	0	16
prb_nt15_001	15×6	100	83	3	14
epTDLxover	12×6	100	81	4	15
t15x6x8	15×6	100	79	3	18
SelfPlay15	12×6	100	77	0	23
tz278_2	278×2	100	76	3	21
Nash70	12×6	100	72	4	24
x30x6x8	30×6	100	71	4	25
pruned-pairs-56t	56×2	100	71	1	28

Table 6.2: Othello League ranking (as of the end of August 2013).

Th best player in the Othello League

The above experiments have shown that the policies produced by ETDL are less versatile than the ones produced by CTDL. However, when evaluated against the SWH player, ETDL appears remarkably successful. As we could see in Fig. 6.3, in an average run it attained the performance level of 80%. Moreover, one of the runs produced a player that reached 87.1% and took the lead when submitted to the online Othello League under the name of epTDLmpx_12x6. Table 6.2 shows the results[1] of the top ten entries in the league as of the end of August 2013. All players in the table are based on the same n-tuple network architecture, but the networks they employ vary in size.

6.3.5 *Analysis of Network Topology*

Frequency of board fields occurrences

Besides comparing the performance of learning algorithms, we were also interested in the internal representation of the best policies. For this reason, we examined topologies of the produced n-tuple networks and gathered statistics on the best-of-run players evolved by the CTDL method. Figure 6.6 demonstrates how many times a particular field of the Othello board was covered by the tuples of the best policies. Shading reflect the number of times a field appeared in networks. Certainly, tuples cumulate around the corners, which appear to be the most important fields on the board. Also, topology mutations pressed networks to abandon the central fields which, on the contrary, have less influence on the board evaluation and four of them are already occupied at the beginning of the game.

1 Since in the online league (see Section 5.1.2.1) players play only 100 games, there is a difference between our estimation of epTDLmpx_12x6 performance (87.1%) and 89.5 points obtained in the league.

510	301	199	165	165	199	301	510
301	292	156	119	119	156	292	301
199	156	72	80	80	72	156	199
165	119	80	58	58	80	119	165
165	119	80	58	58	80	119	165
199	156	72	80	80	72	156	199
301	292	156	119	119	156	292	301
510	301	199	165	165	199	301	510

Figure 6.6: Frequency of board field occurrences in n-tuples.

Topology of the best player

Figure 6.7 presents the topology of the best CTDL 12×6-tuple policy. The arrangement may seem sparse, but due to the 8-fold symmetry mirroring (not shown here but discussed in Section 5.1.2), this strategy in fact covers almost all board fields. Most tuples watch the combinations of fields that are known to be strategically important in Othello: neighboring fields close to corners, or the corners on two opposite sides of the board.

6.3.6 *Results Summary*

Solution concepts perspective

Although our best ETDL-evolved player has taken the lead in the Othello League, it fares much worse when facing head-to-head the other players from the League and the players evolved by means of coevolutionary algorithms. This phenomenon may be explained in terms of *solution concepts* [56]. ETDL uses the evaluation function based on the SWH player, and so optimizes the individuals against this specific opponent. The policies it trains do not have a chance to play with different opponents and learn from such experience. Clearly, randomization of the SWH player, intended to increase behavioral diversity, does not help in this regard. Formally, ETDL implements the specific solution concept of maximization of expected score *against the* SWH *player*[2], which, at least for the game of Othello, does not seem to be a good predictor of general Othello-playing skills (expected utility solution concept, cf. Section 5.1.3.4). This observation applies also to the way the Othello League ranks players, and limits the conclusions that may be drawn from that ranking.

2 In Othello with randomized moves.

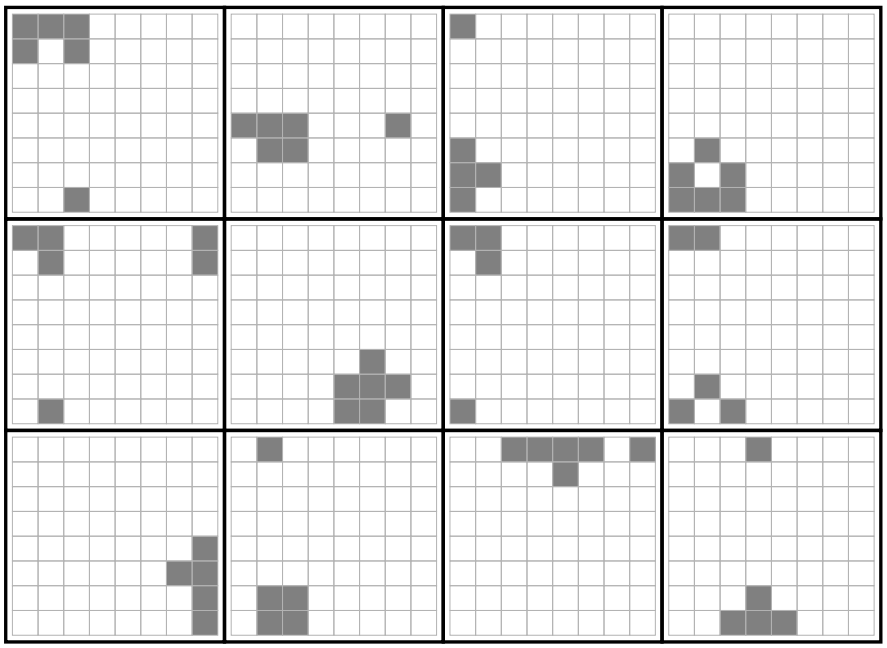


Figure 6.7: The tuples of the best CTDL player superimposed on the Othello board.

Generalization performance

In contrast, CTDL, a self-learning method equipped with dynamic evaluation function and based on coevolution and temporal difference learning, yields policies that generalize much better and successfully compete with a variety of opponents: evolved, coevolved, trained by TDL, and the top players submitted to the Othello League. In particular the last ones, by implementing various approaches and submitted by different researchers, can be claimed to represent a richer repertoire of behaviors. Having said that, we do not argue that CTDL implements any known solution concept. However, the results of extensive round-robin tournaments indicate that it is closer to the solution concept of maximization of expected utility for 1-ply Othello than any other method considered here, in particular the top-ranked strategies from the Othello League.

Search space dimensionality

Lucas and Runarsson [120] have found that coevolution applied to policies represented as WPCs learns much slower than TDL, but eventually converges to solutions of similar quality. The results reported in Section 6.3.2 shed new light on this issue. The performance gap between the coevolutionary algorithms and TDL strongly depends on the dimensionality of the search space. For 7×4-tuple networks (567 weights), the coevolutionary algorithm (CEL) in the long run indeed achieves results comparable to TDL, but TDL proves far better for larger search spaces of 9×5 and 12×6 networks (2187 and 8748 weights, respectively). Its gradient-based learning rule is relatively insensitive to the number of variables of consideration, while coevolution does not seem to be able to catch up, even in the long run.

The evolutionary algorithm (EL), despite obtaining higher absolute score against the SWH player, also tends to attain worse performance for larger networks. The common factor that appears to be responsible for these difficulties is the weight mutation operator, which seems to work reasonably well only in smaller search spaces (cf. Fig. 6.2). On the other hand, some form of mutation is necessary for the evolutionary approach (Fig. 6.2 shows that without mutation the score is even worse). This points to the need of designing better mutation operator for the given search space. Indeed, even random mutation proved effective in high-dimensional spaces in some previous studies [63, 102].

Weight mutation operator

6.4 LEARNING WEIGHTED PIECE COUNTERS FOR THE GAME OF SMALL-BOARD GO

In this section we attempt to verify if CTDL proves beneficial also for the game of small-board Go and policies represented by WPCs (see Section 5.2.3). In contrast to the previous section, here we focus solely on hybridizing coevolution with temporal difference learning and not on the comparison with evolutionary methods (like EL or ETDL). For this reason, the reported experimental result concern only the TDL, CEL and CTDL methods. We conduct also a preliminary experiment to find the advantageous values of the λ parameter for the $TD(\lambda)$ algorithm.

6.4.1 *Experimental Setup*

Similarly as for Othello (cf. Section 6.3.1), all algorithms were implemented within our coevolutionary algorithms library called cECJ [191]. It was assumed that the uttermost element influencing the time of training is the time required to play a game, so the time consumed by such operations as selection, mutation, evaluation, has been neglected. In other words, our unit of computational effort is a single game. To provide fair comparison, all runs were stopped when the number of games played reached 2 000 000. For statistical confidence, each experiment was repeated 25 times.

6.4.1.1 *Policy Representation*

Since we are mainly interested in analyzing the relative improvements that the hybridized CTDL method can bring when compared to its constituents, the absolute player's performance is of secondary importance. Therefore, we employ here WPC, the least sophisticated policy representation among those considered in this thesis.

6.4.1.2 *Learning Algorithms*

COEVOLUTIONARY LEARNING CEL uses a generational coevolutionary algorithm with population of 50 individuals, each being a 5×5 WPC matrix initialized with random weights drawn from the $[-1, 1]$ range. In the evaluation phase, a round-robin tournament is played between all individuals, with wins, draws, and losses rewarded by 3, 1, and 0 points, respectively. For each pair of individuals, two games are played, with players swapping the roles of the black and the white player. The evaluated individuals are subject to tournament selection with tournament size 5, and then, with probability 0.03, their weights undergo Gaussian mutation ($\sigma = 0.25$). Next, individuals mate using one-point crossover, and the resulting offspring form the subsequent generation. As each generation requires 50×50 games, each run lasts for 800 generations to get the total of 2 000 000 games.

TEMPORAL DIFFERENCE LEARNING TDL is an implementation of the gradient-descent temporal difference algorithm $TD(\lambda)$ described in Section 6.2.1. The weights are initially set to 0 and the learner is trained solely through self-play, with random moves occurring with probability $\epsilon = 0.1$. The learning rate was set to $\alpha = 0.01$; the value of trace decay λ will be determined in Section 6.4.2.

COEVOLUTIONARY TEMPORAL DIFFERENCE LEARNING CTDL is a combination of CEL and TDL (see Section 6.2.3), which both use the same parameters as described above. The individuals are initialized randomly like in CEL. The algorithm alternates the TDL and CEL phases until the total number of games attains 2 000 000. In a single TDL phase each individual plays 8 games, so there are $50 \times 50 + 8 \times 50 = 2900$ games per generation, which leads to 690 generations.

6.4.1.3 *Performance measures*

To monitor the progress of learning, 50 times per run (approximately every 40 000 games), we appoint the individual with the highest fitness as the *best-of-generation individual* and assess its performance (for TDL, the single policy maintained by the method is the best-of-generation by definition). In particular, the performance is calculated using the three following measures (see Section 5.2.4 for details):

1. Probability of winning against a predefined, human-designed WPC heuristic policy (see Section 5.2.4.1).
2. Probability of winning against the Liberty player (Section 5.2.4.2).
3. The number of points in a round-robin tournament between the teams of policies produced by particular methods.

λ	against WPC heuristic	against Liberty Player
0.00	0.420 ± 0.129	0.496 ± 0.116
0.20	0.444 ± 0.123	0.532 ± 0.102
0.40	0.465 ± 0.125	0.547 ± 0.104
0.60	0.483 ± 0.130	0.559 ± 0.102
0.80	0.497 ± 0.134	0.560 ± 0.098
0.90	0.543 ± 0.131	0.564 ± 0.093
0.95	0.599 ± 0.140	$\mathbf{0.567 \pm 0.089}$
0.96	0.597 ± 0.143	0.557 ± 0.089
0.97	0.613 ± 0.133	0.563 ± 0.086
0.98	$\mathbf{0.630 \pm 0.148}$	0.557 ± 0.084
0.99	0.617 ± 0.157	0.554 ± 0.094
1.00	0.545 ± 0.195	0.548 ± 0.109

Table 6.3: Probability of winning against WPC heuristic and Liberty Player for a policy found by TD(λ) with different trace decays λ.

To calculate the first two measures, the best-of-generation policy plays 1 000 games against the opponent player (500 as black and 500 as white, with probability of random moves $\epsilon = 0.1$). It should be emphasized that the interactions taking place in all assessment methods do not influence the learning individuals.

6.4.2 *Preliminary Experiments*

In the first experiments, the best value of trace decay λ was determined by running TDL with various settings of this parameter. Technically, because the randomized self-play causes the performance of the TDL learner to vary substantially with time, we decided not to rely on the *final* outcome of the method alone. To make the estimates more robust, we sampled each run every 40 000 games for the last 800 000 games and averaged the performances of strategies (thus, the performance of each run was estimated using 20 individuals).

Table 6.3 shows the results averaged over 25 runs (this holds for all experiments, unless stated otherwise). For WPC-heuristics, the winning rate is maximized for $\lambda = 0.98$, while for Liberty Player, this happens for $\lambda = 0.95$. Because the outcomes of games against the Liberty Player turn out to be less sensitive to λ than for WPC-heuristics, and the differences for $\lambda \in [0.6; 0.99]$ are very small for the former opponent, we chose 0.98 as the optimal value for λ to be used in all further experiments.

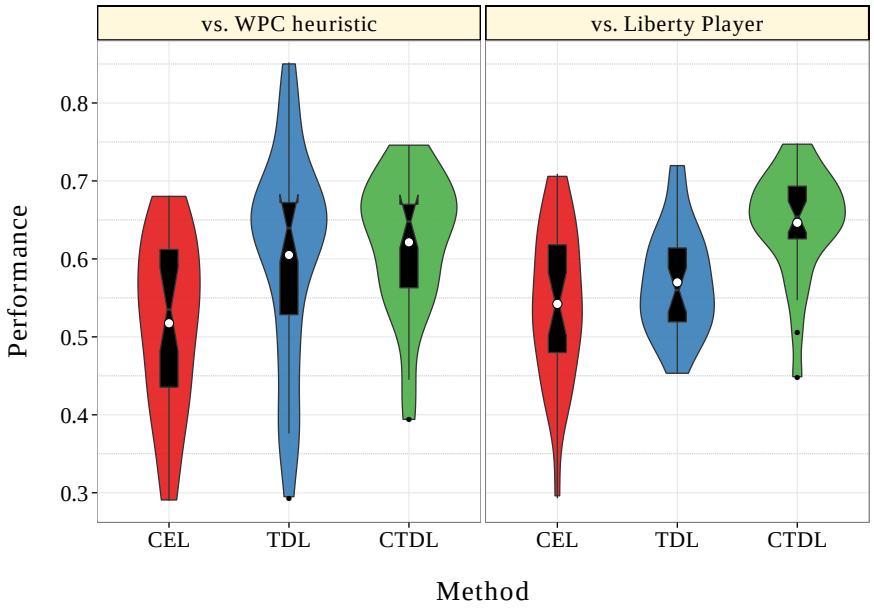


Figure 6.8: Comparison of learning methods. Violin plots show the performance of the best-of-generation individuals measured as the probability of winning against (a) WPC heuristic and (b) Liberty Player. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Narrowings of the box around the median indicate 95% confidence interval. White circles denote the mean performance.

6.4.3 *Method Comparison*

In the main experiment, CTDL was compared with its constituent methods: CEL and TDL. Figure 6.8 shows the distribution of final performance achieved by these methods measured by playing against the randomized external players: WPC heuristic and Liberty Player. Both measures agree that CEL is significantly worse than CTDL and produces policies that win barely more than 50% of games. Interestingly, TDL seems as good as CTDL when playing with WPC heuristic, but is significantly worse (and comparable with CEL) when crossing swords with Liberty Player. The best of CTDL players attained more than 65% winning rate with both external players.

Though the performance of all methods in absolute terms is rather moderate, this should be attributed, in the first place, to the simplicity of WPC, which is not necessarily the most suitable representation for the highly non-positional game of Go. Let us also notice that the performance of the optimal WPC policy for this game is unknown, so judging the above probabilities as objectively good or bad would be inconsiderate.

	CTDL	CEL	TDL	total
CTDL		56.5%	57.0%	**56.8**%
CEL	42.5%		53.0%	**47.7**%
TDL	41.0%	45.5%		**43.3**%

Table 6.4: Results of the round-robin tournament between the teams of individuals from the last generations. Each number represents the percentage of obtained points; the percentages may not total 100% since there were 3 points for win and 1 for draw.

6.4.4 *Round Robin Tournament*

The above comparison demonstrates that the fusion of CEL and TDL can be beneficial in terms of the absolute performance against external heuristic players. To gain insight in the relative performance of policies, we played a round-robin tournament between the three teams representing particular methods, where each team member confronts all $2 \times 25 = 50$ members from the opponent teams for a total of 100 games (50 as white and 50 as black). The final score of a team is determined as the sum of points obtained by its players in overall 2 500 games, using the three-point-for-a-win reward scheme (see Table 5.3). Thus, the maximum number of points possible to get by a team in a single tournament is $2\,500 \times 3 = 7\,500$.

Table 6.4 presents the detailed results of a round-robin tournament among the teams of best-of-run individuals, i.e., the best-of-generation individuals found in the last generation, after 2 000 000 games. Not all our previous conclusions were confirmed in this relative performance assessment. Most notably, although TDL was found clearly better than CEL when gauged against the randomized WPC heuristic player, it is now the worst method in the field. Importantly, CTDL does confirm its superiority and wins in direct confrontations with its two constituent methods.

6.4.5 *Results Summary*

The results presented in this section confirm the observations from Section 6.3, where we demonstrated that hybridizing coevolution with $TD(0)$ proves beneficial when learning n-tuple networks for the game of Othello. Here, we come to similar conclusions with learning WPCs for the game of small-board Go. Additionally, we can observe that extending the lookahead horizon by using $TD(\lambda)$ with λ close to 1 can boost the performance of TDL, and has a positive impact also on the performance of CTDL. Consequently, there is growing evidence to support our claim that hybridizing coevolution with temporal difference learning can be beneficial.

6.5 DISCUSSION AND CONCLUSIONS

This chapter is an attempt to bridge the gap between coevolutionary learning and self-play temporal difference learning — two implicit forms of shaping that have been widely used for learning game-playing policies. The proposed approach of coevolutionary temporal difference learning is an interesting mixture that can be analyzed from many perspectives. Regarding biological inspirations, CTDL can be considered as a realization of Lamarckian coevolution, since organisms (policies) learn here throughout their life (TDL learning phase) to pass the acquired traits on to the offspring. As in nature, learning occurs at many scales of space and time, on two adjacent levels of individuals and populations [1].

Biological perspective

From another perspective, the proposed hybridization performs simultaneous search in two different modes: local (intra-game) and global (inter-game). To conduct the search in the former mode, it employs gradient-based local search operator that works with a single solution at a time and trains it by a randomized self-play. For the inter-game mode, our method relies on population of policies, makes them play against each other, and uses the outcomes of games to guide the process of random sampling of the search space in subsequent generations. Therefore, CTDL hybridizes two radically different techniques that complement each other in terms of exploration and exploitation of the search space.

Search perspective

The empirical evaluation of CTDL demonstrates that, at least for the considered games and policy representations, coevolutionary learning algorithm can autonomously select and maintain a dynamic training environment that cause the resulting policies generalize better than the policies trained by an evolutionary approach. Put another way, the *shaping* process as realized here by coevolution turns out to be effective. We can only hypothesize that the major reason for this is greater behavioral diversity of coevolutionary opponents. What nevertheless follows from the experimental results is that the coevolutionary opponents are diversified "in the right way", i.e., they guide the learning process towards the more versatile policies.

Coevolutionary shaping

However, to find the candidates for a sample of opponents in the first place, an effective search operator is indispensable, particularly when the dimensionality of representation is high. The gradient-based search operator proved most useful in this respect, in contrast to the purely random mutation. The resulting hybrid, CTDL, may be then seen as a successful combination of effective individual learning mechanism (TDL) with an appropriate method for filtering out the right opponents (CEL). This hybrid seems to scale well with the dimensionality of the search space, i.e., the policies it yields generalize better for larger representations. In case of learning n-tuple networks, this hybridization turns out truly advantageous when coevolution

Effective search operator

operates exclusively in the network topology search space, leaving the search in the space of weight values entirely to TDL.

Assessment discrepancies

Another lesson learned from this chapter is that assessing players using various performance measures can lead to qualitatively different outcomes, even if all of them take care of making the opponents diverse. Thus, great caution should be taken when drawing conclusions from such results.

Focus on relative gains

Finally, although our evolved policies would most probably yield to other contemporary players that use more sophisticated representations, we need to emphasize that our primary objective was to hybridize two algorithms that learn fully autonomously and study the *relative* gains that result from their synergy. To quote Arthur L. Samuel's declaration:

> *It should be noted that the emphasis throughout all of these studies has been on learning techniques. The temptation to improve the machine's game by giving it standard openings or other man-generated knowledge of playing techniques has been consistently resisted* (Samuel [169], p. 215).

7 SHAPING IN EVOLUTIONARY LEARNING

This chapter provides the core contribution of the thesis, namely, difficulty-based shaping methods for generalized reinforcement learning domains. The goal of learning in such domains is not to perform well in a single task but to generalize across many related tasks sampled from the given *target* task distribution. The proposed shaping approach consists in providing artificially distorted *training* task distributions that allows to improve the performance of evolutionary learning algorithms. The introduced measure of task *difficulty* allows to shape increasingly more demanding task distributions.

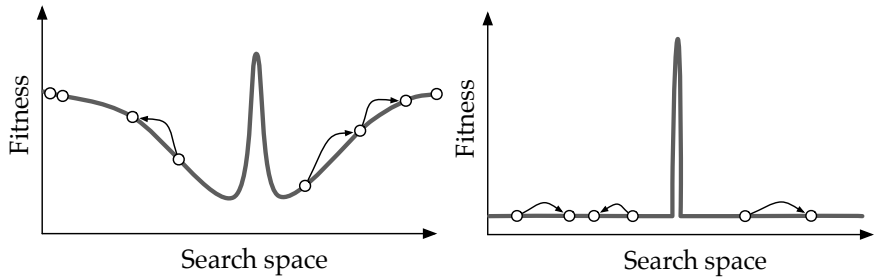
To learn a policy for a multi-task domain, we use an evolutionary approach which relies on modeling a learning problem in terms of optimization. Thus, we start by investigating what contributes to the difficulty of such optimization problems and how the existing shaping approach of incremental evolution allows to scale the problem difficulty (see Section 7.1). By contrast to the incremental evolution approach, which typically requires human supervision, in Section 7.2 we propose a set of more autonomous shaping methods. Most of these methods provide training tasks according to an easily computable task difficulty measure. Apart from defining this measure, we formalize the problem of multi-task learning and discuss how it is conventionally approached by the means of direct evolution.

Section 7.3 briefly discusses the methodology of empirical evaluations and comparisons of shaping methods. The following sections 7.4 – 7.6 provide the results of experiments conducted in three domains including the game of Othello and the problem of cart pole balancing. The results demonstrate that training on purpose-built task distributions is more effective than learning directly on the target distribution.

7.1 INTRODUCTION

The motivation for this chapter originates from the question: what can be done to improve the results of an evolutionary algorithm on a given problem with respect to a specific performance measure?

A typical approach is to modify the algorithm by tuning its parameters. Evolutionary algorithms involve numerous settings (including population size, variation operators, selection scheme) that need to be configured properly to attain satisfactory performance. Since devising such a configuration manually is usually nontrivial, various 'rule-of-thumb' recommendations, good practices and techniques of



(a) Deceptive fitness landscape.

(b) Neutral fitness landscape showing the *needle-in-a-haystack* problem.

Figure 7.1: Examples of difficult fitness landscapes. Circles represent individuals and arrows show how they change by following the fitness gradient. The figure is adapted from Weise et al. [216].

automated parameter tuning have been proposed in the past [50]. Ultimately, if the configured algorithm still does not achieve the expected performance, it can be entirely replaced by a different one. Parameter tuning and algorithm selection can be also automated using hyper-heuristics [26].

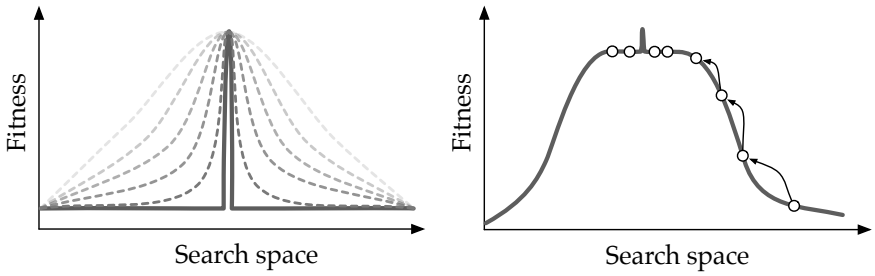
It may seem that there are no alternative answers to the question posed above. Indeed, of the three mentioned elements (algorithm, problem, performance measure), only the first one appears to be in experimenter's control. Contrary to this belief, there is actually another option: rather than adjusting the algorithm to a particular problem, we can take a complementary approach and modify the problem to make it easier for a particular algorithm to reach the original optimum. Such approach can be seen as an implementation of shaping in the realm of problem solving.

In the following sections, we investigate what makes the problem difficult to solve by evolutionary methods (Section 7.1.1) and how to mitigate difficulties by applying shaping in the form of incremental evolution (Section 7.1.2). Finally, in Section 7.1.3 we identify weakness of existing incremental learning methods and propose an unsupervised variant of shaping for evolutionary learning in a generalized domain.

7.1.1 *Problem Difficulty*

Fitness landscapes

The efficiency of evolutionary algorithms, and heuristic optimization methods in general, on a given problem, is largely affected by the shape of the objective function. Many nontrivial optimization problems have been found difficult to solve because of deceptiveness or neutrality in the *fitness landscape* [216], meant as visualization of the objective function in the search space. Examples of fitness landscapes considered as difficult are illustrated in Fig. 7.1.



(a) Smoothening the fitness landscape. (b) Local structure showing neutrality.

Figure 7.2: Needle-in-a-haystack landscapes.

One source of problem difficulty is *deception*. While the original definition of deception was formulated by Goldberg [68] in the context of the building blocks hypothesis [88], intuitively it can be used to describe problems in which an evolutionary algorithm does not reach the desired objective in a reasonable amount of time [113]. In deceptive fitness landscapes (see Fig. 7.1a), the gradient (often meant only informally) of the objective function points to the wrong direction in large part of the search space. As a result, by strictly following the objective function the algorithm gets trapped in local optima and is unlikely to find the path to the globally optimal solution.

Deception

Another difficult fitness landscape is presented in Fig. 7.1b. In this case, although the objective function does not guide the search away from the optimum, it remains extremely uninformative. A large part of the landscape is *neutral*, i.e., it does not provide any information on gradient because all candidate solutions get the same fitness value. Since the search algorithm, with its perception usually limited to a part of the landscape, cannot be reasonably steered by the objective function, it usually has to resort to random search. A similar difficulty has been encountered in the field of evolutionary robotics where it is referred to as the *bootstrap problem* [141, 144]. The problem occurs if all individuals in the early stage of evolution perform equally poorly when attempting to perform a complex behavior.

Neutrality

An extremely neutral fitness landscape is featured by the *needle-in-a-haystack* problem investigated by Hinton and Nowlan [86]. In such a problem, there is only one point of higher fitness which forms a single spike in the landscape. The authors show that useful fitness gradient can be introduced by combining evolution with individual lifetime learning (which can be done by, e.g., hybridizing an evolutionary algorithm with local search techniques — cf. Chapter 6). In this way, as the authors state, "learning alters the shape of the search space in which evolution operates". Figure 7.2a shows the smoothened fitness landscapes, as 'perceived' by the hybrid algorithm. The fitness in the areas around the spike has been boosted, making the optimum easier to reach by some sort of (potentially randomized) learning.

Needle in a haystack

Fitness landscape features

Importantly, the fitness landscape may reveal different problematic features in different regions of the search space. For instance, the fitness function illustrated in Fig. 7.2b provides an informative gradient for almost entire search space. However, after reaching a neutral fitness *plateau*, finding the optimum resembles the needle in a haystack problem again. Noteworthy, the shape of the fitness landscape depends not only on the objective function but also on the structure of the search space which is defined by choosing a specific genetic encoding of solutions and designing search operators.

7.1.2 *Incremental Evolution*

Increasingly complex subtasks

One approach that has been proposed to alleviate the difficulties of deceptive and neutral fitness landscapes is *incremental evolution* [140]. Typically, this approach consists in dividing a complex task into a sequence of *subtasks* which may represent various problem simplifications. In reinforcement learning, subtasks may correspond to intermediate behaviors that should be mastered by the agent, and which are typically easier to evolve than the target one. The concept of incremental evolution has been particularly common in the field of evolutionary robotics [145, 206]. Mouret and Doncieux [141] classify the works in this area into four main approaches including, among others, *staged evolution* and *environmental complexification*, both of which conform to our definition of shaping.

Staged evolution

Staged evolution is the most intuitive incremental procedure and consists in splitting the original task into (typically a few) ordered subtasks. Each subtask defines a fitness function to be used at a particular stage of learning. It is the role of the experimenter to decide when to switch to the next subtask and thus change the objective of the evolutionary algorithm. An early successful example of such approach is given by Harvey et al. [79] who trained vision-based robots to distinguish between triangular and rectangular objects and move towards the former ones. The authors employed three stages in which robots learned recognition, pursuit and discrimination between the two geometric shapes. Since these stages correspond to particular types of behaviors required in the target task, this approach is also referred to as *behavioral decomposition* [34].

Environmental complexification

Environmental complexification operates in a similar manner to staged evolution but the difficulty of subtasks is changed more continuously by modifying some numeric task-specific parameters. For instance, Gomez and Miikkulainen [72] provide an increasingly demanding sequence of tasks in a prey capture domain by gradually increasing prey mobility. In particular, learning started there with an immobile prey, and then the number of initial prey moves and prey speed increased. The authors made a distinction between an *evaluation task* which is used to evaluate agent's fitness at some stage

of the learning process, and the *goal* task which the agent is learned to perform. The agent is trained on a succession of evaluation tasks which constitute steps towards solving the task of ultimate interest.

The bottom line is that subtasks are supposed to specify a sequence of increasingly difficult fitness landscapes culminating in that of the target task. Therefore, an evolutionary algorithm can start from a simplified version of the task and then gradually progress to more challenging ones. As a result, the evolutionary search can be guided through advantageous paths in the solution space and should, in principle, be more likely find the optimum.

Increasingly difficult fitness landscapes

7.1.3 *Unsupervised Shaping*

Despite several successful applications of incremental evolution approaches [73, 103, 148, 229], the main problem with most of them is a requirement of expert domain knowledge. Typically, such expertise is indispensable to build a 'pedagogical' sequence of tasks with increasingly demanding objectives and design the switching criteria which determine when to shift from one task to another. This requirement was accurately formulated by Lehman and Stanley [113]:

Incremental evolution requirements

> *Some researchers incrementally evolve solutions by sequentially applying carefully crafted objective functions to avoid local optima. These efforts demonstrate that to avoid deception it may be necessary to identify and analyze the stepping stones that ultimately lead to the objective so that a training program of multiple objective functions and switching criteria can be engineered. However, for ambitious objectives, these stepping stones may be difficult or impossible to determine a priori. Additionally, the requirement of such intimate domain knowledge conflicts with the aspiration of machine learning* (Lehman and Stanley [113], p.190).

The need of handcrafting both fitness functions and switching criteria causes that existing incremental evolution methods can be seen as supervised approaches that require substantial human intervention. This can be one of the reasons of their limited popularity. Thus, a key question arises: how can the incremental evolution approach be engineered autonomously, without human supervision?

Here we address this question in the context of evolving behaviors for *generalized reinforcement learning domains* [219, 220]. We assume that a problem is defined as a distribution of related MDP tasks and the goal is to find a policy that performs well on average across the entire domain. In such a setting, subtasks for incremental evolution can be naturally represented as subsets of tasks from the given domain. We expect to improve the efficiency of evolutionary algorithm by using carefully selected tasks to evaluate fitness of policies at successive stages of learning.

Generalized reinforcement learning domains

Autonomous incremental evolution

However, we still need to select the right subsets of evaluation tasks and decide when to switch between them. In the following section we show how these two design choices can be made autonomously to remove this burden from the human supervisor. In particular, we introduce a simple measure of task difficulty which is relatively easy to calculate and does not require fitness landscape analysis. Moreover, we propose a set of incremental evolutionary methods based on this difficulty measure that attempt to build progressively more demanding subsets of tasks. Ultimately, we employ coevolution to adaptively change the set of evaluation tasks as the population becomes increasingly more competent.

7.2 DIFFICULTY-BASED SHAPING IN GENERALIZED DOMAINS

In this section we propose a set of methods that realize shaping by providing training tasks that can be expected to facilitate learning of the given target task(s). In Section 7.2.1 we formalize the goal of learning by modeling a distribution of target tasks in the form of generalized reinforcement learning domain. Next, in sections 7.2.2 and 7.2.3 we show how to apply evolutionary approach, in both shaped and unshaped variant, to learn policies for generalized domains. Finally, Sections 7.2.4 and 7.2.5 introduce task difficulty and difficulty-based task pools, respectively, which are crucial for most of the shaping methods presented in Section 7.2.6.

7.2.1 *Generalized Reinforcement Learning Domain*

Multi-task learning scenarios

Although typically reinforcement learning is applied to mastering single tasks, many real-world scenarios require the agent to deal with multiple different but related tasks. To model such scenarios we can use the notion of *generalized domain* [219] (or *generalized environment* [220]) which embodies a distribution over related MDP tasks that vary with respect to some aspect of the problem. For instance, in the generalized helicopter hovering domain [105] the agent faces environments with different wind velocity. Similarly we could define, for that instance, a parameterized Othello-playing domain where the parameter would control the distribution of opponents.

Following the definition proposed by Whiteson et al. [219], a generalized domain $\mathcal{G} = \langle \mathcal{T}, \mathcal{P} \rangle$ consists of:

- $\mathcal{T}$ — a set of MDP tasks. We assume that all tasks in $\mathcal{T}$ share the same state space S and action space A, so we can use the same policy $\pi : S \to A$ across the entire domain. However, each task $\tau \in \mathcal{T}$ fully specifies its individual transition function T_τ, reward function R_τ and initial state distribution I_τ. However, in the considered experimental domains, the tasks differ only in one of these components, while the other two remain constant.

- $\mathcal{P}$ — a probability distribution over $\mathcal{T}$. The distribution can be defined implicitly by describing how a new task is generated.

There are several learning models that have been previously applied in such multi-task domains [182, 194, 228]. Typically, the agent experiences a sequence of MDPs drawn from distribution $\mathcal{P}$ and is allowed to adapt its policy online while interacting with a specific task in the sequence. In this chapter, by contrast, we assume that, after prior learning, agent's policy stays fixed while being evaluated in a series of tasks sampled from the domain. Consequently, the goal of learning is to find an optimal *cross-task policy*, i.e., such that maximizes the expected return in tasks drawn from the given generalized domain:

Cross-task policies

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[J(\pi, \tau) \mid \tau \sim \mathcal{P}\right], \tag{7.1}$$

where $J(\pi, \tau)$ is the expected return obtained by the agent following policy π in task $\tau = \langle S, A, T_\tau, R_\tau, I_\tau, \gamma \rangle$:

$$J(\pi, \tau) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \mid R = R_\tau,\, T = T_\tau,\, s_0 \sim I_\tau \right]. \tag{7.2}$$

In practice, to fully specify such a multi-task learning problem we need to define also the policy representation which determines the space of possible solutions to the problem.

7.2.2 *Evolutionary Algorithms in Generalized Domains*

Assuming that in a generalized domain $\mathcal{G}$ the task distribution $\mathcal{P}$ is known a priori (according to so called *open generalized* methodology [220]), we can naturally cast the multi-task learning problem as an optimization problem. In fact, by treating each task $\tau \in \mathcal{T}$ as a *test* which is used to evaluate fitness of policies, the problem can be also seen as a *test-based* problem (see Section 4.2.2).

Test-based problems

A common example of test-based problems are games, where the set of tests contains all possible opponents, and the objective is to maximize the *expected utility*, i.e., the average score on all tests [40]. Since many games concern sequential decision making and can be formulated as MDPs, such test-based problems can be equivalently modeled as generalized domains with uniform distribution over a set of game-playing tasks with different opponents.

Expected utility

In most nontrivial test-based problems, the large number of tests precludes computationally feasible calculation of objective function (e.g. the expected utility). For this reason, solving problems of this class with evolutionary computation requires substituting the original objective function with a computationally cheaper fitness function that drives the search process towards a possibly similar (and preferably the same) goal.

Computational feasibility

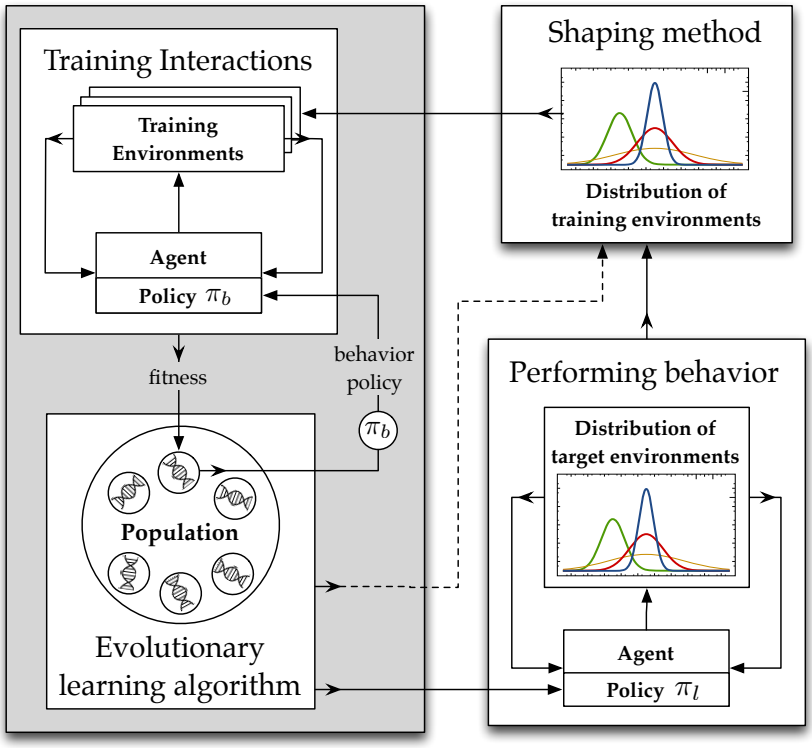


Figure 7.3: A general scheme of shaping in evolutionary multi-task learning.

How to choose training tasks?

In case of evolving cross-task policies for generalized domains with large or infinite set of tasks, fitness can be evaluated by averaging the rewards obtained only in a limited number of tasks. The question that one needs to answer when designing such a fitness function is: how to choose these evaluation (training) tasks? The answer to this question is of utmost importance, as it is the distribution of tasks that allows us to shape the fitness landscapes and, accordingly, guide the search process.

The unshaped approach

A straightforward answer to this question is to select the evaluation tasks by sampling the set of tasks $\mathcal{T}$ according to the given target task distribution $\mathcal{P}$. Evaluating a policy using such a sample of tasks allows to calculate an unbiased estimate of the expected return in the entire domain. Employing the target task distribution for training purposes will be from now on called the *unshaped* approach. A similar approach was recently used by Chong et al. [33], who aimed at maximizing the expected utility of game-playing policies. To this end, they evaluated the fitness of policies by averaging their score against uniformly sampled opponents.

7.2.3 *Shaping in Generalized Domains*

The shaping approach

In contrast to the unshaped approach, which evaluates fitness using the target task distribution, our shaping approach employs alternative (training) task distributions while preserving the same goal of learning. We expect that it may be beneficial to distort the fitness function of the learning problem, even if the objective function is known and can be precisely approximated. Most of the proposed methods employ many training task distributions at successive stages of learning. The motivation for such methods corresponds to that of incremental evolution (cf. 7.1.2) — we attempt to gradually shape difficult fitness landscape to make the problem easier to solve by evolutionary algorithms.

General shaping scheme

Following the abstract scheme of shaping illustrated in Figure 4.1, Figure 7.3 presents a more concrete blueprint for shaping, which particularly involves an evolutionary algorithm in a multi-task setting. As illustrated in the figure, the goal of learning is to perform well in the entire domain defined by the probability distribution over the set of target tasks (environments). The shaping method makes use of this target distribution and provides the learning algorithm with a modified training distribution. Additionally, the role of the shaping method may be more dynamic by exploiting the feedback from the learning algorithm (illustrated with a dashed line). For instance, the provided training distribution may depend on the average performance of the policies learned so far.

Task selection criteria

The main issue is how to identify such training tasks that allow for efficient learning. Since the number of tasks in a domain is typically very large, selecting good evaluation tasks manually just by inspecting their specification is practically infeasible. Thus, in the following sections we introduce an easily computable measure of task difficulty, which allows to group and order tasks within a domain. On this basis it is possible to reasonably select training tasks even manually. However, our main objective is to design unsupervised shaping methods that avoid incorporating human knowledge at all.

7.2.4 *Task difficulty*

Intuitive notion of difficulty

Shaping methods require measurable criteria according to which the training tasks could be selected. For this purpose, we introduce the measure of *expected task difficulty*, that can be calculated without reference to any external source of knowledge. Intuitively, an MDP task (environment) can be said to be *difficult* if a random policy is expected to get a low return in it. On the other hand, *easy* tasks generally allow policies to get higher expected returns. Clearly, tasks within a domain may differ with respect to their difficulty, and thus an inherent characteristic of a domain is its *difficulty distribution*.

Formalized measure of difficulty

In order to formalize and measure difficulties of tasks within the given domain $\mathcal{G} = \langle \mathcal{T}, \mathcal{P} \rangle$, we make the following assumptions:

1. All tasks in $\mathcal{T}$ share the same state space S and action space A, so we can define a space of *domain policies* $\Pi = \{\pi \mid \pi : S \to A\}$.
2. All tasks in $\mathcal{T}$ are episodic and and the total reward per episode is limited from above by a domain-specific constant C.

Without loss of generality, we can assume that $C = 1$ and define the difficulty of task τ as the following function $D : \mathcal{T} \to \mathbb{R}$

$$D(\tau) = D_\Pi(\tau) = \mathbb{E}\left[1 - J(\pi, \tau) \mid \pi \in \Pi\right]. \tag{7.3}$$

Approximate task difficulty

Typically, in non-trivial problems it is computationally infeasible to calculate task difficulty explicitly, because the space of policies is very large and transition function may be nondeterministic. Instead, we can only measure *approximate task difficulty* by using a finite sample of policies $P \subset \Pi$:

$$\hat{D}_P(\tau) = \frac{1}{|P|} \sum_{\pi \in P} \left(1 - J(\pi, \tau)\right). \tag{7.4}$$

If P is uniformly sampled from $\mathcal{P}$, $\hat{D}_P$ is an unbiased estimator of D.

Domain difficulty distribution

The measure of task difficulty allows us to estimate the *domain difficulty distribution* (*difficulty distribution* for short), i.e., the distribution of tasks in a given domain with respect to their approximated difficulty. To estimate such distribution, we discretize the range of task difficulty, which by assumption 2 is $[0, 1]$, splitting it into family of n disjoint intervals of equal width, i.e, $\mathcal{B} = \{B_i \mid i = 0, \ldots, n-1\}$, where:

$$B_i = \begin{cases} \left[\frac{i}{n}, \frac{i+1}{n}\right) & i = 0, ..., n-2 \\ \left[\frac{n-1}{n}, 1\right] & i = n-1 \end{cases} \tag{7.5}$$

Each such interval constitutes a *difficulty bin* to which tasks from a domain can be assigned according to their (approximate) difficulty. As a result, we can identify a subset of tasks of a given difficulty:

$$\mathcal{T}_B = \{\tau \in \mathcal{T} \mid D(\tau) \in B\}. \tag{7.6}$$

Estimating difficulty distribution

Figure 7.4 illustrates the process of estimating difficulty distribution by drawing a sample of tasks T from a domain and dividing them into subsets $\{T_B \mid B \in \mathcal{B}\}$ according to Equation 7.6. Instead of using the exact task difficulty D, it is approximated by sampling the space of policies and calculating the approximate difficulty (see Equation 7.4). Eventually, by counting tasks assigned to particular subsets, we can visualize difficulty histogram to roughly assess the difficulty distribution.

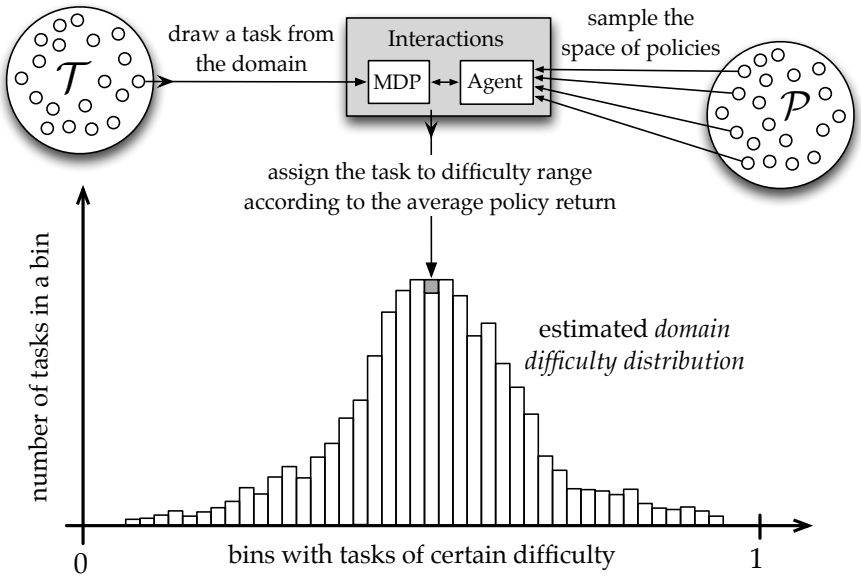


Figure 7.4: An outline of estimating domain difficulty distribution.

7.2.5 *Difficulty-Based Task Pool*

Training difficulty distribution

As already mentioned, the concept of expected task difficulty is introduced in order to equip the shaping method with measurable criteria for selecting the training tasks. Indeed, most shaping methods proposed in this chapter work by providing training tasks according to a *training difficulty distribution*, that intentionally differ from the domain difficulty distribution, but both are defined over the common set of difficulty bins $\mathcal{B}$. Given such a distribution $\mathcal{X}$, each time the learning algorithm requires an evaluation task, a difficulty bin is sampled $B \sim \mathcal{X}$ and a shaping method supplies a task τ of a corresponding difficulty, i.e., such that $D(\tau) \in B$. Since generating *ad hoc* a task of a desired difficulty is generally nontrivial, a precomputed *difficulty-based task pool* is employed for this aim.

Building the task pool

The task pool is created once for a domain and it can be reused by any difficulty-based shaping method. Ideally, for each difficulty bin B the task pool contains a corresponding task set T_B of the same size N, i.e., $\forall_{B \in \mathcal{B}} |T_B| = N$. Building such a pool can be realized similarly to estimating domain difficulty distribution (see Fig. 7.4): a task is drawn from a domain, its difficulty is approximated by interactions with randomly sampled policies, and it is assigned to a corresponding difficulty bin B and added to the task set T_B. This process is repeated until all tasks sets reach capacity N. Typically some task sets can be easily filled, but filling others is computationally infeasible. For this reason, in practice we limit the task pool difficulty range, so that e.g. $\cup\mathcal{B} = [0.1, 0.9]$, to avoid necessity of generating the extremely easy or the difficult tasks which are few and far between.

Task pool applications

Building difficulty-based task pool can be computationally demanding, especially when both N and the number of bins are high. Nevertheless, this is a one-off process — once the pool task sets have been filled, they can be used *ad infinitum* for shaping in this domain. Importantly, such task pools have more applications than just shaping. In particular, they can be used for multi-criteria assessment of learned policies (cf. Section 7.4.3.3). Indeed, each difficulty bin B can be associated with a single criterion that indicates the performance of a given policy in tasks ($\mathcal{T}_B$) of this difficulty. We employed this idea to propose *performance profiles* which are essentially a multi-objective way of evaluating solutions for test-based problems [94].

7.2.6 *Difficulty-Based Shaping Methods*

Implementation of difficulty distributions

In this section, we propose several difficulty-based shaping methods which act as providers of training tasks that are then used by the evolutionary algorithm to calculate fitness of evolving policies (cf. Fig. 7.3). Most of these methods maintain training difficulty distributions (in short: training distributions) defined over the set of difficulty bins $\mathcal{B}$. On a technical note, the distributions are realized by means of difficulty-based task pools. Thus, each time a set of training tasks is needed, they are sampled from the pool according to the specific training distribution. In all shaping methods introduced here, sampling with replacement is used, so a task can be sampled many times. The number of training tasks sampled at each generation, typically much smaller than the number of tasks in the pool, is one of the learning algorithm's parameters.

For instance, drawing a sample task from a uniform difficulty distribution $\mathcal{U}(0.5, 0.9)$ boils down to drawing a task from the union of corresponding task sets in the pool $U = \{\tau \in T_B \mid B \subset (0.5, 0.9)\}$. In the following we will assume that there are $n = 100$ difficulty bins and each of them covers difficulty range of width 0.01. We will use percentage points to identify the sets of task in the pool, i.e, we will refer to union U by saying that task difficulty is in range $(50, 90)$.

Taxonomy of shaping methods

In Fig. 7.5, we propose a taxonomy of difficulty-based shaping methods. Most importantly, we divide them into *static* and *dynamic*. Static methods need a difficulty-based task pool and one or more predefined training distributions, which are used in a fixed order during training. Such methods strongly rely on the selection and ordering of training tasks as specified by an experimenter. Dynamic methods, by contrast, do not require so much effort when setting up, because they attempt to provide training tasks adaptively, on the basis of the current level of learners. They employ to this aim an additional feedback from the learning algorithm.

In the following subsections we describe particular shaping methods in detail.

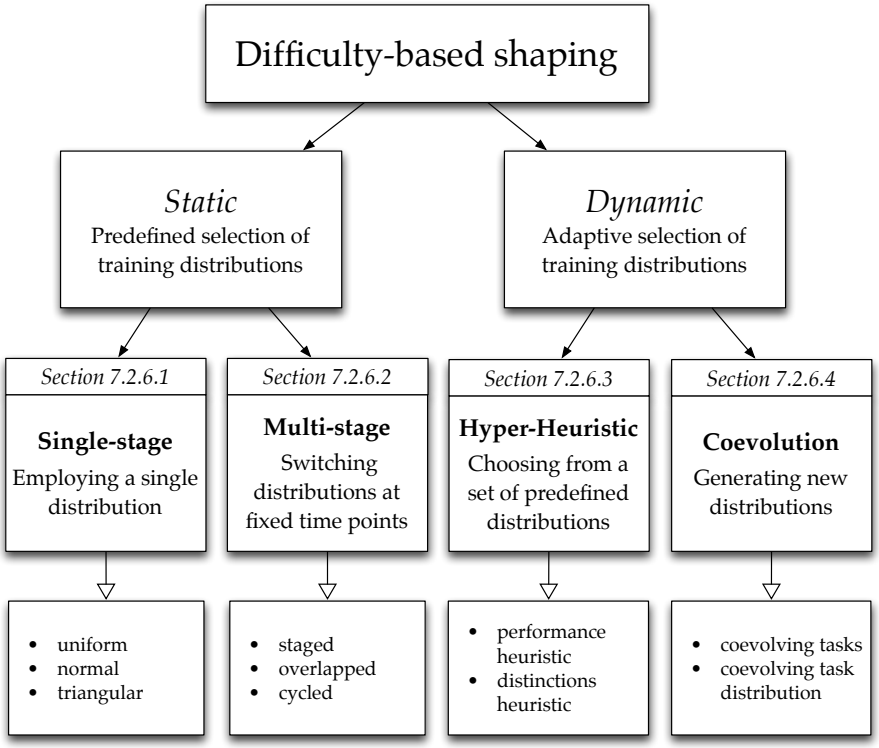


Figure 7.5: A classification of difficulty-based shaping methods.

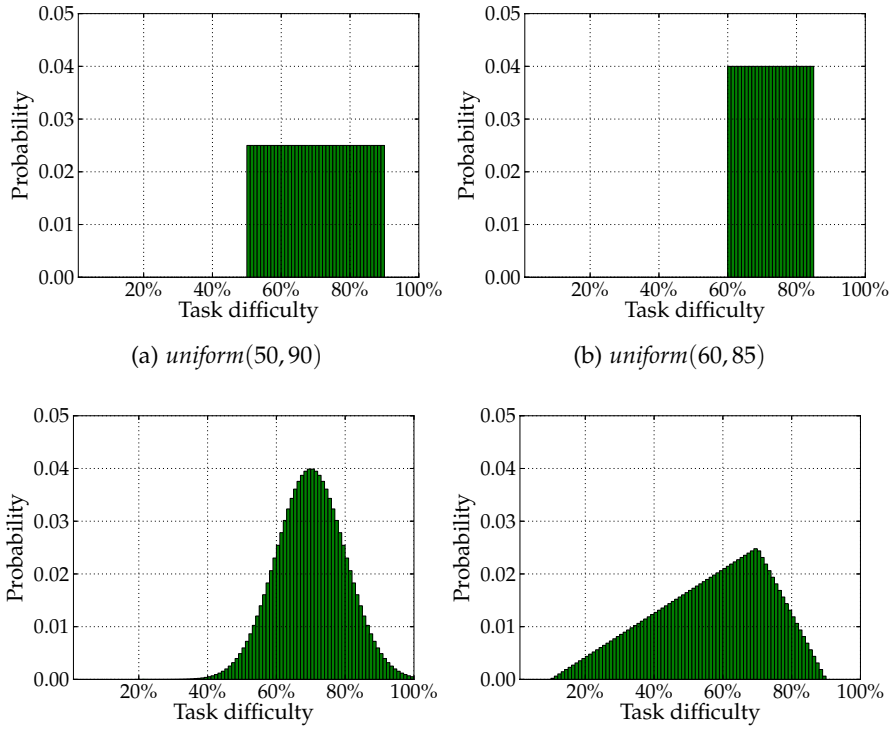
7.2.6.1 *Single-Stage Shaping*

Single-stage shaping is the simplest of the considered approaches and consists in replacing the domain difficulty distribution with a predefined training distribution and using that distribution for the entire learning process. In this work, we employ three well-known distributions, illustrated in Fig. 7.6:

- *uniform* distribution in the specified range of difficulty $[a, b)$, denoted as $\mathit{uniform}(a, b)$ (Fig. 7.6a and 7.6b),
- *normal* distribution, specified by its mean μ and standard deviation σ, denoted as $\mathit{normal}(\mu, \sigma)$ (Fig. 7.6c),
- *triangular* distribution, specified by range $[a, b)$ and mode c, denoted as $\mathit{triangular}(a, c, b)$. (Fig. 7.6d).

7.2.6.2 *Multi-Stage Shaping*

Multi-stage shaping uses a sequence of training distributions which are switched as learning proceeds. As a result, difficulty of provided training tasks varies with the generation of the evolutionary algorithm. This approach allows to create a series of distributions characterized by progressive difficulty, what realizes the shaping as meant in most works on incremental evolution (cf. Section 7.1.2).



(a) *uniform*(50, 90) (b) *uniform*(60, 85)

(c) *normal*(70, 10) (d) *triangular*(10, 70, 90)

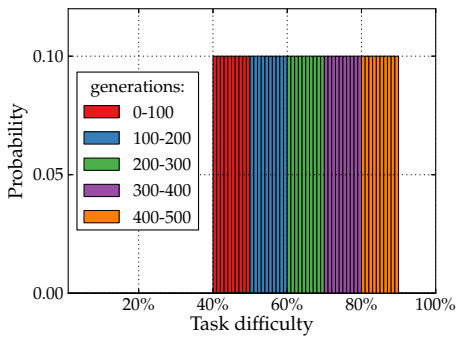
Figure 7.6: Examples of training difficulty distributions.

Component training distributions

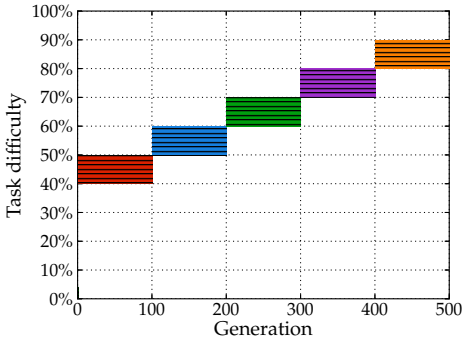
We assume that a multi-stage shaping method holds k fixed *component training distributions* (components) and uses them in a prescribed order during learning. We employ only uniform training distributions as components. We consider the following variants of multi-stage shaping (see Fig. 7.7) :

- *staged*(a, b, k) — each component training distribution features the tasks with difficulty from an interval of width $(b - a)/k$. The components' intervals do not overlap and cover together the specified range of difficulty $[a, b)$. Each stage lasts the same number of generations and corresponds to a part of the difficulty range. Figures 7.7a and 7.7b show how the *staged*$(40, 90, 5)$ method works if the learning algorithm runs for 500 generations. For instance, the first component training distribution *uniform*$(40, 50)$ is used for the first 100 generations.

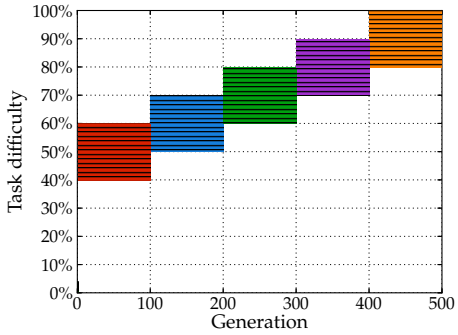
- *overlapped*(a, b, k, w) — a variation of the staged method in which component training distributions cover parts of the difficulty range of specified width w and thus may partially overlap. Figure 7.7c shows *overlapped*$(40, 90, 5, 20)$ using uniform component distributions in the ranges $[40, 60)$, $[50, 70)$, $[60, 80)$, $[70, 90)$, $[80, 100)$.



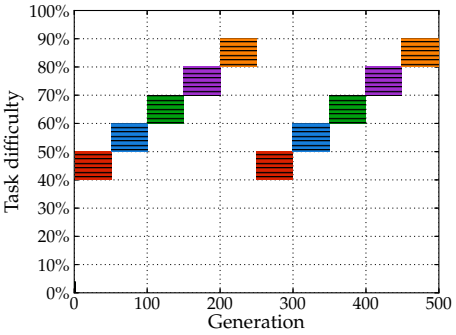
(a) Probability mass function for task difficulty in *staged*$(40, 90, 5)$.



(b) Difficulty progression over generations in *staged*$(40, 90, 5)$.



(c) Difficulty progression over generations in *overlapped*$(40, 90, 5, 20)$.



(d) Difficulty progression over generations in *cyclic*$(40, 90, 5, 50)$.

Figure 7.7: Training distributions provided by multi-stage shaping methods.

- *cyclic*(a, b, k, g) — a cyclic variation of the staged method with non-overlapping difficulty intervals of the component distributions. Each component is used for g successive generations after which it is replaced by the next one. After every $k \cdot g$ generations the cycle is repeated starting from the first component. For example, Fig. 7.7d illustrates *cyclic*$(40, 90, 5, 50)$.

7.2.6.3 *Hyper-Heuristic Shaping*

This shaping method is inspired by the hyper-heuristic approach, in particular by the class of methods which is referred to as 'heuristics to choose heuristics' [26]. The general idea is to provide a fixed set of low-level heuristics to a high-level selection method which attempts to switch between them. Most importantly, the selection method is *responsive* by relying on a feedback from the low-level heuristics.

Selection heuristic

To apply this idea for shaping, we allow a set of component training distributions (see previous subsection) to act as low-level heuristics. Although, strictly speaking, they are not search operators, they influence the search process by changing the fitness function.

Low-level heuristics

Component distributions

Similarly to the *staged* shaping approach we provide a set of k uniform component training distributions covering given range of difficulty $[a, b)$ with non-overlapping difficulty intervals. However, in contrast to *staged* shaping there is no predefined order of using them. Every g generations a high-level selection heuristic makes a decision and chooses which distribution will be used for subsequent g generations. For this aim, we employ two different selection heuristics:

- *distinctions*(a, b, k) — at each generation ($g = 1$) this selection heuristic samples the tasks from the component training distributions, counts how many *distinctions* between policies in the current population makes each of them and selects the component which resulted in the largest number of distinctions. The concept of distinctions is borrowed from coevolutionary algorithms [41, 58]. Here we will say that a task $\tau \in \mathcal{T}$ distinguishes between two policies $\pi_a, \pi_b \in \mathcal{P}$ if and only if policy π_a gets significantly higher return in this task than policy π_b:

$$dist(\tau, \pi_a, \pi_b) \iff J(\pi_a, \tau) - J(\pi_b, \tau) \geqslant Y, \tag{7.7}$$

  where Y is a problem-specific constant $0 < Y \leq C$ defined by the experimenter. By making distinctions between policies, tasks are believed to build a learning gradient. To check if a nondeterministic task makes a distinction, it is necessary to simulate more than one interaction episode with each policy and compare the estimated expected policy returns.

- *performance*(a, b, k, g) — this selection heuristic maintains statistics describing the influence of using particular component training distribution on the performance of evolving policies. For instance, the performance can be measured as an average policy return on the target task distribution. Every g generations the selection heuristic inspects the most recent use of each component distribution and chooses the one which resulted in the largest performance improvement.

Computational effort

Note that, when compared to other shaping methods, the hyper-heuristic ones need an extra computational effort for invoking the selection heuristic. However, in our experiments we ignore this additional effort and count only the interactions performed for the purpose of fitness evaluation.

7.2.6.4 *Coevolutionary Shaping*

Hyper-heuristic shaping is responsive by using the feedback from the learning process to select the component training distributions. However, those distributions remain fixed, as they were predefined prior to learning. Therefore, the overall training experience provided

by hyper-heuristic shaping is still limited. In certain contexts, it is possible to make the training episode, meant as an interaction between a learner and task, to affect not only the former, but also the latter. In this way, the task difficulty distribution can be dynamically adjusted to the capabilities of learners.

Adaptive training distributions

The idea of adaptive adjustments of training difficulty distributions can be implemented by means of coevolutionary algorithms, which rely on viewing a multi-task reinforcement learning problem as a test-based problem (cf. Section 7.2.2). In this context, in a two-population coevolutionary algorithm (see Section 4.2.1), the role of shaping is played by a population of tests which attempts to adaptively provide a learning gradient for coevolving policies. Tests represent tasks and their fitness is evaluated by counting distinctions these tasks make in the latest population of policies. Distinctions are defined as in distinctions-based hyper-heuristic approach (see Section 7.2.6.3).

Two-population coevolution

In contrast to other shaping approaches, coevolutionary shaping is self-sustaining because it does not require predefined training task distributions. On the other hand, to effectively evolve useful tasks, we need to specify tests population details including individual representation, selection scheme and variation operators. In particular, we propose the two following variants of coevolutionary shaping which differ in test representation:

Requirements

- *coev-task* — A conventional application of coevolutionary algorithm to a test-based problem, in which the individuals in the test population embody tasks. Tasks undergo conventional evolutionary workflow that includes selection and domain-specific mutation and crossover operators. Importantly, this is the only shaping approach considered here which does not require a precomputed difficulty-based task pool. Instead, it attempts to synthesize tasks by itself and to scale their difficulty online to provide increasingly complex challenges while the learning progresses.

- *coev-diff* — an alternative idea is to assume that the tests embody not single tasks, but entire training difficulty distributions. For simplicity, we consider here a simple variant of this approach, where a tests represents a single difficulty bin (cf. Section 7.2.4). In such case, an interaction between a candidate solution (policy π) and a test (difficulty bin $B \in \mathcal{B}$) consists in drawing a task $\tau \in T_B$ from a corresponding set in a difficulty-based task pool and calculating the policy return $J(\pi, \tau)$. Although, in contrast to coevolution of tasks, this approach requires a precomputed task pool, there is no need of choosing a specific training distribution.

7.3 EMPIRICAL EVALUATION OF SHAPING METHODS

Evaluation overview

In the following sections we demonstrate how shaping methods perform in three different multi-task domains. The first two domains concern the board game of Othello (see Section 5.1), while the third one is based on the cart pole-balancing problem (described in Section 5.3). Sections 7.4 to 7.6, which present the results of applying shaping methods to particular domains, are organized in the same manner. In each of them we start by defining a multi-task domain and estimating the domain difficulty distribution. Afterwards, we report the outcomes of computational experiments and apply statistical analysis to compare the performance of policies learned with different shaping methods. Shaping methods turn out to significantly improve evolutionary learning in a multi-task domain, and this result constitutes one of the main contributions of this thesis.

Common learning algorithm

Accordingly to one of the main assumptions of our shaping approach, the learning algorithm itself is not a subject to modification, so it does not change between particular methods. For all experiments we use the $(\mu + \lambda)$ evolution strategies described in Section 2.2.3, and presented as a pseudocode in Listing 2.3. The algorithm begins with a population of $\mu + \lambda$ randomly generated individuals — real vectors of parameters for neural networks that represent behavior policies. In every generation, each of the μ fittest individuals produces λ/μ offspring through a straightforward uniform perturbation operator. Detailed experimental setups are provided in the following sections.

Differences in fitness assignment

We emphasize that in all setups within a single domain the evolutionary operators of selection and mutation and their parameters are the same. The differences between them lie only in the choice of training tasks for fitness evaluation. The baseline for all comparisons is the conventional *unshaped* approach, in which fitness is calculated as the average policy return in a sample of tasks drawn directly from the target domain according to domain distribution. All the other considered setups employ the shaping methods (see Section 7.2) to provide evaluation tasks in a different way. In particular, we examine the performance of static shaping methods, in which tasks are sampled from predefined difficulty distribution(s) and dynamic shaping methods, like coevolution, which provide tasks adaptively to build the learning gradient for evolving policies.

Performance measure

Despite using different fitness functions to drive evolution, the ultimate goal of learning remains the same: maximization of expected utility, i.e., the expected policy return in tasks from a given domain. To objectively assess how well the evolved individuals perform on that measure, we use the approximate measure of expected utility. This measure is the average return obtained by a policy in 25 000 interaction episodes against the tasks drawn from a target domain. From now on, the term 'performance' refers to this measure.

7.4 OTHELLO OPPONENT DOMAIN

The first considered domain involves the game of Othello described in Section 5.1. The goal of learning is to find a game-playing policy that copes well against all possible Othello opponents. The multi-task learning reinforcement problem is defined as follows:

- *Task set* $\mathcal{T}$. Each task τ is an instance of Othello game with a fixed opponent policy π_o. Tasks across the domain vary only with respect to the opponent which determines the transition function of the underlying MDP. Since all other aspects of tasks, including state space, action space and reward function, remain constant, a task is posed by a game-playing policy of the opponent. Here we assume that policies are represented by Weighted Piece Counters (WPCs) as described in Section 5.1.2.2. As a result, the space of tasks is equivalent to the parametric policy space where each point represents a vector of 64 real-valued WPC weights. Besides that, each task has two possible initial states corresponding to playing white or black, respectively[1].

- *Task distribution* $\mathcal{P}$. Each element from the space of tasks $\mathcal{T}$ is equally likely to be drawn. Note however, since tasks are encoded as vectors of policy parameters, multiple vectors may result in the same policy, i.e., the same state-action mapping. To generate a *random task*, a random opponent policy is created by drawing WPC weights uniformly from the interval $[-10, 10]$.

- *Domain policies* Π. To represent a game-playing policy (candidate solution to the learning problem) we employ the WPC architecture to act as a state evaluator in 1-ply setup (see Section 5.1.2.2). As a result, the parametric policy space is identical to the space of tasks, and a *random policy* is generated in an analogical way — by uniformly drawing WPC weights. Random policies and random tasks are crucial for estimating domain difficulty distribution and constructing difficulty-based task pool. Domains like this one, in which the roles of tasks and policies are equivalent and thus interchangeable can be regarded as *symmetric*. Importantly, in such domains difficulty of a task is strictly related to the performance of its opponent policy.

- *Interaction episode*. An interaction episode corresponds to a single game of Othello between a candidate solution policy and an opponent policy π_o defined by the task. The maximal total reward that can be received by a policy is $C = 1$ when it wins the game. In case of draw a policy gets reward equal to 0.5 and there is no reward if the game is lost.

1 One could argue that playing versus black is a different task than playing versus white player. Following this reasoning, we could alternatively encode a task as an opponent policy and a color of player for which it is used.

Setting	Value
Learning algorithm	$(\mu + \lambda)$ Evolution Strategy
Population size	$\mu = 25,\ \lambda = 25$
Initialization	$\sim \mathcal{U}(-0.2, 0.2)$
Mutation type	uniform perturbation
Mutation strength	$\delta = 0.1$
Number of training tasks	50
Episodes per generation	5 000
Generations	500
Number of runs	100

Table 7.1: Experimental settings of evolutionary learning in the Othello opponent domain.

7.4.1 *Experimental Setup*

Learning algorithm parameters

As the learning algorithm, we employ $(\mu + \lambda)$ generational evolution strategy, with $\mu = 25$ and $\lambda = 25$, to evolve vectors of 64 real-valued WPC weights. The algorithm starts by filling the initial population with individuals whose weights are randomly drawn from the range $[-0.2, 0.2]$. The only search operator is a simple uniform perturbation mutation that modifies all the weights using additive noise. The WPC weight $w_i^{'}$ of the offspring is obtained by adding a small random value to the corresponding WPC weight of the parent:

$$w_i^{'} = w_i + 0.1 \cdot \mathcal{U}(-1, 1), \tag{7.8}$$

where $\mathcal{U}(-1, 1)$ is a real value drawn uniformly from the range $[-1, 1]$. Weights resulting from mutation are clamped to the interval $[-10, 10]$, effectively keeping the value within the respective bound. The experimental settings are summarized in Table 7.1.

Evaluation phase details

In the evaluation phase, each individual is faced with 50 tasks provided by the shaping method. In the case of unshaped learning, tasks are generated uniformly according to the target task distribution $\mathcal{P}$. Importantly, since each task $\tau \in \mathcal{T}$ has exactly two possible initial states, it is explicitly used in two successive interaction episodes starting from these states. In other words, when a policy faces a task it plays one game as black and one game as white player. We use the term 'double game' to refer to such a pair of interactions. The average policy return in a total of 100 episodes is used to determine individual's fitness.

Computational effort

Each method requires the same computational effort of 5 000 interaction episodes per generation ($(25 + 25)$ individuals $\times$ $(50 + 50)$ training tasks). Each evolutionary run lasts for 500 generations, what makes the total effort of 2 500 000 episodes (games) per run. For the

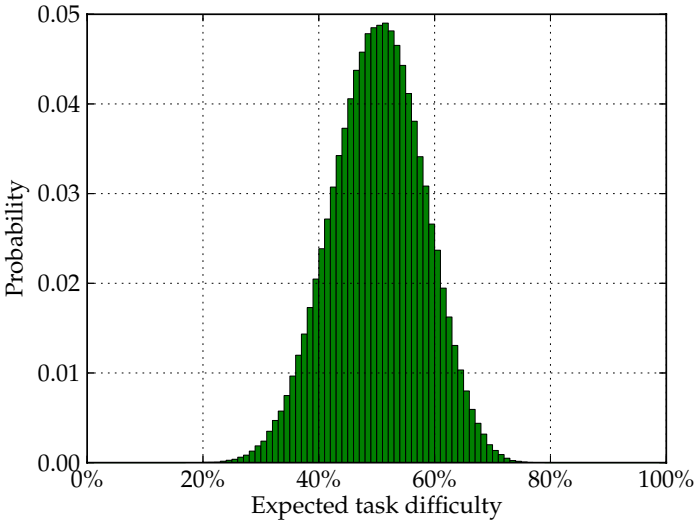


Figure 7.8: Domain difficulty distribution of the Othello opponent domain.

purpose of statistical analysis, we performed 100 independent runs for each considered shaping method. The detailed results of Kruskal-Wallis and Mann-Whitney tests can be found in Appendix A.1. The *best-of-generation* individual is the individual with the highest fitness in the population. By the *best-of-run* player we mean the best player of the last generation. We characterize method's performance using the performance of its best-of-generation player.

7.4.2 *Domain Difficulty Distribution*

To get insight into properties of tasks in the considered domain, we estimated the domain difficulty distribution (see Section 7.2.4). Distribution illustrated in Fig. 7.8 was obtained by sampling 500 000 random tasks and calculating their approximate difficulty (see Equation 7.4) on the basis of double games with a set of 1 000 random policies. Task's difficulty determines its assignment to one of 100 difficulty bins, which correspond to difficulty ranges of width 1%.

Estimating a distribution

Clearly, the resulting histogram resembles the normal distribution with a mean value of 50.2% and a standard deviation of 7.91%. Since this particular domain is symmetric and thus both random tasks and random policies are sampled from the same space, we can interpret the expected difficulty of a random task as the expected policy return of a random (opponent) policy. In this context, the figure shows the expected performance of a random policy. The mean value of around 50% expresses the intuitive fact that a random policy is equally likely to win and lose the game against another random policy. Moreover, the estimated distribution indicates that most tasks in this domain have close to average difficulty — strong opponent policies are few and far between.

Task difficulty interpretation

Motivation for shaping

The estimated difficulty distribution strongly encourages the use of shaping. As the unshaped approach samples the opponents uniformly from this distribution, it is unlikely for an individual to face a difficult task in the evaluation phase. For this reason, the fitness function will be often unable to differentiate between two policies that, e.g., play well against the average opponents but one of them is much better at beating the strong ones. This is unfortunate as, preferably, we would like to lead the evolution towards the policies that are able to win with skilled players without losing the capability of winning with the opponents of average strength.

Importantly, the difficulty distribution indicates also that finding the extremely difficult or easy tasks by random sampling is very hard. This has substantial implications for building the pool of tasks used by most of the proposed shaping methods. For that reason, we were unable to fill all task sets T_B in a difficulty-based task pool (cf. Section 7.2.5). To overcome this obstacle, the tasks of difficulty $\geq$ 81% or $\leq$ 13% were obtained by multiple independent runs of dedicated evolutionary algorithm. In this way, we filled 80 task sets, each of size $N = 1\,000$, corresponding to difficulty bins in range $[10\%, 90\%)$.

Difficulty-based task pool

7.4.3 *Single-Stage Shaping Methods*

Distribution parameters

In the first experiments, we examined the single-stage shaping methods which provide the training tasks with the same difficulty distribution for the entire learning process. Each of these methods, including uniform, normal and triangular shaping (see Section 7.2.6.1), requires a small number of parameters that determine the specific task distribution. Although in principle we could perform a systematic search in a parameter space to tune the method for the problem at hand, the aim of this thesis is to demonstrate the usefulness of shaping rather than meticulously tune method parameters. Therefore, for the sake of simplicity we rely on an arbitrary series of preliminary experiments to achieve reasonable method settings.

7.4.3.1 *Overall Performance*

Violin plot

The results of the single-stage shaping methods with selected parameters are presented in Fig. 7.9 as a violin plot, which combines a box plot and a density trace [87]. The plot summarizes the distribution of final performance obtained with particular methods.

The most important observation is that each of the proposed shaping methods allowed evolutionary algorithm to learn stronger policies than those found with the conventional unshaped approach. However, the performance of particular shaping methods largely depends on the selected parameters of training task distributions. If these parameters are not chosen properly, shaping performs significantly worse than the reference approach. In the case of this specific prob-

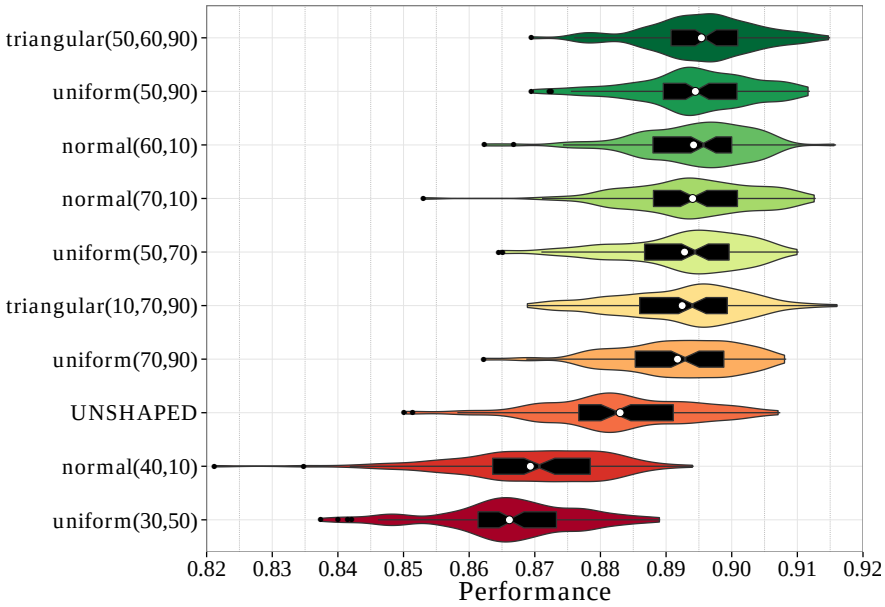


Figure 7.9: Performance of **single-stage shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted descendingly according to their mean performance (shown by white circles).

lem, we can generally observe that the learning algorithm benefits from biasing distributions towards more difficult task and limiting time number of easier tasks. For instance, the $\mathit{uniform}(30, 50)$ and $\mathit{normal}(40, 10)$ approaches, which focus solely on tasks of average difficulty, are the weakest among considered methods. On the other hand, the most of the successful configurations completely ignore the tasks of difficulty lower than 50% and put emphasis on training only in more challenging environments.

Impact of training distributions parameters

Nevertheless, the relative improvement gained by the best shaping methods in comparison to unshaped approach is barely larger than 1%. We conducted a statistical analysis to verify our observations and determine whether using shaping methods has a significant effect on the performance of learned policies [42]. On the basis of density plots we assumed that particular samples of results have distributions of similar shape (except for any difference in medians). Under this assumption, we performed a Kruskal-Wallis test which confirmed ($p < 0.001$) a significant difference in medians. A post-hoc analysis using one-sided Mann-Whitney tests with Holm correction measured the significance of pairwise differences between particular experimental setups. The results of pairwise comparisons are shown in Table 7.2.

Statistical analysis

	triangular(50,60,90)	uniform(50,90)	normal(60,10)	normal(70,10)	uniform(50,70)	triangular(10,70,90)	uniform(70,90)	unshaped	normal(40,10)
uniform(50,90)	1.000								
normal(60,10)	1.000	1.000							
normal(70,10)	1.000	1.000	1.000						
uniform(50,70)	1.000	1.000	1.000	1.000					
triangular(10,70,90)	0.485	1.000	1.000	1.000	1.000				
uniform(70,90)	0.106	0.485	0.532	0.872	1.000	1.000			
unshaped	0.000	0.000	0.000	0.000	0.000	0.000	0.000		
normal(40,10)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
uniform(30,50)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.228

Table 7.2: p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction. Each value lower than the significance level $\alpha = 0.01$ indicates that a method in a row has significantly lower performance than another method in the corresponding column.

7.4.3.2 *Learning Speed*

Learning speed assessment

The results discussed above concern the final performance of the considered methods, but they do not describe how this performance was achieved. To get more insight into the learning process, we assessed the speed of learning by evaluating the methods not only at the end of evolutionary run, but after each generation. Figure 7.10 shows the performance of selected methods as a function of computational effort (number of training episodes). Each point in the plot is the performance of best-of-generation individuals averaged over 100 runs. Let us emphasize that, although this performance measurement involved a large number of interactions in the target domain, it did not influence individuals' fitness; the learning process was driven only by the tasks from a predefined training distribution.

Performance over time

Because the previous experiment showed that there is little difference in performance between the shaping methods that use similarly biased ('located') triangular and uniform distributions, the learning speed was gauged only for a set of uniform shaping methods and the unshaped approach. Although the considered methods achieve visibly different final performance, all of them except *uniform*$(10, 30)$ learn initially with a similar speed and reach the performance level of 0.8 relatively quickly. Only above this level the learning curves start to diverge and both *uniform*$(30, 50)$ and *unshaped* methods learn slower than the best *uniform*$(50, 70)$ and *uniform*$(70, 90)$, which stay very close to each other throughout the entire evolution. Importantly, all methods steadily improve their performance, and continue doing so even in the late stages of runs.

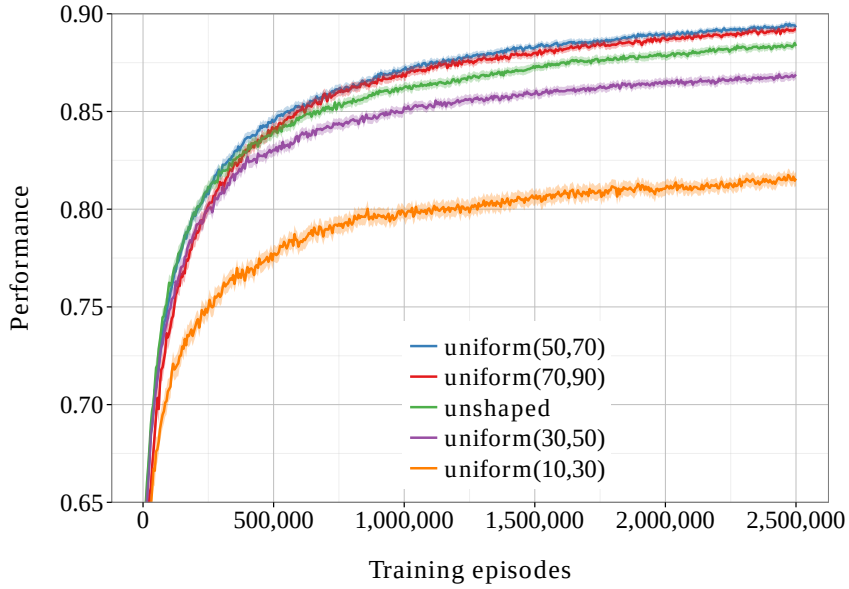


Figure 7.10: The performance of uniform shaping methods over time.

7.4.3.3 *Multi-Criteria Performance Evaluation*

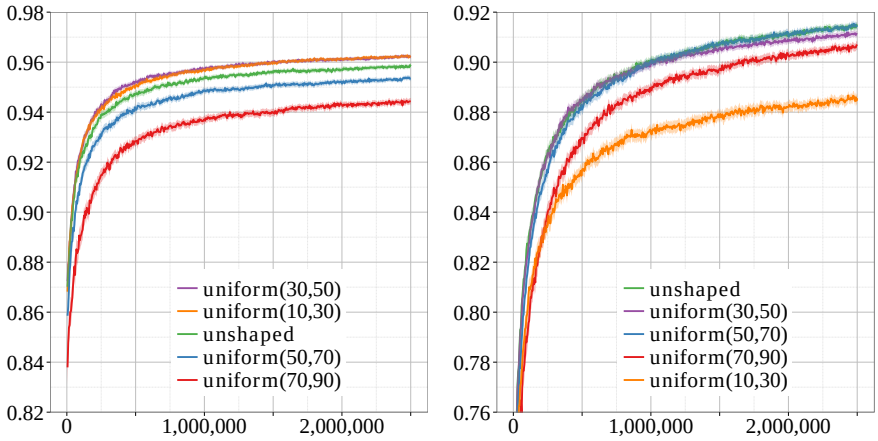
Multi-criteria assessment

In order to better understand the characteristics of particular methods we broke down the performance measure into more detailed information on how policies cope with the tasks of various difficulty. For this purpose we employed the opponents drawn from uniform task distributions not only to evaluate the fitness of evolving individuals, but also to assess the performance of already learned policies.

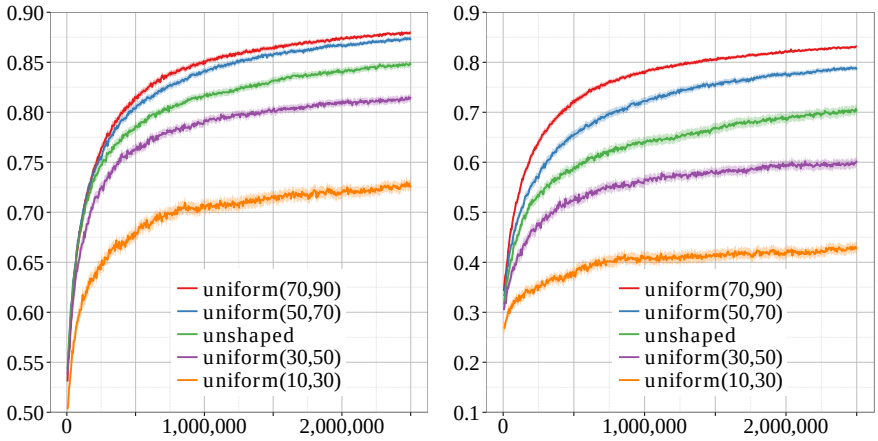
In Figure 7.11 we can observe how the shaping methods perform with respect to four distributions of tasks of increasing difficulty. Each of these distributions can be considered as a separate performance criterion, so that together they describe the characteristics of the developed polices in a more explanatory way than a single aggregated performance measure. This figure sheds new light on the comparison between the unshaped approach and uniform shaping.

Performance in diverse tasks

In particular, let us focus on a pair of methods $\mathit{uniform}(50,70)$ and *unshaped*. Although the difference in aggregated performance between these methods is only around 1% (see Figures 7.9 and 7.10), their performance in tasks of varying difficulty turn out to substantially diverge. The $\mathit{uniform}(50,70)$ method fares almost 10% better in very difficult tasks (see Fig. 7.11d) and 3% better in the moderately difficult ones (see Fig. 7.11c). The unshaped approach, by contrast, is just slightly more rewarding only in very easy tasks, while both methods are comparable in tasks of difficulty in range $[30,50)$. Nevertheless, attaining much higher performance in demanding tasks by shaping methods is not sufficient for them to gain a substantial advantage in terms of the overall performance, because in the target domain such difficult tasks occur much more infrequently.



(a) Performance in $\{T_B \mid B \subset [10, 30)\}$ (b) Performance in $\{T_B \mid B \subset [30, 50)\}$



(c) Performance in $\{T_B \mid B \subset [50, 70)\}$ (d) Performance in $\{T_B \mid B \subset [70, 90)\}$

Figure 7.11: Performance of shaping methods on different task distributions.

Training vs. target task difficulty

An interesting regularity observable in this analysis is that the policies that achieve the highest performance in the given range of task difficulty were those ones that were taught on the slightly harder tasks. For instance, in the difficulty range $[30, 50)$ (see Fig. 7.11b) the top performing policies are developed by *uniform*$(50, 70)$, while in range $[50, 70)$ (see Fig. 7.11c) the best results are achieved by *uniform*$(70, 90)$. In the context of Othello game, this observation confirms an intuitive belief that in order to defeat a given opponent it is useful to master game-playing skills against stronger players. On the other hand, the difference in the level of play between the teacher and the target opponent can not be too large. Indeed, if trained solely by expert opponent one can miss the skills required to deal with the easiest strategies. We hypothesize that for this reason *uniform*$(70, 90)$ is the worst method in terms of the performance on the easiest tasks. Broadly speaking, the fitness evaluated only on difficult tasks is not the best predictor of general multi-task solving abilities.

Clearly, dividing the set of target tasks into subsets of varying difficulty allows to reveal the strong points of particular methods, which could not be noticed using the measure of aggregated performance in the entire target domain. Following this idea, we can further increase the number of task subsets and thus consider even more criteria in such multi-criteria method comparison. In our recent study [94], this idea has been implemented as *performance profiles* which display solution's performance as a function of task difficulty.

Performance profiles

Finally, although in principle the goal of learning is the overall performance in a given domain, in practice we may be more concerned about solving harder tasks. In this context, the above analysis provides convincing evidence that, at least for Othello, we should prefer shaping approach over the unshaped one, albeit at first sight they perform nearly the same on average.

Practical aspects

7.4.4 *Multi-Stage Shaping Methods*

In the next experiment we verify whether the best uniform shaping setup identified in the previous section, i.e., $\mathit{uniform}(50, 90)$, can be improved by employing the multi-stage shaping approach. To this aim, we split the task difficulty range $[50, 90)$ into several intervals and employ them in successive training phases to provide tasks of systematically varying difficulty (see Section 7.2.6.2).

Figure 7.12 compares the performance of selected multi-stage shaping methods with the reference unshaped and $\mathit{uniform}(50, 90)$ approaches. Although multi-stage shaping outperformed the unshaped approach, it did not allow to improve on performance of the single-stage shaping. Moreover, some setups, including $\mathit{staged}(50, 90, 20)$ led to significantly inferior results. Generally, we can conclude that the performance decreases with the growing number of stages and thus with reducing the width of successive difficulty intervals.

Staged shaping

The adverse effect of narrow difficulty intervals used at particular stages of learning can be partially canceled by repeating these stages in a cyclic manner. If the length of the training phase in such cyclic shaping is relatively short, the obtained results closely resemble those of the original single-stage method. This similarity can be noticed in the violin plots of $\mathit{cyclic}(50, 90, 2, 10)$ and $\mathit{uniform}(50, 90)$.

Cyclic shaping

An alternative approach which follows the idea of incremental evolution but avoids the problem of narrow difficulty intervals is overlapped shaping. Let us recall that for instance $\mathit{overlapped}(50, 70, 2, 30)$ divides the learning process into two phases in which it employs the following task distributions: $\mathit{uniform}[50, 80)$ and $\mathit{uniform}[60, 90)$. By employing wide and overlapping ranges of tasks difficulty this method was able to significantly improve the straightforward *staged* shaping. However, when compared with the reference $\mathit{uniform}(50, 90)$ method, there was no significant increase in the mean performance.

Overlapped shaping

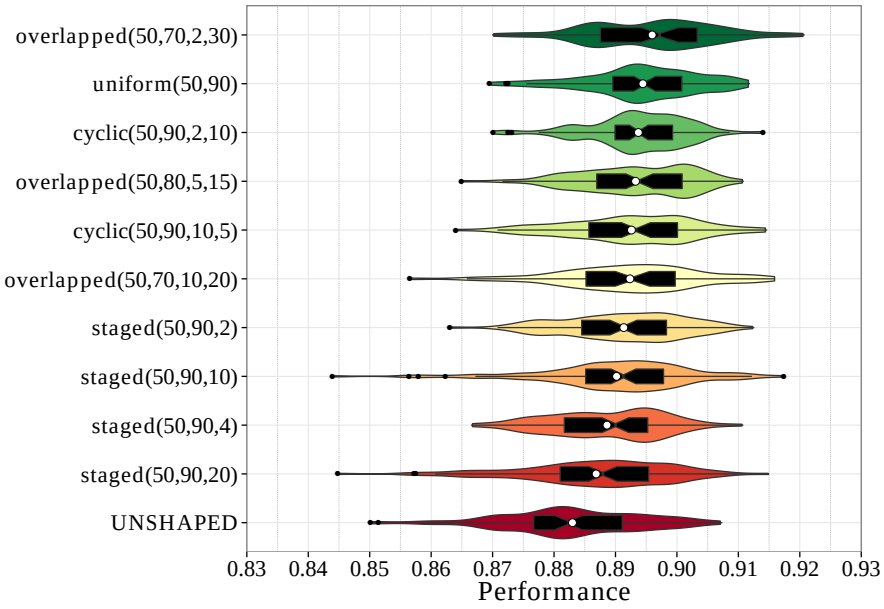


Figure 7.12: Performance of **multi-stage shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

The above remarks are supported by statistical analysis. The results of conducted tests can be found in Table A.1 in Appendix A.1.

7.4.5 *Hyper-Heuristic Shaping Methods*

Towards autonomous shaping

The multi-stage shaping approach relies on a *a priori* selection of training task distributions and, what is even more important, assigning them to successive phases of learning. Manual adjustment of settings that control shaping distributions and their usage during learning requires a substantial amount of human effort and repeated computational experiments. Employing hyper-heuristic methods (see Section 7.2.6.3) is a first step towards autonomous shaping. Here, we still need to equip a shaping method with a set of predefined training distributions, but the method will try to automatically discover when to switch from one distribution to another.

Performance shaping

In the experiments we employ two types of hyper-heuristic shaping, namely *distinctions* and *performance* methods, which differ in how they choose a training distribution to provide the sample of tasks. Figure 7.13 shows that neither of these methods was able to beat the single-stage *uniform*$(50, 90)$ shaping. However, according to statistical

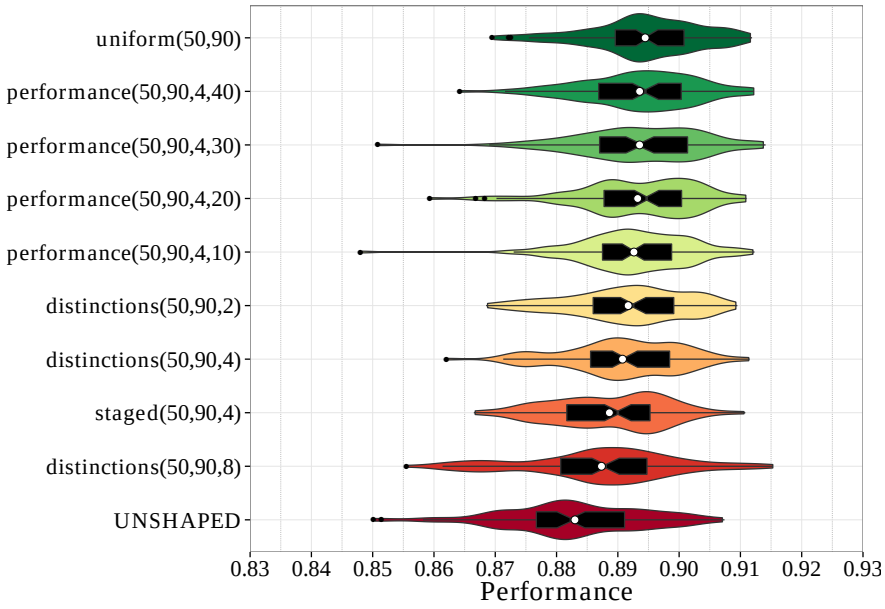


Figure 7.13: Performance of **hyper-heuristic shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

analysis conducted in the same way as in previous sections (cf. Table A.2 in the appendix) we can claim that *performance* shaping significantly improved upon the results of the *staged*$(50, 90, 4)$ method.

Distinctions shaping

The hyper-heuristic methods based on the notion of distinctions, were much more sensitive to its parameter settings. Similarly to the previous experiments, increasing the number of considered task distributions and at the same time reducing the width of corresponding difficulty ranges resulted in decreased performance. Nevertheless, all of the evaluated hyper-heuristic shaping methods were still significantly superior to the conventional unshaped approach.

Unequal computational effort

Let us emphasize that when compared with previously considered shaping approaches, both methods can be regarded as 'unfair' as they require additional interaction episodes for making these choices. The *performance* shaping method evaluates the policies in the target domain, while the *distinctions* method calculates policies return on all predefined distributions and only then chooses the best one. We decided to ignore this extra computational effort and count only interactions performed by the learning algorithm because the primary aim of the experiment was to verify whether the failure of multi-stage shaping may be ascribed to unfortunate assignment of training distributions to successive learning phases.

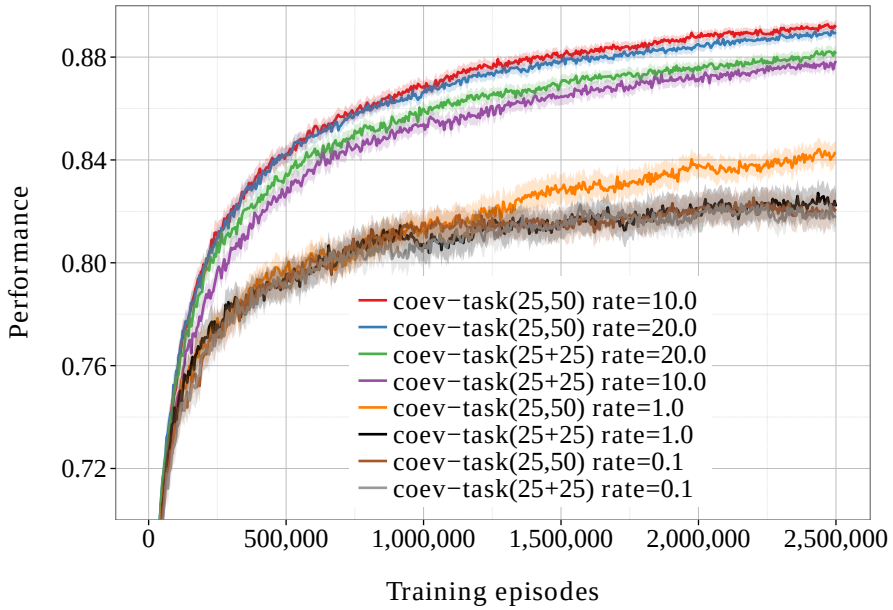


Figure 7.14: Performance of *coev-task* shaping methods with different selection schemes and mutation rates. Semi-transparent ribbons around the curves show 95% confidence intervals for the mean.

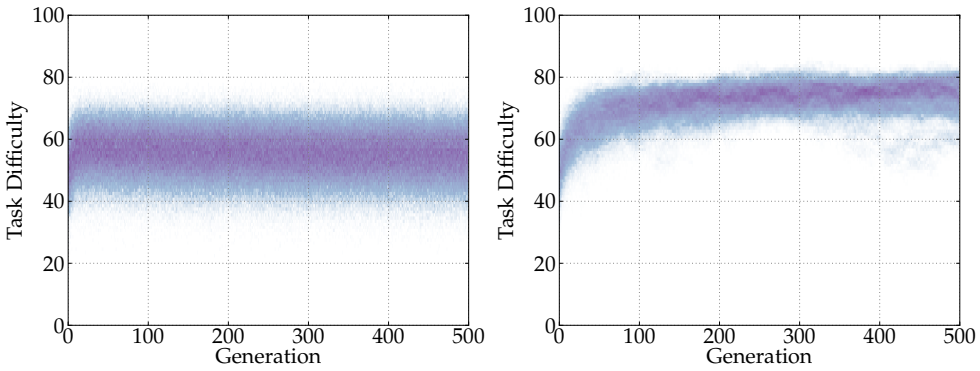
7.4.6 *Coevolutionary Shaping*

Lower requirements

The final experiment in the random-opponent Othello domain consisted in employing coevolutionary shaping (see Section 7.2.6.4). This approach goes one step further beyond hyper-heuristic in terms of being autonomous, since it does not require any predefined training task distribution. However, in both variants of coevolutionary shaping, namely *coev-task* and *coev-diff*, we need to specify how to search the space of tests which, depending on the variant, is represented directly by tasks themselves or indirectly by task difficulties.

Experimental settings

A common experimental setting for both methods was to utilize the distinctions to calculate the fitness of tests. Moreover, we fixed the size of the tests population to be equal to the number of tasks required for policies evaluation, i.e., 50. Therefore, during the evaluation phase, each policy interacted with every test from the coevolving population in a round-robin manner. Ultimately, the experimental setup of coevolutionary shaping methods was limited to two choices concerning the population of tests. First, we had to choose the mutation operator to effectively search the adopted space of tests. Second, as for the selection scheme we considered between $(\mu + \lambda)$ with $\mu = \lambda = 25$ and (μ, λ) with $\mu = 25$ and $\lambda = 50$. The latter one, called *comma* selection strategy, does not allow any of μ parents to be included in the next generation, even if they are better than all λ offspring. As a result, the entire population is replaced at each generation so the tests used to evaluate policies change much faster.



(a) *coev-task*$(25, 50)$, mutation rate = 10.0 (b) *coev-task*$(25 + 25)$, mutation rate = 0.1

Figure 7.15: Difficulty distributions provided by *coev-task* shaping methods over generations. Color saturation indicates relative occurrence frequency of tasks of certain dificulty.

7.4.6.1 *Coev-Task Methods*

Initial coev-task setup

We start with the analysis of the *coev-task* method, where each test individual is a single task. Since in this particular problem, tasks are represented in the same way as solutions (in the form of WPC) we could use the same experimental setup for both populations, i.e. $(25 + 25)$-*ES* with uniform weight mutation of rate 0.1 (see Section 7.4.1 for details). To determine whether such setup is reasonable, we conducted a preliminary experiment with different mutation rates and an alternative (μ,λ) selection scheme.

Preliminary experiment

According to Fig. 7.14 it is favorable to employ the comma selection strategy and much larger mutation rates, as they result in both superior performance and substantially lower variance. These settings explicitly increased genotypic diversity within coevolving population. Indeed, regarding the size of the task space ($[-10, 10]^{64}$), such large mutations make the provided tasks almost random.

coev-task difficulty distribution

To scrutinize the dynamics of task distribution in the coevolving population, we assessed task difficulty in each generation by performing interactions with 1 000 randomly sampled policies. Figure 7.15 shows heat maps illustrating how difficulty distribution evolved across generations in case of two extreme values of mutation rate and different selection strategies. When compared to difficulty of randomly sampled tasks (cf. Fig. 7.8), training distributions provided by coevolutionary shaping are biased towards more difficult tasks, even in case of very large mutation rate. The important question is why distribution in Fig. 7.15b resulted in significantly lower performance than any other method considered before. Following earlier research in coevolutionary learning [32], we hypothesize that it is the matter of diversity within the coevolving population. If the mutation rate is too small, tests quickly become too similar to each other and they do not evaluate the individual's expected utility in the entire domain.

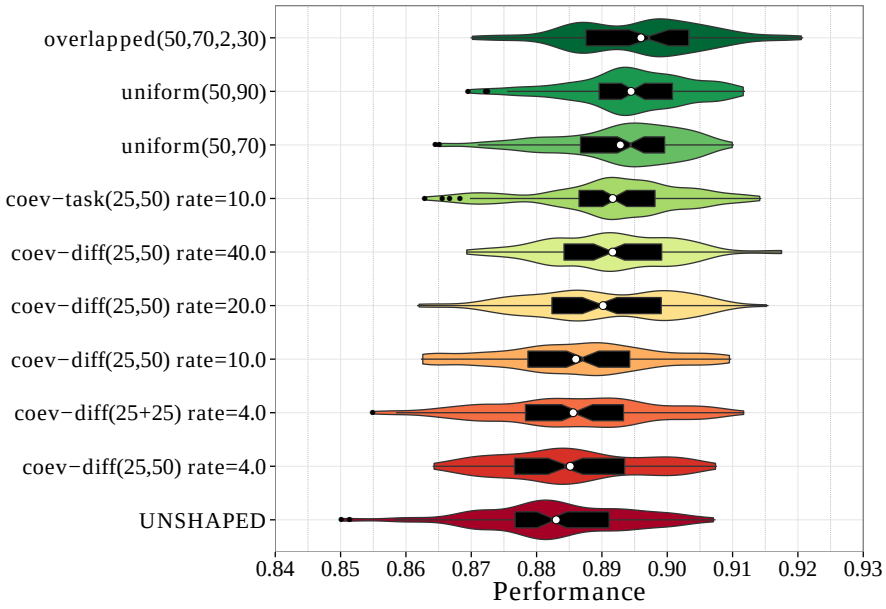


Figure 7.16: Performance of **coev-diff shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

7.4.6.2 *Coev-Diff Methods*

coev-diff test representation

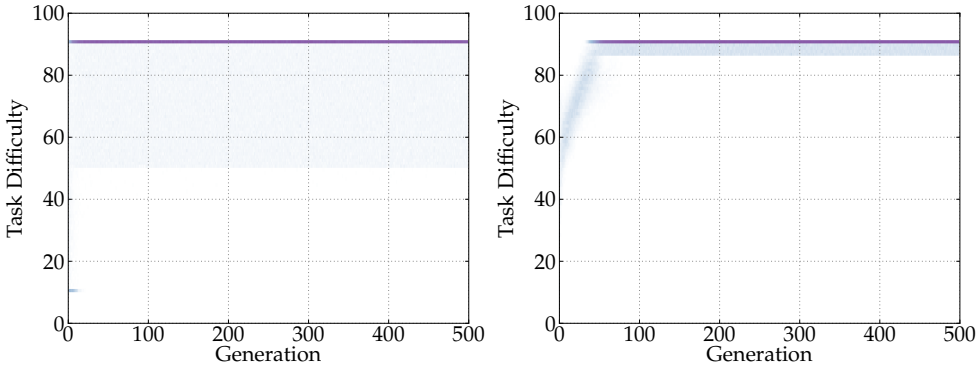
In the *coev-diff* coevolutionary shaping method, , each test is encoded as a single double value d which is interpreted as task difficulty measured in percentage points, and refers to a difficulty bin containing d, i.e., $B : d \in B$. To evaluate a policy, a single task is drawn from the corresponding task set T_B in a pool, and a double game is played. Mutating a test consists in adding a random value to d:

$$d' = d + rate \cdot \mathcal{U}[-1, 1], \tag{7.9}$$

where $\mathcal{U}[-1, 1]$ is a real value drawn uniformly from the range $[-1, 1]$ and *rate* is the mutation rate. As a result of mutation, the difficulty bin a test points to is possibly changed.

Final comparison

The preliminary experiment with different selection schemes and mutation rates revealed that this variant of coevolutionary shaping is much less susceptible to changing these parameters. Nevertheless, it was still advantageous to use relatively large mutation rate. Fig. 7.16 illustrates how *coev-diff* shaping methods perform when compared with the reference unshaped approach and the best previously considered shaping methods. Both variants of coevolutionary shaping, when properly tuned, were able to significantly outperform the unshaped approach (cf. Table A.3 in Appendix A.1).



(a) *coev-diff*$(25, 50)$, mutation rate = 40.0 (b) *coev-diff*$(25 + 25)$, mutation rate = 4.0

Figure 7.17: Difficulty distributions provided by *coev-diff* shaping methods over generations. Color saturation indicates relative occurrence frequency of tasks of certain dificulty.

coev-diff difficulty distribution

Moreover, it is also interesting to observe how the distributions of task difficulty change over generations in this coevolutionary approach. Figure 7.17 shows that the distributions converged towards the most difficult tasks in the pool. Apparently, these tasks were making the most distinctions among the population of policies. The speed of convergence depended on the mutation rate. Moreover, we can observe that by using mutation rate of 40, the resulting difficulty distribution shown in Fig. 7.17a covers the range $[50, 90]$, which corresponds to the best *uniform*$(50, 90)$ method considered in Section 7.4.3. We can also hypothesize that using distinctions as the only measure of test fitness could be a flawed idea because it can make distribution squeeze in the very narrow difficulty range.

Practical point of view

Finally, when compared with the best static shaping methods, represented by specific *overlapped* and *uniform* setups (cf. Fig. 7.14), coevolutionary shaping is not significantly weaker. Importantly, from a practical point of view, the comparison between these approaches should also involve the amount of human and computational effort needed to prepare successful setups. In this respect, static shaping methods require both computational power for building a difficulty-based task pool and human supervision for specifying training distributions. Coevolutionary shaping, by contrast, allows us to avoid some of these efforts. In particular, *coev-task* shaping creates training tasks *ad hoc* without access to the precomputed task pool. On the other hand, the process of tuning its test population parameters is burdensome while designing mutation operators may require some knowledge about the problem domain. The *coev-diff* method is much less sensitive to parameter settings and operates on task difficulties, which can be mutated by problem-independent operators. However, it does require a task pool.

7.5 OTHELLO INITIAL STATE DOMAIN

In the second Othello-based domain the goal is to learn a game-playing policy that maximizes the expected return when playing with the Standard WPC Heuristic Player (SWH, described in Section 5.1.3.1) starting from any valid game state. The multi-task learning problem which describes this goal is defined as follows:

- *Task set* $\mathcal{T}$. Each task $\tau \in \mathcal{T}$ is an instance of Othello game against the SWH player with one-point initial state distribution. Since the tasks across the domain differ only in the initial state, each of them is represented as an Othello state, i.e., a board encoded as a vector of 64 ternary-valued fields and the color of the starting player. In principle, this entire domain could be represented as a single MDP task with a properly specified initial state distribution. However, for the sake of consistency with other considered problems we decided to use the multi-task setting.

- *Task distribution* $\mathcal{P}$. Distribution of tasks is specified implicitly by the way the random task, i.e., its initial state, is generated. We adopted the following procedure to create a random but valid Othello state. First, we generate a path in the game tree by starting from the default Othello initial state (see Fig. 5.1a) and making a sequence of random legal moves until reaching the terminal state. Next, from the states encountered on such random path we uniformly choose one state. However, to avoid playing games that are already *resolved* (i.e., when one player is certain to win the game), we excluded the last five states on a random path.

- *Domain policies* Π. Similarly to the previous domain considered in Section 7.4, to represent a game-playing policy (candidate solution to the learning problem) we employ the WPC architecture as a state evaluator in 1-ply setup (see Section 5.1.2.2). Let us recall that for the purpose of estimating domain difficulty distribution and constructing a task pool, a *random policy* is generated by uniformly drawing the WPC weights from the range $[-10, 10]$.

- *Interaction episode*. An interaction episode corresponds to a single game of Othello between a candidate solution policy and the SWH policy, starting from the initial state defined by the task. The rewards depend on the game outcome in the same way as in Section 7.4. The maximal total reward of $C = 1$ is granted to a policy when it wins a game. In case of draw, a policy receives reward of 0.5, and there is no reward if the game is lost.

Setting	Value
Learning algorithm	$(\mu + \lambda)$ Evolution Strategy
Population size	$\mu = 25,\ \lambda = 25$
Initialization	$\sim \mathcal{U}(-0.2, 0.2)$
Mutation type	uniform perturbation
Mutation strength	$\delta = 0.1$
Number of training tasks	50
Episodes per generation	2 500
Generations	1 000
Number of runs	100

Table 7.3: Experimental settings in the Othello inital state domain.

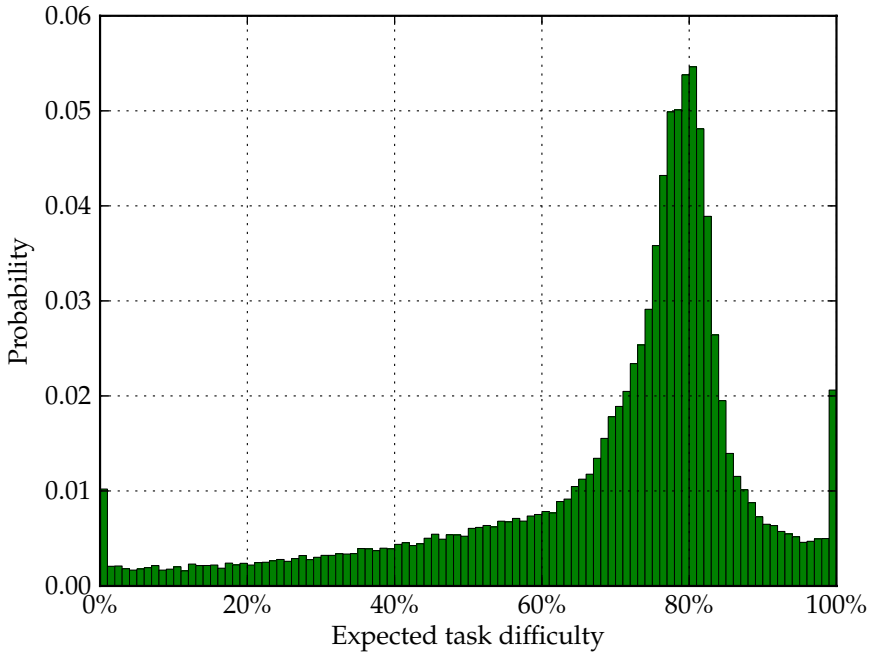
7.5.1 *Experimental Setup*

We retain most of the evolutionary parameters from Section 7.4, such as $(\mu + \lambda)$ selection strategy, population size of 50 and uniform mutation operator. To evaluate individuals, each of them is faced with a series of 50 tasks provided by the shaping method. However, in contrast to the opponent-based domain, here each task is used only for a single interaction episode. Consequently, the computational effort, meant as the number of training episodes per generation, is twice smaller than before. We doubled the number of generations to 1 000 while keeping the total effort equal to 2 500 000 training episodes (games) per run. The experimental settings are summarized in Table 7.3.

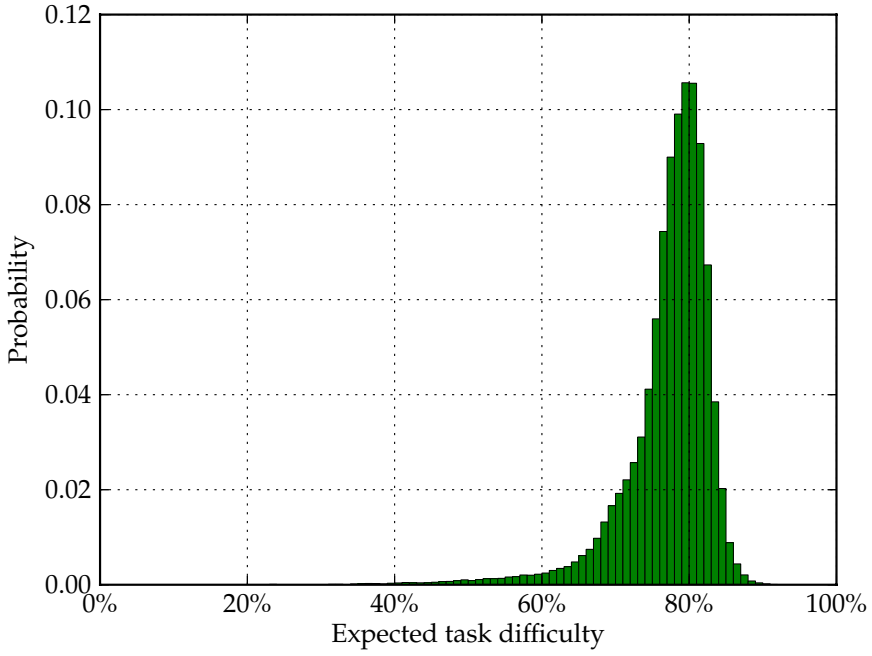
We conducted a series of experiments to judge the performance of shaping methods in this domain. However, contrary to the experiments in the opponent-based Othello domain, we omitted the hyper-heuristic shaping methods, as the experiment in Section 7.2.6.3 showed that they have limited practical value. Similarly as before, the experimental results were subject of the statistical analysis including Kruskal-Wallis and Mann-Whitney tests on each pair of considered methods. The results of these tests for particular experiments can be found in Appendix A.2.

7.5.2 *Domain Difficulty Distribution*

In analogy to Section 7.4.2, we started with estimating the domain difficulty distribution. Distribution illustrated in Fig. 7.18a was obtained by generating 100 000 random tasks (Othello initial states) and calculating their approximate difficulty (see Equation 7.4) on the basis of interactions with a set of 1 000 random policies. Each task was assigned to one of 100 difficulty bins, which correspond to difficulty



(a) Initial states are chosen from all but the **last 5** encountered states on a random path through the game tree.



(b) Initial states are chosen from the **first 20** encountered states on a random path through the game tree.

Figure 7.18: Difficulty distributions in the Othello initial state domain.

ranges of width 1%. Recall that we refer to task difficulties using percentage points, e.g., task difficulty of 60% means that randomly sampled policies get on average 40% of possible points when playing against SWH from the initial state specified by this task.

The estimated distribution shown in Fig. 7.18a is skewed towards the more difficult tasks. The mode of the distribution is equal to 80%, which can be explained by the strength of the SWH policy that plays the role of opponent in this domain. In fact, SWH is a fairly strong policy — its expected utility in full games (starting from the default initial state) against random opponents (estimated in an independent experiment) equals to 0.7871 ± 0.0024.

SWH's strength

Moreover, we can observe that long tails of the distribution span the entire range of task difficulty. As a result, we were able to easily fill all 100 task sets $\{T_B\}$ in a difficulty-based task pool to the assumed capacity of $N = 1\,000$. Let us emphasize that building such a pool is a required by most of the shaping methods proposed in this chapter.

Filling task-pool

Additionally, it is worth to notice the unusual peaks of the distribution for the tasks of difficulties 0% and 100%. These tasks correspond to games which are resolved (either already won or already lost). Clearly, such tasks may occur when the game starts very deep in the game tree with only a few moves left and no matter how the policy plays the result is always the same.

Resolved games

Importantly, the domain difficulty distribution largely depends on the procedure of random task generation, which is specified by the domain (task distribution $\mathcal{P}$). We can demonstrate how this distribution would change if we limit the set of tasks with respect to their initial depth in the game tree. Fig. 7.18b illustrates such distribution of tasks generated by choosing only among the first 20 states encountered on a random path down the game tree. This distribution is much more concentrated around the expected utility of the SWH policy. Moreover, since in such distribution all games last more than 30 moves, the resolved tasks of difficulty 0% or 100% are absent — task difficulties occupy mostly the range $[40, 90]$.

Another distribution

The observed change in difficulty distribution can be further explained by investigating the dependency between the depth of an initial state in the game tree and the difficulty of the corresponding task. For this purpose, we generated 100 000 random tasks. Figure 7.19 illustrates their expected difficulty as a function of initial state depth. For instance, in tasks starting at depth 0 (i.e., starting from the default initial state of the Othello game) the expected difficulty is around 80% . Thus, as expected, random policies are able to gain only 20% of all points when playing full Othello games against the SWH policy. However, quite surprisingly, it is easier to defeat SWH when games start from the states at odd depths, i.e. when playing as white. Higher difficulty of tasks starting at even depths is reflected in the 'sawtooth' shape of the red curve that plots the mean difficulty.

Depth dependency

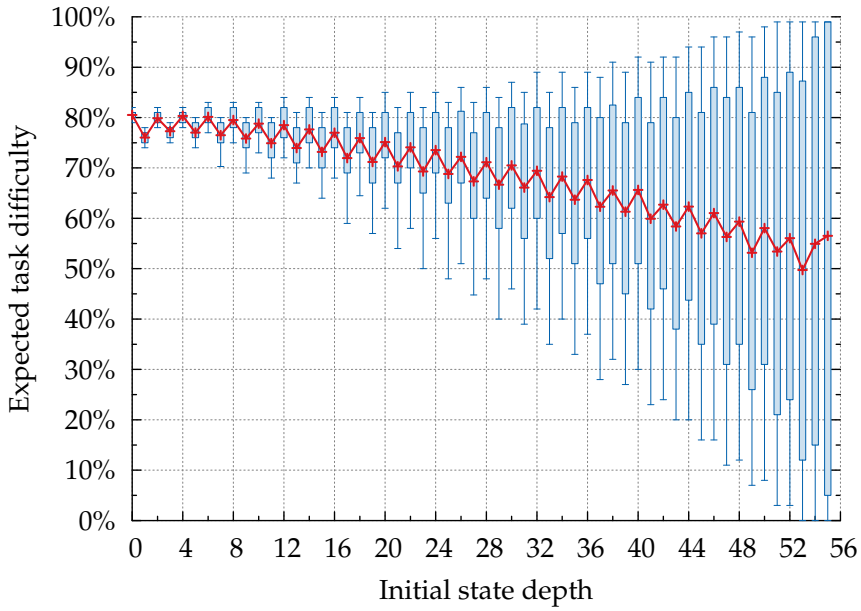


Figure 7.19: Expected task difficulty as a function of of its initial state depth. The red line illustrates the average difficulty, boxes denote the first and the third quartile while whiskers show the 9th and the 91st percentile.

Task difficulty variance

The most important observation is that the average difficulty of tasks evidently decreases with the growing depth of the initial state. This is accompanied by an exponential growth of the number of possible tasks and increasing variance of their difficulty. Contrary to the tasks that start at shallow depth, where everything depends on players policies, when starting a game deeper, the final game outcome is to a large extent affected by the specific initial state. Ultimately, the tasks starting very deep in the game tree exhibit very large difficulty variance. In particular, the tasks at depths of over 52 cover the the entire difficulty range.

7.5.3 *Single-stage shaping*

Similarly to Section 7.4.3, the first experiment in the Othello initial state domain consisted in applying the simplest single-stage shaping methods which rely on the handcrafted training task distributions. However, as illustrated in Fig. 7.20, this time such straightforward approach did not allow to significantly improve the learning performance with respect to the unshaped approach. Moreover, the obtained results indicate that this domain is generally much more challenging than the previously considered one: the average performance of each of the considered methods did not exceed 0.55.

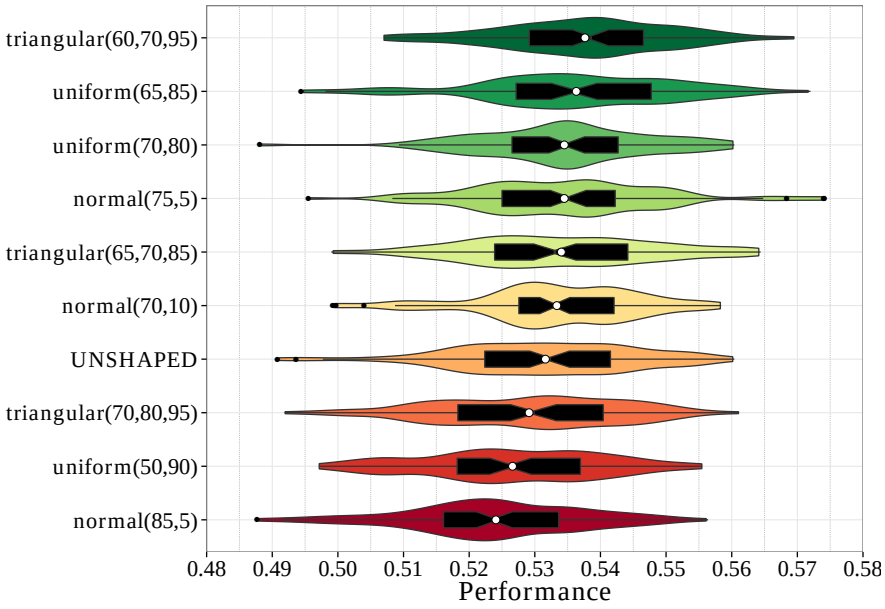


Figure 7.20: Performance of **single-stage shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

Despite statistical insignificance, we can identify the training distributions which resulted in the highest and lowest mean performance. Noteworthy, like in the Othello opponent domain, the overall training performance is relatively insensitive to the presence of easier tasks in the shaping distribution. Here it was enough to train on tasks of difficulty over 60%. On the other hand, extremely difficult training task distributions, like this of $normal(85, 5)$ resulted in decreasing the performance.

Difficulty-based transitivity

These observations can be related to the notion of *transitivity* considered e.g. in games [25, 168]. In transitive games, if player A beats player B and player B beats player C, then player A beats player C. Here, we would expect to observe some sort of *difficulty-based transitivity*, i.e., if a policy solves a harder task it should be also able to deal with an easier one. In this sense, the Othello initial state domain is apparently less transitive than the opponent domain. The degree of transitivity in a given domain could be a useful guideline for designing effective training distributions.

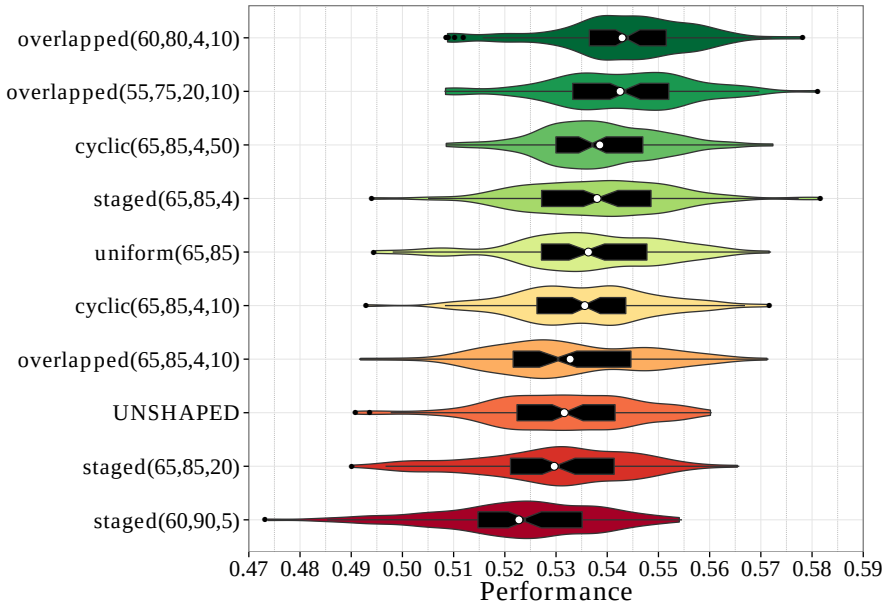


Figure 7.21: Performance of **multi-stage shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

7.5.4 *Multi-stage shaping*

In the next experiment we used the best identified uniform setup, i.e., *uniform*$(65, 85)$ as a starting point for comparing three multi-stage shaping methods. Figure 7.21 shows that neither dividing this difficulty range into four parts in *staged*$(65, 85, 4)$ nor iterating many times over these four subranges in *cyclic*$(65, 85, 4, 50)$, proved to be successful. In fact, only the *overlapped* method benefited from the incremental evolution technique. These results confirm our previous observations (see Section 7.12) that among the multi-stage shaping methods the *overlapped* one results in the most effective learning.

Superiority of the overlapped approach

That being said, it is quite surprising to see the relatively weak performance of the *overlapped*$(65, 85, 4, 10)$ method which does not differ much in terms of its training task distribution settings from the best in this field *overlapped*$(60, 80, 4, 10)$. Noteworthy, a significant difference in the obtained results occurs also between another pair of similar setups, namely *staged*$(60, 90, 5)$ and *staged*$(65, 85, 4)$. To explain this discrepancy we inspect the progress of learning by assessing the performance of these methods after each generation of the evolutionary algorithm.

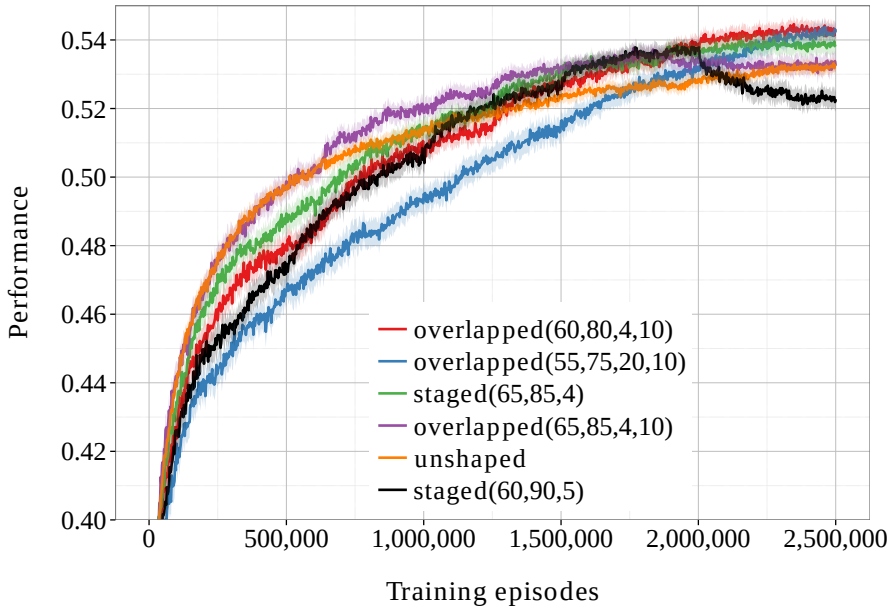


Figure 7.22: The performance of multi-stage shaping methods over time.

Performance over time

The performance of the selected multi-stage shaping methods over time is shown in Fig. 7.22. The learning curves quite clearly reflect the temporal changes in training distributions[2]. When one training distribution is switched to another, the learning speed tends to temporarily increase. For instance, *staged*$(60, 90, 5)$ changes the task distribution every 500 000 training episodes. After four stages of learning, i.e., 2 000 000 episodes, this method is one of the best among the considered ones. However, the last stage in which training tasks are sampled from the difficulty range $[84, 90)$ leads to substantial decrease in performance. A similar, albeit less significant drop in performance can be observed in the case of *overlapped*$(65, 85, 4, 10)$ which in the last stage of learning provides tasks of difficulty in range $[80, 90)$.

Evolutionary forgetting?

These observations allows to conclude that using the tasks of difficulty over 85% for training may lead to some degree of forgetting of the previously learned skills. Therefore, the inferior performance of *staged*$(60, 90, 5)$ and *overlapped*$(65, 85, 4, 10)$ with respect to their slightly altered counterparts can be ascribed mainly to the last phase of learning. Additionally, it is worth to notice the learning curve of the *overlapped*$(55, 75, 20, 10)$ method which reveals the slowest but the most stable learning progress. As a result, despite being the weakest most of the time, in the final learning stage it managed to surpass almost all the other methods.

2 By using the overlapped configurations, the transitions between successive training distributions are more 'smooth', and thus less noticeable in the figure, than in the case of staged methods.

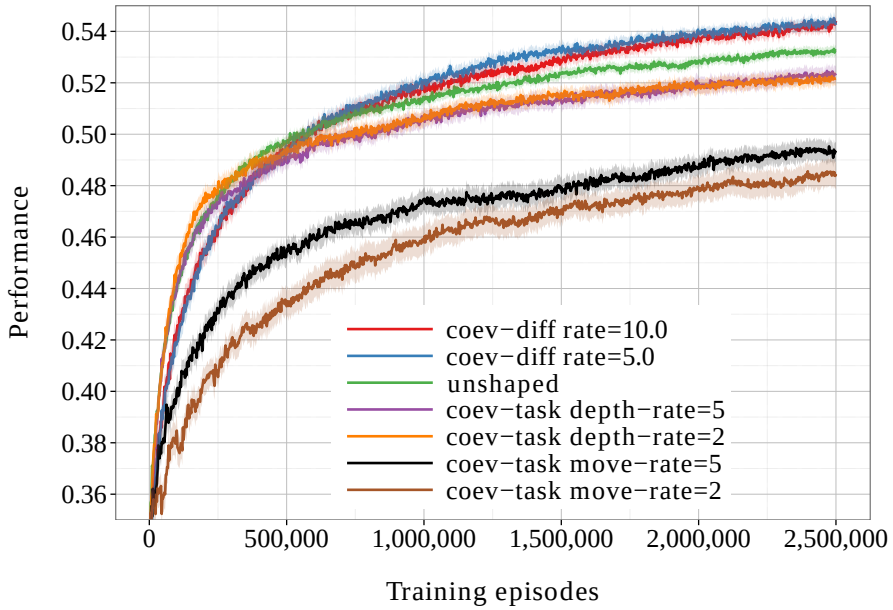


Figure 7.23: The performance of coevolutionary shaping methods over time.

7.5.5 *Coevolutionary Shaping*

Finally, we consider the coevolutionary shaping methods. Although these methods can be seen as the most self-sustaining among all the considered ones, they still require setting several parameters for the population of tests (tasks). To this end, we tried to retain most of the settings that were found useful in the previous application of coevolutionary shaping (see Section 7.4.6).

Tests population parameters

Particularly, as for the population of tests, we preserved both its size (50) and the comma selection strategy with $\mu = 25$ and $\lambda = 50$. Additionally, in the case of *coev-diff* methods we employed the same uniform perturbation operator because tests were represented as single double values corresponding to task difficulties. In fact, the only parameter left that needed to be determined was the mutation operator for the *coev-task* method. In this domain, the space of tasks was no longer equivalent to the space of policies, so we could not use the same search operator in both populations. Consequently, we proposed two dedicated operators that implement mutation of individuals representing Othello states:

- *move* operator — Starting from a given state (current individual), this operator performs up to k randomly selected legal moves down the game tree. If a terminal state is reached in the course of such mutation process, the current state is discarded, and a completely new state is drawn according to the domain-specific random state generation procedure.

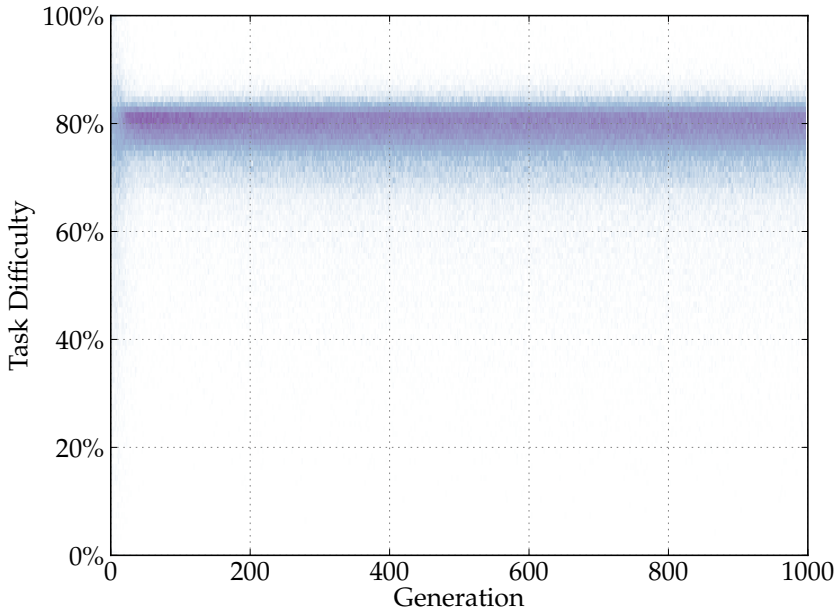


Figure 7.24: Difficulty distribution provided by *coev-task* shaping with the ***depth*** mutation operator (depth rate = 2) over generations. Color saturation indicates relative task occurrence frequency.

- *depth* operator — Given a state at depth d, this operator generates a completely new random state at depth $d' \in [d-k, d+k]$ using the domain-specific random state generation procedure.

Figure 7.23 illustrates the performance of the coevolutionary shaping methods, each of which was examined with two mutation rate values. Among the different variants of the *coev-task* method, the *depth* operator was significantly more effective than the *move* operator. However, no matter how configured, the *coev-task* approach resulted in lower performance than the reference unshaped approach. The *coev-diff* method, by contrast, was again one of the most successful in the field. Although initially it progressed slightly slower, after approximately 500 000 training episodes it surpassed the unshaped method and later only increased the advantage over the other other methods. Similarly as in Section 7.4.6.2, we found the *coev-diff* method less sensitive to parameter settings than *coev-task*.

Performance comparison

To explain the inferior performance of *coev-task*, we investigated the difficulty of the training tasks provided by particular variants of this method. Figure 7.24 demonstrates that at the beginning, especially between roughly 50th and 100th generation, the distribution of tasks evolved with the *depth* mutation operator resembles the target domain difficulty distribution (cf. Fig. 7.18a): it is concentrated in a narrow region, with a peak (the darker area) just above 80%. For this reason, the initial learning progress of this setup is almost identical to that exhibited by the unshaped approach (see Fig. 7.23). Only later the difficulty distribution gets more dispersed.

coev-task difficulty distributions

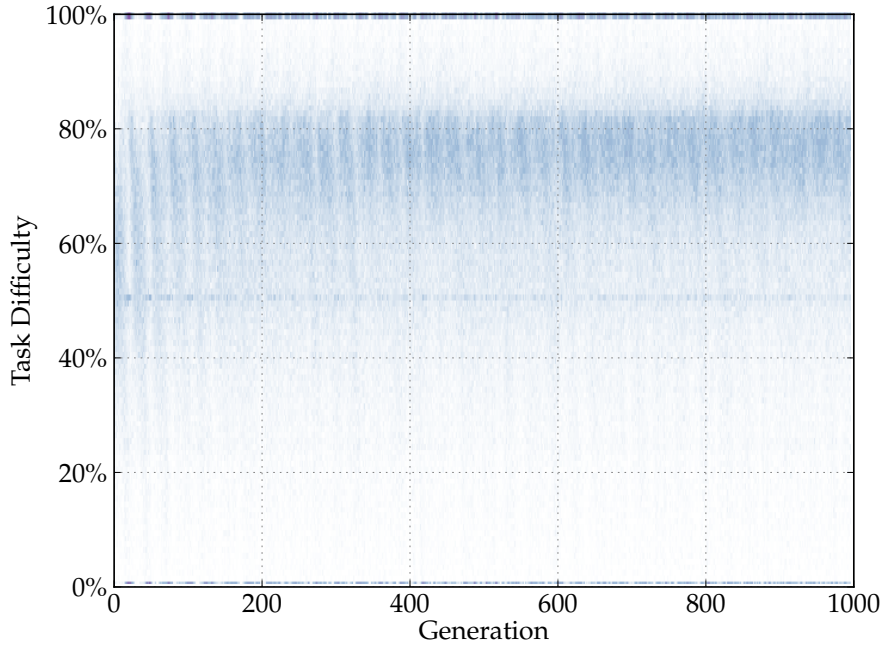


Figure 7.25: Difficulty distribution provided by *coev-task* shaping with the ***move*** mutation operator (move rate = 2) over generations. Color saturation indicates relative task occurrence frequency.

Cyclic regularities

The training difficulty distribution evolved by the means of the *move* mutation operator is illustrated in Fig. 7.25. The graph displays interesting regularities which appear to occur cyclically every few tens of generations. It can be observed that over time the task difficulties oscillate between three values — 0%, 50% and 100%, and then suddenly scatter over a wide range of difficulties. This phenomenon can be attributed to the characteristics of the mutation procedure and the dependency between task difficulty and the initial state depth illustrated in Fig. 7.19.

Explanation of regularities

By examining the distribution shown in Fig. 7.25 at the very early generations we can observe that most tasks are of difficulty between 40% and 70% (supposedly because they make the most distinctions). Due to the strength of the SWH opponent, such relatively easy tasks do not appear frequently at shallow depths of the game tree (cf. Fig. 7.19). Therefore, we can assume that initially the evolutionary selection promotes tasks starting from the states in the lower part of the game tree. However, after several generations, mutation of these states pushes them downward the game tree and causes them to get very close to the end of the game. In such situations it is often the case that the game outcome is already determined as victory (task difficulty of 0%) or defeat (difficulty of 100%). Moreover, in relatively many states, especially with just two empty board locations, the chances of

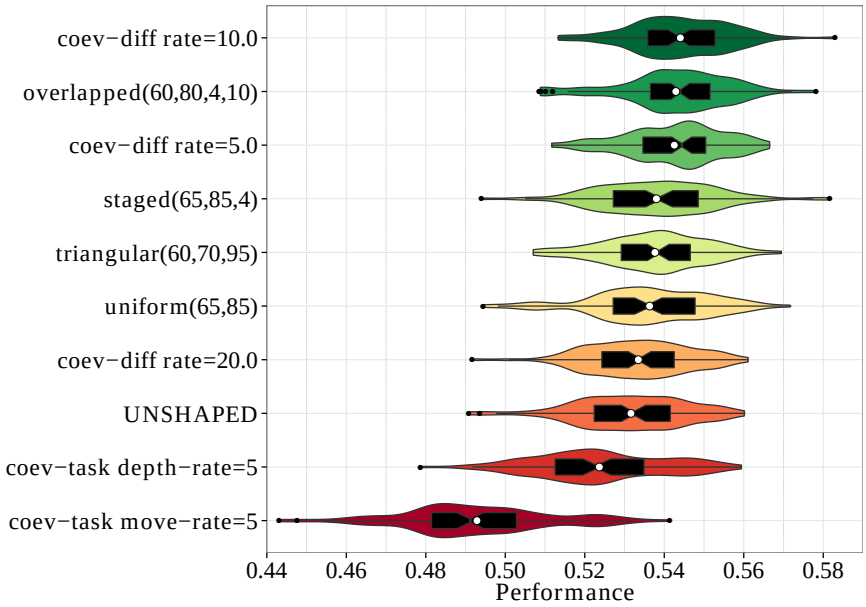


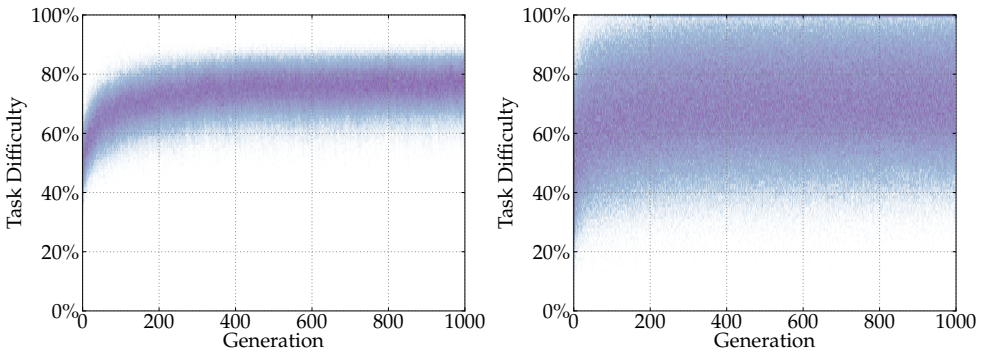
Figure 7.26: Performance of **coevolutionary shaping** methods shown as violin plots. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Narrowings of the box around the median indicate 95% confidence interval. Methods are sorted in descending order according to their mean performance (shown by white circles).

winning the game are exactly equal for both players[3] (task difficulty of 50%). Next, when such tasks are to be mutated, the *move* operator often generates completely new tasks in their place (due to reaching the terminal state in the game tree). Task difficulties became dispersed again, and start converging to the three mentioned difficulty values, leading to the observed cyclic behavior.

Consequences of regularities

Noteworthy, the dramatic changes in difficulty distribution evolved with the *move* operator are reflected in the learning curves shown in Fig. 7.23. Clearly, a few sudden drops in performance can be spotted in the initial period of learning. In the later generations this phenomenon is not so apparent because as the tasks get more difficult they can be more diversified with respect to the depth of their initial states (cf. Fig. 7.19) and they do not reach terminal states at the same time (probably for the same reason, the cyclic behavior becomes less prominent in the later stages of evolutionary search, Fig. 7.24). Nevertheless, all things considered, this investigation demonstrates that the *move* operator is essentially flawed and certainly should be redesigned.

3 This is what Othello owes its name: even if player's scores dramatically differ at a given point of the game, a single move can still change the game outcome.



(a) *coev-diff*, mutation rate = 5.0 (b) *coev-diff*, mutation rate = 20.0

Figure 7.27: Difficulty distributions provided by *coev-diff* shaping methods over generations. Color saturation indicates relative occurrence frequency of tasks of certain dificulty.

Comparison of the best methods

The *coev-diff* shaping methods employ universal mutation to search for the most learnable task difficulties. Figure 7.26 illustrates how they perform when compared with the best previously considered approaches. Clearly, the well-configured *coev-diff* method (with mutation rate equal to 5 or 10) achieves one of the highest performances in this comparison. Recall that this method requires much less parameter tuning than e.g. the *overlapped* approach. In fact, there is only a mutation rate to be fixed, but as shown in the figure, it does not need to be determined in a very fine-grained tuning. In preliminary experiments we achieved a comparable performance with any mutation rate between 2 and 10. Only increasing the rate to 20 resulted in a significant drop in the final performance.

coev-diff difficulty distributions

We analyzed the difficulty distribution provided by the *coev-diff* shaping methods with two different values of the mutation rate. The results of this assessment are illustrated in Fig. 7.27. As expected, larger mutation rate caused the difficulty distribution to be much more dispersed. At first sight, the distribution shown in Fig. 7.27b may be compared with that provided by the $\mathit{uniform}(50, 90)$ method evaluated in Section 7.5.3. However, if we examine carefully the color intensity, we note that this distribution is not flat but rather bell-shaped. This characteristic can explain the difference in results obtained by these two methods.

Increasing difficulties

Finally, let us note that both distributions presented in Fig. 7.27 conform to the original definition of shaping meant as the method of successive approximations. Initially most of the training tasks are relatively easy but their difficulty increases over time and in a longer run the mean task difficulty roughly matches that in a target domain difficulty distribution. The above experiments provide evidence that training in such increasingly complex environments is significantly more effective than learning directly in the target domain.

7.6 POLE BALANCING DYNAMICS DOMAIN

The last domain considered in this chapter involves the problem of pole balancing described in Section 5.3. The objective is to learn a policy that maximizes the expected return in a domain of pole balancing tasks with different physical properties. The multi-task learning problem which describes this goal is defined as follows:

- *Task set* $\mathcal{T}$. Each task $\tau \in \mathcal{T}$ is a single pole balancing task formulated as an MDP in Section 5.3.2. The tasks across the domain differ with respect to selected physical parameters, namely: the length of the pole $l \in [0.1, 1.0]$, the mass of the pole $m \in [0.1, 1.0]$ and the mass of the cart $M \in [1.0, 10.0]$. Since these parameters directly influence the dynamics of the cart-pole system, the tasks in the domain vary with respect to the transition function. All other elements of MDPs remain constant, so the space of tasks is essentially a three dimensional real space, where each dimension corresponds to a single problem parameter. Table 5.5 shows the remaining pole balancing parameters, including track length, failure angles and friction coefficients. Values of these parameters are based on previous works [13, 72].

- *Task distribution* $\mathcal{P}$. Each task is equally likely to be drawn from the considered three dimensional problem configuration space. A *random task* is generated by uniformly drawing value of each parameter from the real intervals given above.

- *Domain policies* Π. We emply neural networks to represent policies for the pole balancing domain. Although in general we assumed a multilayer architecture (see Section 2.2.1.1), the preliminary experiments revealed that it is enough to use a single nonlinear neuron to effectively operate in this domain. Given the four state variables $(x, \dot{x}, \theta, \dot{\theta})$, the neuron outputs the amount of force to push the cart (with force direction encoded by the sign). For the purpose of estimating domain difficulty distribution and constructing a task pool, a *random policy* is generated by uniformly drawing network weights from the range $[-6, 6]$.

- *Interaction episode*. An interaction episode corresponds to a single simulated trial of pole balancing. Each such trial starts from the state $(0, 0, 1°, 0)$ and lasts until the task is failed or until 100 time steps have passed. Because the reward of 1 is given at each time step before a failure, the maximal policy return achievable in a single episode $C = 100$.

Setting	Value
Learning algorithm	$(\mu + \lambda)$ Evolution Strategy
Population size	$\mu = 10,\ \lambda = 10$
Initialization	$\sim \mathcal{U}(-6.0, 6.0)$
Mutation type	uniform perturbation
Mutation strength	$\delta = 0.1$
Number of training tasks	20
Episodes per generation	400
Generations	1000
Number of runs	100

Table 7.4: Experimental settings of evolutionary learning in the cart pole balancing domain.

7.6.1 *Experimental Setup*

Population of solutions

To learn policies for the pole balancing domain, we employed $(\mu + \lambda)$ evolution strategy, with $\mu = 10$ and $\lambda = 10$, which evolved weights for a single nonlinear neuron. After drawing initial weight values from the range $[-6.0, 6.0]$, the space of possible weight configurations was searched by uniform mutation operator:

$$w_i' = w_i + \delta \cdot \mathcal{U}(-1, 1), \tag{7.10}$$

with the mutation strength δ equal to 0.1.

Fitness evaluation

In the evaluation phase, each candidate solution was used as a cart control policy in a series of 20 training tasks. In the case of unshaped learning, the tasks were generated uniformly from the task space. Otherwise, they were provided by a shaping method. Regardless of the source of training tasks, single fitness assignment required simulating 400 training episodes in total ($(10 + 10)$ individuals $\times$ 20 training tasks). Because each evolutionary run comprised 1 000 generations, the overall computational effort added up to 400 000 training episodes. The experimental settings are summarized in Table 7.4.

Limiting the scope of comparison

We conducted a series of experiments to empirically evaluate the performance of the methods presented in this chapter. However, in this domain we focused only on two shaping approaches, namely, single-stage static shaping and coevolutionary shaping. These two approaches were found the most robust in previous experiments. Similarly to the Othello domains, we executed 100 independent runs of each method and the obtained results were subject to statistical analysis involving Kruskal-Wallis and Mann-Whitney tests on each pair of considered methods. The results of these tests for particular experiments can be found in Appendix A.3.

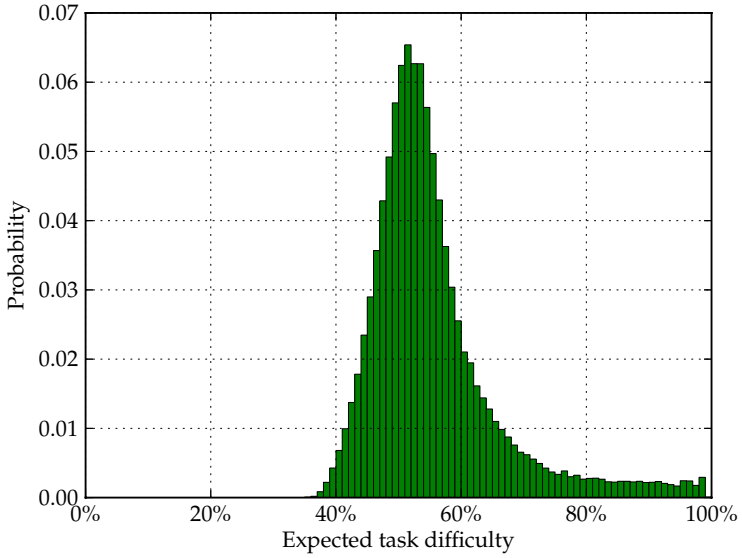


Figure 7.28: Domain difficulty distribution in the pole balancing dynamics domain.

7.6.2 *Domain Difficulty Distribution*

First, we generated 100 000 random tasks to estimate the domain difficulty distribution. Each task was confronted with 1 000 randomly sampled policies represented by single neurons. The average return obtained by random policies was used to approximate the difficulty of the task (according to Equation 7.4). Recall that the return of a policy is determined by the number of time steps until failure. For instance, if a random policy balances the pole for 20 time steps on average, the difficulty of the task is equal to 80% (because a single episode can last maximally 100 time steps).

Figure 7.28 illustrates the distribution of tasks with respect to their difficulty. Although a random policy is able to balance a pole for $40 - 60$ time steps on average for most of the tasks, the distribution is positively skewed with task difficulty extending with a long tail towards the higher values. The distribution indicates that challenging tasks can be generated without much difficulty, while it is strenuous to find such easy tasks that a random policy keeps the pole upright for more than 70 time steps (i.e., corresponding to task difficulty of 30% or less).

Skewed distribution

As a result, we were unable to fill all task sets in a difficulty-based task pool to a assumed capacity $N = 1\,000$. Ultimately, the task pool comprised only 62 task sets corresponding to difficulties in the range $[37\%, 98\%]$. Noteworthy, even in the extremely difficult tasks, it takes a few time steps for the pole to reach the failure angle. For this reason the range of task difficulties does not extend to 100%.

Unfilled task pool

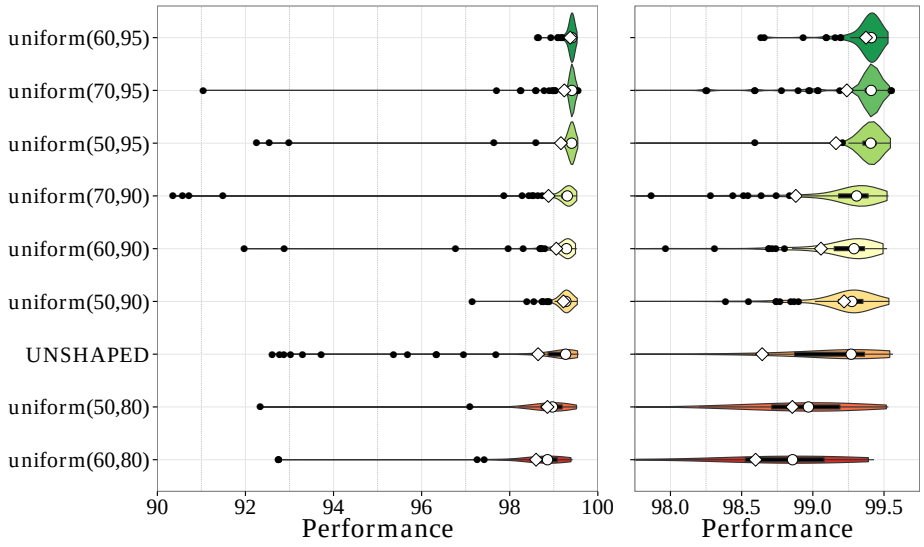


Figure 7.29: Performance of **uniform shaping** methods. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Methods are sorted descendingly according to the median of their performance (shown by white circles). White squares mark the mean performance.

7.6.3 *Single-Stage Shaping*

In the initial experiment, we employed selected single-stage shaping methods. For the sake of simplicity, we limited the scope of considered methods only to those based on the uniform task distribution. This decision was motivated by the fact that in the previously considered domains the *uniform* approach was not found significantly worse than the *triangular* or the *normal* one.

Figure 7.29 compares the final performance of single-stage shaping with that achieved by the unshaped approach. Although the majority of policies obtain very high return, there are also relatively many outliers which make the violin plots very wide and thus difficult to compare. For this reason we provide also a magnified view of the most interesting region of the plot, which allows to observe the differences in means and medians between the learning methods.

The results confirm our previous observations that even the straightforward single-stage shaping approach, if properly tuned, can significantly outperform the conventional learning method. Indeed, we can observe that most shaping methods resulted in both higher mean and lower variance of the final performance. The best policies were produced by training on a wide spectrum of tasks, including the very difficult ones which were found to be crucial. For instance, the superiority of $\mathit{uniform}(50, 95)$ over $\mathit{uniform}(50, 90)$ can be ascribed to training on tasks of difficulty between 90% and 95%.

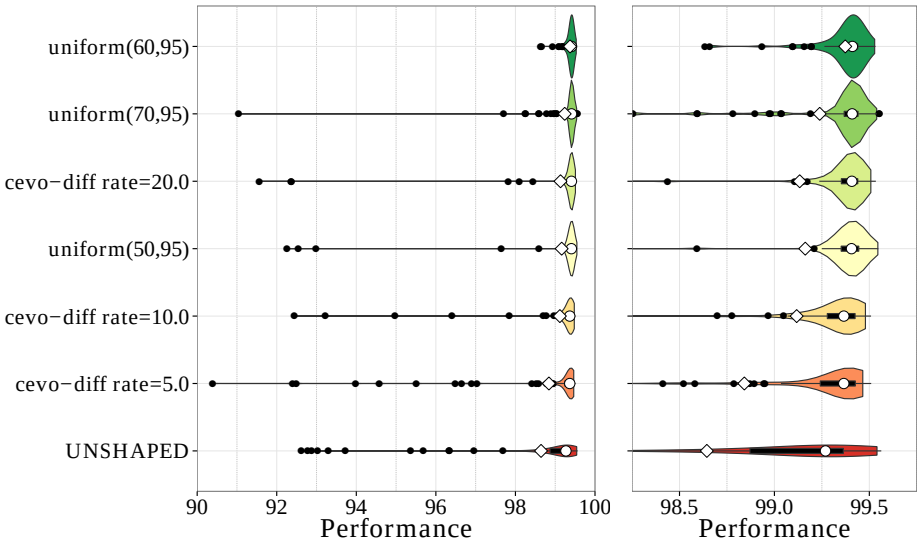


Figure 7.30: Performance of **coevolutionary shaping** methods. Each black box spans from the first to the third quartile (the interquartile range or IQR), while the whiskers extend to the highest and lowest observations within 1.5·IQR from the box. Outliers beyond this range are denoted by black dots. Methods are sorted descendingly according to the median of their performance (white circles). White squares mark the mean performance.

7.6.4 *Coevolutionary Shaping*

The second approach examined in the pole balancing domain is coevolutionary shaping. Here, we focus solely on the *coev-diff* method, which was found the most successful among all dynamic shaping methods applied in the Othello domains.

The population of tests contained 20 individuals, to provide a fair comparison with the uniform and unshaped methods that used samples of 20 training tasks. Tests were evolved by means of (μ, λ) evolution strategy with $\mu = 10$ and $\lambda = 20$. They were mutated by the same uniform perturbation operator as before (cf. Section 7.4.6.2) and their fitness was calculated as the number of distinctions they made among the individuals in the current population of policies. We assumed that a test (task) made a distinction between two policies if their returns differed by more than $Y = 2$ (see Equation 7.7). Intuitively, a distinction occurred if one policy was able to balance a pole at least two time steps longer than another.

Figure 7.30 shows the performance obtained by *coev-diff* shaping methods with varying mutation strength. Like in the opponent-based Othello domain, it was beneficial to use larger mutation rates and thus ensure greater diversity of training tasks. Importantly, each of the considered *coev-diff* methods achieved significantly higher performance than the unshaped approach. On the other hand, they did not outperform the best single-stage shaping methods.

7.7 DISCUSSION

Let us rephrase the original research question with which we began this chapter: how can we improve the results of evolutionary algorithms applied to multi-task reinforcement learning problems? We have attempted to answer this question by referring to the concept of shaping. Specifically, we have modified the distribution of training tasks employed to evaluate the evolving policies. By doing so we expected to shape the fitness function and make it easier for an evolutionary algorithm to explore other areas in the solution space.

Machine learning perspective

From the machine learning perspective, the rewards gathered in interactions with evaluation tasks constitute the training experience used by the learning algorithm to improve the policies. Therefore, modifying the distribution of evaluation tasks indirectly guides the learning process. The role of proposed shaping methods was to provide such tasks that facilitate learning by letting the population of agents gather the informative training experience.

The role of difficulty measure

The main question was how to select evaluation tasks from a potentially infinite domain. To answer this question we introduced the measure of task difficulty which allows to order the tasks in the given domain. Conceptually, the abstract notion of task difficulty conforms to motivations of shaping and incremental evolution methods which in principle aim at making tasks easier to solve. Practically, the measure of task difficulty is easily estimated and allows to devise numerous shaping methods that do not require intimate domain knowledge.

Benefits and costs of shaping

Most of the proposed difficulty-based shaping methods were able to significantly improve the learning performance when compared to the conventional unshaped approach. Moreover, in two out of the three experimental domains, the straightforward single-stage shaping was enough to gain the performance benefits. Although some of the considered shaping methods require careful parameter configuration, the parameters are typically domain-independent, so they do not require intricate knowledge of the problem. Among the considered methods, coevolutionary shaping can be regarded as the most autonomous because it does not need predefined training task distributions. Coevolution strives to adaptively provide such training tasks that construct a learnable gradient for the population of learners. Importantly, the coevolutionary approach proved quite general in beating other methods in all three domains.

Common observations

The common conclusion from the experiments conducted in three different multi-task domains is that the evolutionary learning algorithm tends to benefit from focusing fitness evaluation on challenging tasks. A shaped fitness function that encourages solving such tasks allowed evolution to find the policies that are generally better with respect to the expected utility (return) in the entire domain. This observation is somewhat surprising because, technically, the unshaped

approach is tailored to maximize expected utility by employing its unbiased estimate to steer evolution; in other words, it learns from the same distribution of tasks on which it is later assessed. On the other hand, the observed results confirm an intuitive belief that if a policy copes with a difficult task it will be able to handle related but less demanding tasks too. These observations point to the need of investigating further the concept of difficulty-based transitivity, and measuring the degree to which solving a difficult task implies solving the easier ones too.

Limitations of difficulty-based shaping

The applicability of difficulty-based shaping is limited to domains containing tasks of diversified difficulty. Indeed, in order to build a useful difficulty-based task pool, which is required by most shaping methods, randomly generated tasks should fill many difficulty bins. Therefore, the domain should include tasks that enable random policies to get very different returns. For instance, it would be hard to apply shaping methods for a domain of relatively challenging tasks, like, e.g., double pole balancing, where random policies are very unlikely to keep the poles upright for more than a few time steps. As a result, all tasks would be considered as equally difficult and shaping methods would not be able to construct any training task distributions. The only method that could successfully operate in such a domain is *coev-task* which does not require predefined training distributions, but constructs them dynamically.

Applications of difficulty measure

Finally, the introduced measure of task difficulty was found useful for two additional purposes. First, it allows to perform multi-criteria assessment of trained policies with respect to how well they deal with tasks of different difficulties. In this context, a single task difficulty distribution can be used as a separate criterion for comparing the performance of policies. For instance, we have observed that, although at first sight the improvement elaborated by our shaping methods is not particularly large, they attain much higher performance in the most difficult tasks. The second application of task difficulty measure is 'reverse engineering' of task distributions in coevolutionary shaping. By investigating the difficulty of tasks in the coevolving population, we were able to better explain the reported results.

8

SHAPING IN TEMPORAL DIFFERENCE LEARNING

The previous chapter introduced a number of shaping methods that provide training tasks according to the proposed measure of task difficulty. These methods rely on a heuristic assumption that task difficulty alone constitutes a sufficient criterion to synthesize such training task distributions that can facilitate learning. In this chapter, rather than relying on heuristic measures, we consider the problem of selecting the training tasks optimally with respect to the assumed learning algorithm.

The motivation for this chapter is similar to that of the previous one: we expect that it is possible to identify training tasks that make the learning process more effective by letting the agent to observe more informative training experience. This leads to mapping the original learning problem of optimizing an agent's policy into a *dual* problem of finding the best input for the policy learning algorithm, while preserving the ultimate goal of learning — maximization of an adopted quality measure.

However, in contrast to the previously assumed perspective, here we do not assume the availability of a generalized domain from which the training tasks can be selected. Instead, given a single target task, we need to derive a family of related tasks hoping to identify such variations of the target task that allow to gather useful training experience. By related tasks we mean tasks that vary with respect to one of the elements of the underlying MDP (see Section 3.2.2). For instance, given a game-playing task with a specific opponent, we consider a family of related tasks with different opponents or another set of tasks with modified initial state distribution.

We start this chapter by framing the tasks experienced by the agent during learning as the *shaping task sequence* (see Section 8.1). On this basis, we formalize the problem of optimal shaping sequence (Section 8.1.1) and design a coevolutionary method which attempts to find such a sequence of training tasks for a temporal difference learning algorithm (Sections 8.1.2 – 8.1.3). In Section 8.2 we consider two types of shaping tasks that can be used to facilitate learning of game-playing policies for the game of Othello. Finally, in Section 8.3 we provide detailed experimental setup and apply the proposed methods to learning Othello policies. The results demonstrate that training on the identified task sequences results in both faster learning and increased final performance.

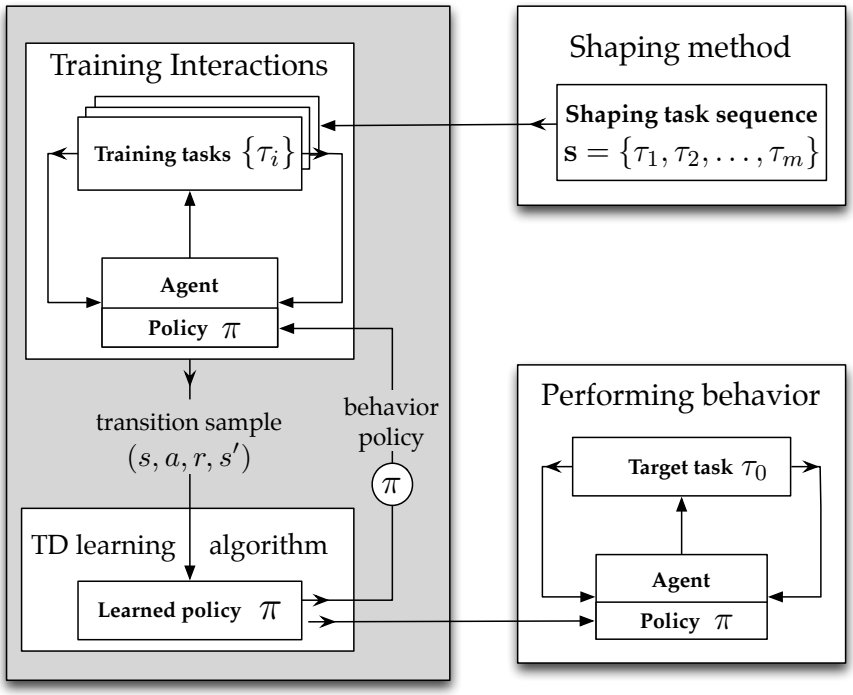


Figure 8.1: The outline of shaping for temporal difference learning

8.1 OPTIMIZATION OF SHAPING TASK SEQUENCES

The perspective of the shaping framework

Like in the previous chapter, we start by referring to Fig. 4.1 to see how the shaping setup considered here fits into our unified shaping framework. As illustrated in Fig. 8.1, the goal of learning in this setup is to perform well on a given single *target task* τ_0. To this aim, the temporal difference learning algorithm maintains a single policy π and adjusts it after every action taken in the training environment. The adjustment is based on the recently collected training experience represented as a tuple of the form (s, a, r, s'). Each such tuple corresponds to an observed environmental transition and describes the consequences of taking action a in state s, i.e., a received reward r and a successive state s'.

A family of task variations

In a typical unshaped approach, the training interactions take place directly and only in the target task τ_0. The considered shaping approach relies on viewing the task τ_0 as only one member of a family of tasks $\mathcal{T}$ that contains all possible variations of τ_0 with respect to selected problem-specific parameters. Consequently, the family $\mathcal{T}$ forms a *problem configuration space* [173]. Here we assume that the parameters may influence the transition function or the initial state distribution but all the tasks share the same state space S and action space A. As a result, when learning a policy $\pi : S \to A$, training experience can be gathered not only in the target task τ_0, but in any other task $\tau_i \in \mathcal{T}$.

In this framework, the role of a shaping method is to identify such variations of the original goal task τ_0 that can facilitate the learning process by letting the agent to gather more informative training experience. However, in contrast to the approaches described in the previous chapter, here we are not interested in how the shaping method operates. Instead, we focus solely on its final effect — the resulting *shaping task sequence* $\mathbf{s}$ which includes m training tasks $\{\tau_i \in \mathcal{T} \mid 1 \leq i \leq m\}$. In the following section we consider how the shaping sequence influences the results obtained by the learning algorithm and we formalize the problem of optimal shaping sequence.

Shaping task sequences

8.1.1 *Optimal Shaping Task Sequence*

In the supervised approach to classification problems, a learning algorithm can be regarded as a function that given a set of labeled training examples constructs a classifier [6]. Such perspective is adopted in *batch reinforcement learning* [55, 109], where training experience, represented as a set of tuples of the form (s, a, r, s'), is fixed and given *a priori*. Consequently, the batch-mode reinforcement learning algorithm $\mathcal{L}_{batch}$ can be framed as a mapping from a set of n such transition samples $\mathcal{F} = \{(s_i, a_i, r_i, s'_i) \mid 1 \leq i \leq n \wedge s_i \in S \wedge s'_i \in S \wedge a_i \in A\}$ to a decision making policy $\pi : S \to A$:

Batch reinforcement learning

$$\mathcal{L}_{batch} : (S \times A \times \mathbb{R} \times S)^n \to \pi. \tag{8.1}$$

In the most general formulation of the batch reinforcement learning problem, no assumptions are placed on the way the set of transitions is generated. In particular, they do not need to form connected trajectories and can be sampled from one or more interaction episodes.

The problem of selecting an optimal input for the batch-mode reinforcement learning algorithm is considered by Rachelson et al. [158]. The authors attempt to identify a set of transitions which, when supplied to the given learning algorithm, lead to the optimal behavior with respect to the specific performance measure. In this method the learning proceeds independently from the selection of training experience — the learner cannot affect the way the experience is gathered. In other words, the learning algorithm is assumed to work in an offline manner, i.e., it exploits a fixed, prepared in advance set of training examples (sample of transitions), without a need of dynamically interacting with the environment.

Optimal sample selection

Our shaping approach abstracts from the character of the learning algorithm and is more coarse-grained — instead of selecting single transitions, we attempt to identify useful training tasks which can be used as a source of informative training experience. Particularly, in the online case, the training experience is not given *a priori*, but is alternately sampled during interactions in a training task and then used for adjusting the policy. Essentially, in the proposed shaping

Optimal training task selection

approach training interactions take place not only in a single target task τ_0 but in a sequence $\mathbf{s}$ of related tasks from a family of tasks $\mathcal{T}$. Importantly, besides the shaping task sequence, training experience (sampled transitions) depends also on the exploration strategy used to determine actions taken in such training environments. However, if we assume that this strategy remains fixed or is specified by the learning algorithm, we can redefine the learning algorithm as a function of the shaping task sequence:

$$\mathcal{L} : \mathcal{T}^m \to \pi. \tag{8.2}$$

Optimal shaping task sequence

From now on, we will use $\mathcal{L}$ in this particular meaning. In this context, the ultimate goal of the shaping method is to identify the optimal shaping task sequence, i.e., such sequence $\mathbf{s} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ that maximizes the expected return obtained in the target task τ_0 by the policy $\pi_{\mathbf{s}}$ learned on this sequence of training tasks:

$$\mathbf{s}^* = \arg\max_{\mathbf{s} \in \mathcal{T}^m} \mathbb{E}\left[J(\pi_{\mathbf{s}}, \tau_0) \mid \pi_{\mathbf{s}} = \mathcal{L}(\mathbf{s})\right]. \tag{8.3}$$

Practical issues

Elegant as it is, such formulation requires clarifying two important practical issues. First, the mapping from a shaping task sequence $\mathbf{s}$ to a decision making policy $\pi_{\mathbf{s}}$ realized by the learning algorithm $\mathcal{L}$ can be non-deterministic. The learning process may be random due to stochasticity of the transition and reward functions of the MDP or the environment exploration strategy such as ϵ-greedy policy. Second, the mapping implemented by the algorithm $\mathcal{L}$ is rarely a single-step process. In the online learning scenario considered here, the algorithm works incrementally and improves the policy gradually after observing consecutive transition samples.

8.1.2 *Learning from a Shaping Sequence*

Temporal difference learning algorithm

Algorithm 8.1 presents an implementation of the temporal difference learning algorithm $\mathcal{L}$ which constructs a policy $\pi_{\mathbf{s}}$ from a given shaping task sequence $\mathbf{s}$. After initializing the policy in some arbitrary way, the algorithm processes the consecutive tasks of the shaping sequence in n_c cycles. Each task is used for n_e successive training episodes. In the online temporal difference learning variant considered here (see Section 2.2), policy improvement is interleaved with training environment exploration. As a result, each training episode consists in taking a series of actions leading to a terminal state with a single learning step taking place after each state transition. The policy learning step in line 10 can be implemented according to a concrete TD-update rule (cf. Section 2.2.2). Note also that in the considered pseudocode learning is realized in an on-policy manner, i.e., a policy being learned is also employed to take actions in the training environments.

Algorithm 8.1 Temporal difference learning from a shaping sequence

Require: $\mathbf{s} = \{\tau_1, \tau_2, \ldots, \tau_m\}$, n_c, n_e
function $\mathcal{L}(\mathbf{s})$
  $\pi_{\mathbf{s}} \leftarrow$ INITIALIZE POLICY()
  for $c = 1$ **to** n_c **do**
    for all $\tau_i \in \mathbf{s}$ **do**
      for $e = 1$ **to** n_e **do**
        $s \leftarrow$ INITIALIZE STATE(τ_i)
        while $\neg$IS TERMINAL STATE(s) **do**
          $a \leftarrow \pi_{\mathbf{s}}(s)$
          $s', r \leftarrow$ TAKE ACTION(s, a, τ_i)
          $\pi_{\mathbf{s}} \leftarrow$ TD-UPDATE POLICY($\pi_{\mathbf{s}}, (s, a, r, s')$)
        end while
      end for
    end for
  end for
  return $\pi_{\mathbf{s}}$
end function

8.1.3 *Coevolutionary Selection of Shaping Sequences*

The selection of training tasks included in the shaping sequence is essential for the effectiveness of the learning process. Exposing the learner to the right training experience is particularly important when a problem involves a large state space or when the learner has limited learning capacity and can easily forget what it once has learned. In such situations the shaping sequence can guide the training experience gathering phase towards such parts of the common state space S that are most representative. Clearly, one training task may allow the learner to sample transitions from certain areas of the state space S while another task can make reaching these areas impossible.

In absence of objective guidelines that would help choosing the shaping sequence, we delegate this task to an evolutionary algorithm, which maintains a population of individuals, each defining a sequence of tasks. Evaluating fitness of individuals consists of two phases. In the first phase (policy derivation), each task sequence is mapped to the policy by means of the learning algorithm 8.1, i.e., $\pi_{\mathbf{s}} = \mathcal{L}(\mathbf{s})$. In view of such mapping, the shaping sequence $\mathbf{s}$ can be regarded as the genotype of an individual, while its phenotype is the policy $\pi_{\mathbf{s}}$ derived from $\mathbf{s}$, i.e., trained on tasks from this sequence. In the second phase (policy evaluation), the policies created in this way are evaluated on the target task τ_0, possibly multiple times if indeterminism is involved. The fitness of an individual could be defined as, e.g., the average reward obtained by $\pi_{\mathbf{s}}$ in a series of such episodes in τ_0. In this way, the evolutionary process searches the space of training task sequences towards an optimal shaping sequence (cf. Equation 8.3).

Two phases of sequence evaluation

Algorithm 8.2 Coevolutionary optimization of shaping sequences.

1: $\mathcal{P} \leftarrow$ POPULATION OF RANDOM SHAPING SEQUENCES
2: **while** $\neg$ TERMINATION CONDITION **do**
3: $\quad \Pi \leftarrow \emptyset$
4: $\quad$ **for all** $\mathbf{s} \in \mathcal{P}$ **do**
5: $\quad\quad \pi \leftarrow \mathcal{L}(\mathbf{s})$
6: $\quad\quad \Pi \leftarrow \Pi \cup \pi$
7: $\quad$ **end for**
8: $\quad \mathcal{F} \leftarrow$ ROUND-ROBIN TOURNAMENT(Π)
9: $\quad \mathcal{S} \leftarrow$ SELECT BEST SHAPING SEQUENCES($\mathcal{P}$, $\mathcal{F}$)
10: $\quad \mathcal{P} \leftarrow$ RECOMBINE AND MUTATE($\mathcal{S}$)
11: **end while**
12: **return** GET BEST SHAPING SEQUENCE($\mathcal{P}$)

Evaluation in a competitive environment

However, since we apply the above optimization procedure in a *competitive environment* [5], we can restrain the evolutionary algorithm from using the target task for fitness evaluation purposes. Instead of maximizing the expected reward in a single static task, a competitive nature of the domain allows us to evaluate policies in the environments formed by other policies being learned in parallel. This idea can be implemented using coevolutionary algorithms (see Section 4.2.1), in which the fitness of an individual depends on the outcomes of its interactions with the other individuals in the population.

Single-population coevolution

Technically, we implement single-population competitive coevolution [154] which is presented in Algorithm 8.2. After initializing the population of shaping sequences with randomly created variations of the target task, at each generation the set of policies Π is derived from population individuals (lines 3-7) by means of the learning algorithm $\mathcal{L}$ described in Section 8.1.2. In the policy evaluation phase, each strategy $\pi_{\mathbf{s}} \in \Pi$ derived from the shaping sequence $\mathbf{s}$ plays a round-robin tournament with all the other policies in Π (line 8). Total scores received by particular policies determine the vector of their fitness values $\mathcal{F}$, which is then used to select the corresponding shaping sequences as the parents of the next evolutionary generation.

Dual approach to learning

Disregarding the choice of the algorithm used for optimizing shaping sequences, the proposed approach can be then considered dual with respect to traditional methods of policy learning. Rather than aiming at acquisition of maximum knowledge from the target task by, e.g., tuning the parameters of the training algorithm, the focus of the method is on shaping, i.e., exposing the learner to the 'right' training experience represented by selected variations of the original task. In this context, the choice of the actual learning algorithm $\mathcal{L}$ is of secondary importance: its parameters, if any, remain fixed during the entire training process, and it only serves as a means to assess the usefulness of particular sequence of training tasks.

8.2 SHAPING TASK SEQUENCES IN THE OTHELLO DOMAIN

The critical question one needs to answer to implement the proposed form of shaping is: how to create the variations of the target task τ_0 that could expose the learner to different training experience than that provided by the original task? Importantly, the tasks from the family $\mathcal{T}$ of such variations are used to build shaping task sequences. In this section we answer the above question by describing possible modifications of the assumed target task τ_0, which consists in playing Othello against a predefined opponent (see Section 5.1).

8.2.1 *Initial State Shaping Sequences*

One possible variation of the target task τ_0 consists in modifying its initial state distribution. In the case of Othello, the state transition graph is acyclic and thus a state space S can be regarded as a tree with the initial game state s_0 situated in its root. Typically, the tasks in this domain have one-point initial state distribution at s_0 and concern playing full games, starting from s_0 being the default initial board state (see Fig. 5.1a) and ending when the board is full or no player can make a move. If such a target task is used directly for training purposes, the set of states traversed in a single training episode is a directed path from the root to a leaf in the game tree.

Default initial state

The conventional training scenario based on full games seems obvious, as, in the end, we want to learn a policy capable of solving the entire problem (i.e. playing the game from its beginning till the end). However, for many problems the number of states that can be reached in the initial steps of problem solving is low, and grows exponentially with subsequent steps. As a result, a learner that starts from s_0 is doomed to overexplore the initial stages of problem solving while underexploring the final ones. In this context, changing the initial state of training tasks may be beneficial, as it allows to gather training experience in regions of the game tree that are rarely reached when starting from the default initial state.

Motivation for changing initial states

For the above reasons, we consider shaping by changing the initial state distribution. In this approach, given a target MDP task $\tau_0 = \langle S, A, T, R, I, \gamma \rangle$, training is conducted on a sequence $\mathbf{s}$ of tasks selected from the family $\mathcal{T}$ that contains all possible variations of τ_0 of the form $\tau = \langle S, A, T, R, I', \gamma \rangle$. We will refer to such a sequence as *initial state shaping sequence*. Since we assume that both the original I and modified I' initial state distributions are one-point distributions, each task in the initial state shaping sequence can be represented by a single initial state. Specifically in Othello, an initial state corresponds to a subtree of the game tree and a unique endgame limited to this subtree. Importantly, endgames naturally include the final rewards, which are essential to do any learning at all.

Shaping sequences

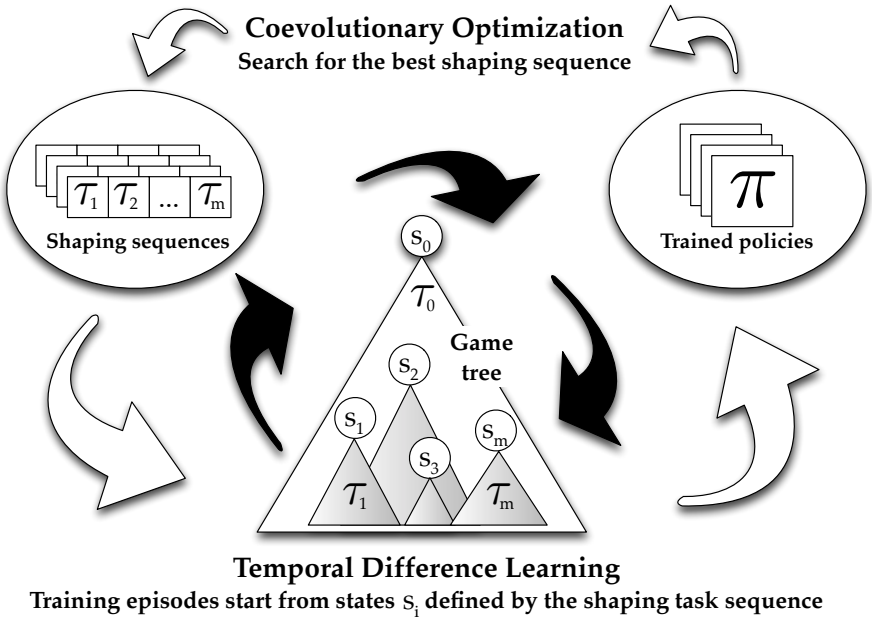


Figure 8.2: A conceptual diagram of coevolutionary optimization of shaping sequences containing tasks with modified initial states.

Optimizing initial state shaping sequences

The abstract scheme of optimizing initial state shaping sequences is illustrated in Figure 8.2. The outer loop corresponds to Algorithm 8.2 which basically performs a population-based search in the space of possible shaping sequences. Each sequence contains m training tasks which differ only with respect to their initial states. The inner loop illustrates the temporal difference learning algorithm 8.1 used to derive a single policy $\pi_{\mathbf{s}}$ from each shaping sequence $\mathbf{s}$ in the population, by playing games starting from states in $\mathbf{s}$. The learning algorithm iterates over the elements of the sequence n_c times (cf. Algorithm 8.1). The performance of the policy $\pi_{\mathbf{s}}$ calculated by playing against the other policies obtained in this way is assigned as a fitness to the shaping sequence $\mathbf{s}$. Thus, we evaluate the shaping sequences by judging the relative performance of policies created with their guidance.

8.2.2 *Opponent Shaping Sequences*

Another way of shaping in the Othello domain consists in providing tasks that differ with respect to the game-playing opponent. This approach can be useful, e.g., if the original opponent is an expert level player and the learner hardly ever receives any rewards, which are necessary ingredient of useful training experience. In the considered shaping approach, the learner is allowed to gather training experience in a sequence of tasks with different (potentially less demanding) opponents.

Changing the opponent in a game-playing task may expose the learner to completely different training experience than that provided by the original task. In this context, an opponent acts as a trainer which attempts to guide the learner through pedagogical paths in the game tree. The role of such trainers is particularly important in large state spaces. Indeed, if the learner can not visit the entire state space, it should at least be directed to such regions from which it can learn the most.

Motivation for changing opponents

Since changing the opponent influences the transition function of the MDP task, the considered task variations take the form $\tau = \langle S, A, T', R, I, \gamma \rangle$. We will call a sequence of such tasks an *opponent shaping sequence*. Technically, each task in such a sequence can be represented simply as an opponent policy.

8.3 EXPERIMENTAL SETUP AND RESULTS

In this section we provide the empirical evidence of the effectiveness of learning from shaping task sequences. In the experiments, we assume that the target task τ_0 concerns playing the Othello game against the standard WPC heuristic player (SWH) described in Section 5.1.3.1. We consider two types of shaping sequences discussed in previous section, which differ in the task aspect being modified, namely initial state shaping sequences and opponent shaping sequences.

The complete process of developing a game-playing policy using the considered shaping approach involves two stages. First, the coevolutionary method from Section 8.1.3 attempts to evolve the optimal shaping sequence for the given learning algorithm $\mathcal{L}$ (see Algorithm 8.1). Second, the best sequence $\mathbf{s}$ found in this way is employed to learn a policy $\pi = \mathcal{L}(\mathbf{s})$, which becomes the final outcome of the overall learning process. Note that although the same learning algorithm $\mathcal{L}$ is employed in both stages, it can use different number of training episodes ($t = n_e \times n_c$) in each of them. For instance, it is reasonable to use lower number of training episodes (denoted as t_1) in the first stage, for roughly estimating the fitness of shaping sequences, and larger (t_2) in the second stage, for learning the final policy.

Two stages of policy development

The performance of developed policies is measured as the percentage of points (according to Othello League scoring scheme, cf. Table 5.3) obtained in 1 000 games (500 as black and 500 as white) against the SWH player (see Section 5.1.3.1). Since all policies in our experiments are deterministic, as well as the game of Othello itself, we force both players to make random moves with probability $\epsilon = 0.1$.

Performance measure

In the following we first describe the experimental settings of coevolutionary shaping sequence optimization process and details of the algorithm $\mathcal{L}$ used to devise policies from evolving task sequences. Then we experimentally compare the performance of learning from optimized shaping sequences to training directly in the target task.

Finally, we investigate the properties of the best sequences to find out what kind of tasks are most advantageous to use as the source of training experience.

8.3.1 *Experimental Setup*

POLICY REPRESENTATION To represent game-playing policies for Othello, we employ weighted piece counters (WPCs, Section 5.1.2.2).

LEARNING FROM SHAPING SEQUENCES To learn a game-playing policy $\pi = \mathcal{L}(\mathbf{s})$ from the shaping sequence $\mathbf{s}$ we employ the algorithm presented in Algorithm 8.1. Before learning, policies are initialized by setting all WPC weights to zeroes. The weights are modified after every observed transition by the gradient-descent $TD(0)$ temporal difference update rule with the learning rate parameter set to $\alpha = 0.01$ (cf. Equation 2.14). $TD(0)$ configured in this way was previously applied for Othello [120, 192], proving capable of producing relatively good players in short training times.

TD update rule

Given a shaping sequence $\mathbf{s}$ consisting of m tasks and a total number t of training games to play, we considered two schemes of repeating tasks from a shaping sequence (cf. Algorithm 8.1). In the *sequential* scheme there is only one cycle (i.e., $n_c = 1$) and each task τ_i is used for the successive $n_e = t/m$ training episodes. Conversely, in the *cyclic* scheme, there are $n_c = t/m$ cycles, each of which consists in experiencing τ_i just once. We decided to use the cyclic scheme which proved more effective in the preliminary experiments.

Repeating tasks from a shaping sequence

The number of training games used by $\mathcal{L}$ in the first stage (evaluating fitness of a shaping sequence) is set to $t_1 = 5\,000$, while in the second stage (learning a final policy) to $t_2 = 10\,000$. The number m of tasks in a shaping sequence varies between initial state sequences and opponent sequences and is equal to 50 and 5, respectively.

The number of training games

SHAPING SEQUENCE OPTIMIZATION Shaping sequences forming input to the learning algorithm $\mathcal{L}$ are optimized by the means of the coevolutionary algorithm 8.2. The initial population comprise 50 shaping sequences, each composed of m tasks featured either by a modified initial state or by a different game-playing opponent. Coevolutionary run involves 100 generations bred by crossover followed by mutation. Both operators depend on the type of shaping sequences and are described in the corresponding sections below. Evaluation consisted of learning policies from all the shaping sequences and playing a population-wide round-robin tournament between policies created in this way. The policies scored 3, 1, or 0 points for winning, drawing, and losing, respectively. The total score earned in the tournament forms individual's fitness, which was then subject to tournament selection of size 5.

Coevolutionary algorithm parameters

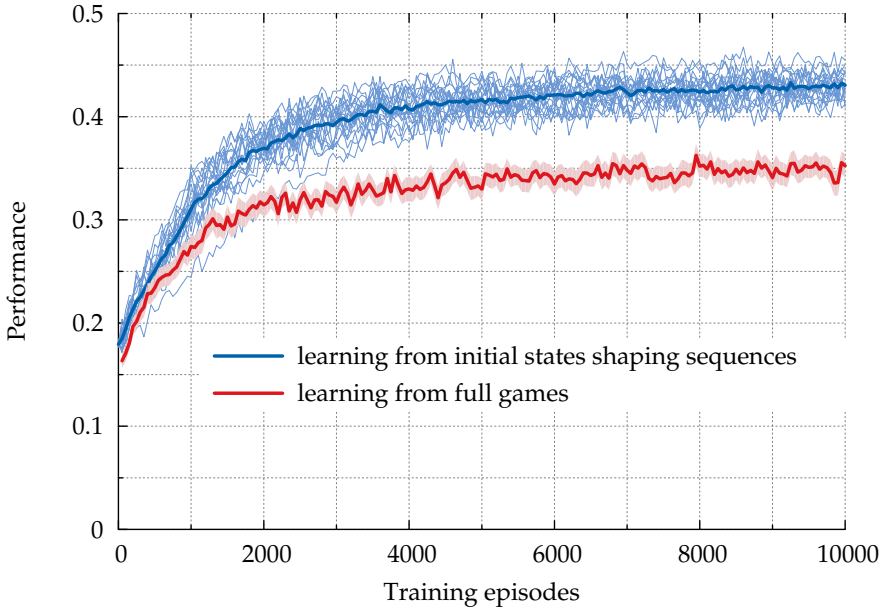


Figure 8.3: The average performance of the learners trained with the best *initial state shaping sequences* (one plot per sequence) vs. the average performance of the learners trained from full games, as a function of the number of training episodes. The performance is calculated as the average score against the SWH player.

8.3.2 *Initial State Shaping Sequences*

In the first experiment we applied the shaping approach based on initial state modification (see Section 8.2.1). Each initial state shaping sequence contained 50 tasks. Technically, they were represented only by their initial states, as all other aspects of training tasks remain the same. Initially, the states were randomly sampled from Othello games played between two random players. The crossover operator was uniform and homologous, so an offspring inherited 25 randomly selected states from the first parent and the rest from the second one, and the order of states was preserved. Mutation was applied to the offspring with probability 0.05 per state and consisted in replacing a state with a newly generated random state.

Mutation and crossover

Although the ultimate goal of learning was to play well against the SWH player, the training was realized without reference to this handcrafted strategy. Instead, we employed the *self-play* training paradigm (cf. Section 6.2.1), which allows to learn a policy autonomously, with zero knowledge built in. Thus, learning a policy $\pi = \mathcal{L}(\mathbf{s})$ from the initial state shaping sequence $\mathbf{s}$ involved a series of self-teaching episodes, in which policy being learned played against itself starting from a particular state in a sequence[1].

Self-play training paradigm

1 Consequently, training tasks differ from the target task with respect to both the opponent and the initial state.

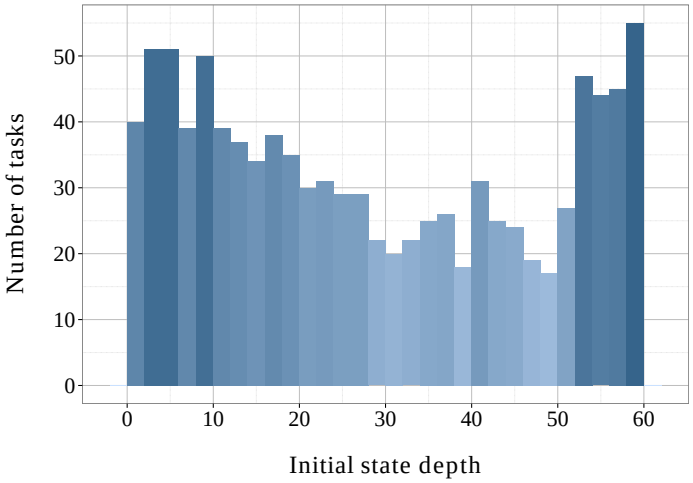


Figure 8.4: The histogram showing the distribution of tasks in the best initial state shaping sequences with respect to the depth of their initial states (20 sequences $\times$ 50 tasks $=$ 1000 tasks in total).

Performance comparison

Figure 8.3 visualizes the performance of learning from the best initial state shaping sequences compared with learning from full games (in both cases self-play training is employed). Let us emphasize that this graph characterizes the post-evolution training based on best-of-run individuals (best state sequences found). Every thin blue curve depicts the mean performance of a strategy trained using the best shaping sequence found in one of 20 coevolutionary runs. Each training episode corresponds to collecting experience through self-play from a single initial state in a sequence interleaved with online learning from this experience with the $TD(0)$ algorithm. Since shaping sequences are processed in the cyclic scheme, the k-th training episode corresponds to learning from $k \mod m$ element in the sequence.

The thick red line in Fig. 8.3 depicts the behavior of the standard self-play TDL, starting always from the default initial state s_0, which gathers experience by ϵ-greedy action selection scheme (with ϵ equal to 0.1). The unshaped version of TDL clearly stalls much earlier than the shaping approach, and attains substantially worse performance at the end of training.

Inspecting depth of the best initial states

Finally, we investigated the best initial state shaping sequences produced by each of the 20 coevolutionary runs to see how their tasks differ with respect to the depth of initial states. Figure 8.4 demonstrates that almost half of the considered $20 \times 50 = 1000$ shaping tasks have their initial states either very shallow (below the depth of 10) or very deep (over the depth of 50) in the game tree (a typical Othello game involves roughly 60 moves). Such distribution can be explained by the fact that the tasks starting from deep initial states are relatively information-rich, as there are only a few actions to be taken before receiving a reward (and credit assignment is thus easier). Shallow

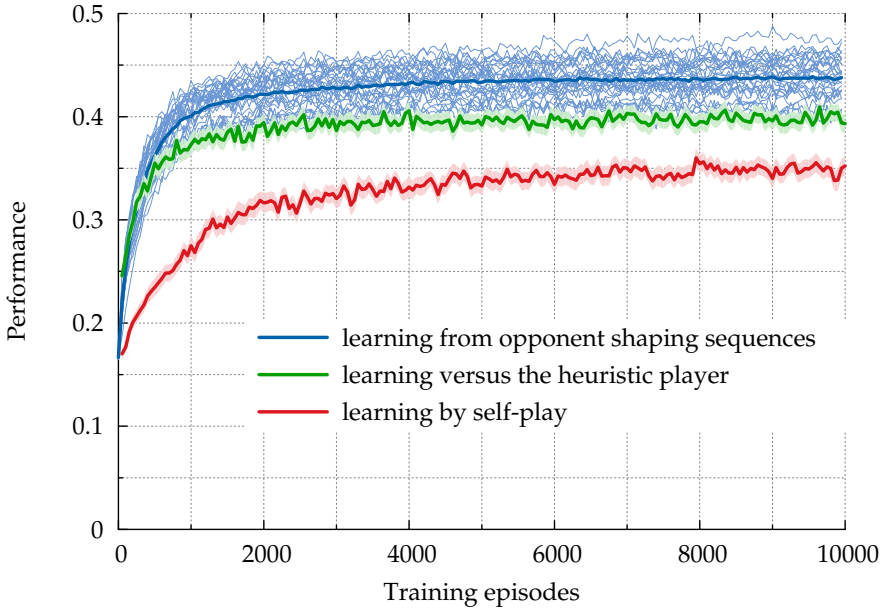


Figure 8.5: The average performance of the learners trained from the best *opponent shaping sequences* (one plot per sequence) vs. the average performance of the learners trained against the target SWH opponent or against itself (through self-play), as a function of the number of training episodes. The performance is calculated as the average score against the SWH player.

states, on the other hand, resemble the ultimate goal of learning, i.e., playing full games starting from the state s_0 situated at the depth of 0 in the game tree.

8.3.3 *Opponent Shaping Sequences*

In the second experiment we verify the effectiveness of training on selected variations of the target task which this time was modified by exchanging the game-playing opponent. In contrast to initial state modification, here all training episodes were started from the default initial state s_0, and only the training opponents could vary. Opponent shaping sequences contained 5 tasks with varying opponent policies represented as shared weighted piece counters (SWPCs, see Section 5.1.2.3). At the start of coevolution, the weights of policies were initialized by drawing them randomly from the range $[-10, 10]$. Under the uniform crossover operation applied here, an offspring inherited each element of the sequence from the first or the second parent, with equal probability, while preserving the order of elements. Mutation was applied to the offspring with probability 0.2 per weight and consisted in adding a random value from the range $[-0.1, 0.1]$ to the given weight.

Genetic operators for opponent sequences

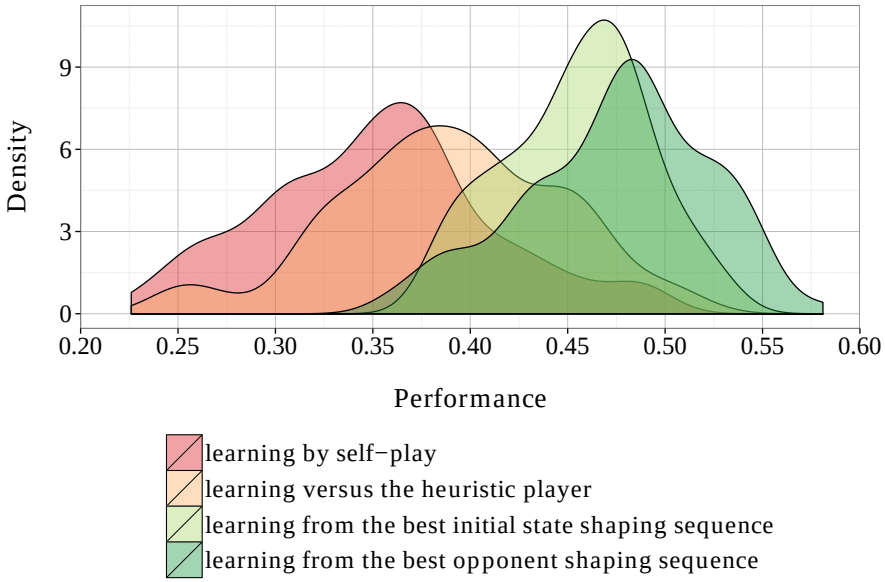


Figure 8.6: Estimated density of the distribution of final performance achieved with conventional learning methods and through the use of shaping methods.

Measuring performance

Figure 8.5 shows the performance achieved by learning from the best opponent shaping sequences. Analogously to Fig. 8.3, this reflects the post-evolution process of $TD(0)$ learning, not evolution itself. The performance was measured as the expected cumulative reward obtained in a target task, which consisted in playing a full Othello game against the SWH player. Every thin blue curve depicts the mean performance of a policy trained using the best shaping sequence found in one of 30 coevolutionary runs. Each training episode consisted in a game with a consecutive opponent specified by the shaping sequence, followed by learning from the observed state transitions. Clearly, the policies trained with the shaping approach were able to outperform not only those obtained by the basic self-play TDL but also those trained directly against the target opponent.

Distributions of the final performance

To verify whether the obtained results were statistically significant we estimated the distribution of the final performance of particular methods. Figure 8.6 illustrates the Gaussian kernel density estimates[2] based on the performance achieved after 10 000 training episodes in 100 runs. Since the shapes of particular distributions were similar (except for any difference in medians), we conducted the Kruskal-Wallis test followed by a post-hoc analysis using one-sided Mann-Whitney tests with Holm correction. The tests confirmed that learning from the best shaping sequence resulted in significantly higher performance ($p < 0.01$) than that achieved by learning either by self-play or against the target opponent.

2 The analyses were conducted using R version 2.5.13 (http://www.r-project.org).

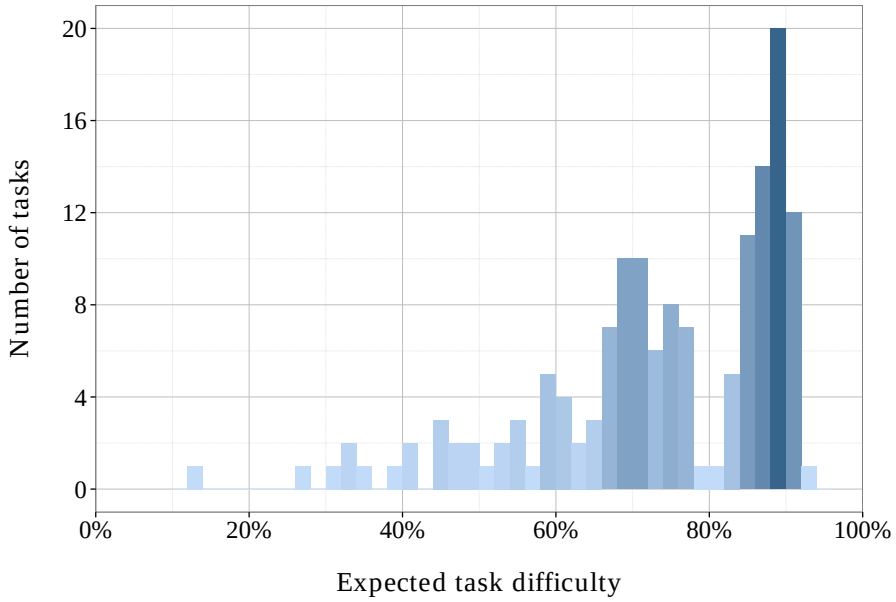


Figure 8.7: The histogram showing the distribution of tasks in the best opponent shaping sequences with respect to their expected difficulty (30 sequences $\times$ 5 tasks $=$ 1000 tasks in total).

Features of the best shaping sequences

Similarly to the previous section, we investigated the characteristics of the best evolved opponent shaping sequences. Here we measured the expected difficulty (cf. Section 7.2.4) of the tasks included in the shaping sequence, which in this particular case corresponded to the strength of the opponents specified by such sequence. Figure 8.7 visualizes the histogram based on the total of $30 \times 5 = 150$ tasks contained in the best opponent shaping sequences from all the coevolutionary runs. Clearly, the distribution is concentrated mostly on the relatively challenging tasks of difficulty over 60%. Such distribution confirms our previous observations from Section 7.4 that stronger opponents are generally more instructive and thus more useful for training.

Difficulty distribution within opponent sequences

Additionally, we assessed also the difficulty distribution within the particular opponent shaping sequences. Figure 8.8 shows the heat map corresponding to difficulties of the tasks in the best found sequences. By inspecting the columns of the figure, we can observe that the tasks within particular shaping sequences are quite diversified in terms of their difficulty. Most sequences include at least one weaker opponent representing difficulty of around 50% or even lower. Nevertheless, no general pattern across the shaping sequences is evident here. We expect that difficulty is not enough to judge the pedagogical character of the task. Other measures of task 'instructiveness' could be proposed based on, e.g., evaluating the diversity of the paths traversed in the game tree when playing against particular opponents.

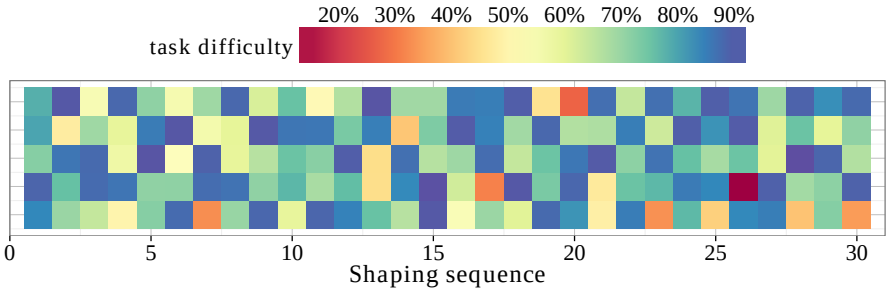


Figure 8.8: The heat map illustrating the distribution of task difficulties in particular opponent shaping sequences. Each column of the map corresponds to a single shaping sequence composed of 5 tasks.

8.4 DISCUSSION

The main premise of the proposed approach is that training a policy on a well-assorted, properly diversified and representative set of tasks can be more beneficial than confronting it directly with the target task. The reason why learning from a shaping task sequence can be advantageous is that the learner can be guided to informative state transitions which allow to extract general knowledge useful in the entire problem domain, including the target task.

Different forms of guidance

In learning game-playing strategies, it is typically the role of a *trainer* to guide the learner through the paths of the game tree from which it can learn the most [53]. This chapter revolved around the observation that such a guidance can take on different forms. In general, we are not looking for an ideal trainer here, but for an ideal training experience. In the most natural setting, the experience is embodied by a sequence of opponents (see Section 8.3.3). By coevolving such sequences we tried to identify the most advantageous opponents in terms of the relative performance of learners they instruct. Interestingly, the coevolutionary selection pressure favored sequences comprised mainly of skillful opponents, which apparently allowed learners to excel in a competitive environment.

In another shaping approach proposed here (see Section 8.3.2), the role of guidance is delegated to a sequence of initial states. Self-play training started from these states focuses the training experience on the corresponding subtrees of the game tree. In particular, to alleviate the problem of delayed reinforcement, it is beneficial to start deep in the game tree and consider only a short sequence of decisions followed by a reward. Eventually, the goal is to shape the learning process so that it produces proficient learners prepared to perform well in every region of the environment. This goal has been attained in this study for the game of Othello: rephrasing a learning task in a way that enables shaping led to better performing players.

We expect that learning from a pre-selected experience will converge faster and improve the final performance of the trained agents also for other interactive domains beyond Othello. Additionally, the dual problem definition can bring even more benefits. Firstly, the identified training tasks are a valuable source of knowledge about problem structure. Indeed, a sequence of task variations can be considered as an analog to the concept of *underlying objectives* of the problem [41, 92], which here can be interpreted as the set of skills needed to successfully operate in the given environment. Secondly, diversification of training experience embodied by shaping sequences is a natural answer to the *exploration-exploitation* trade-off (see Section 2.1). Performing random moves to explore the environment (for instance, according to the ϵ-greedy action selection scheme) may no longer be needed if the shaping sequence leads to gathering experience in different parts of the state space.

Underlying objectives

Throughout this chapter we have ignored the computational effort needed by the proposed coevolutionary method to optimize shaping sequences. It is worth to note that finding useful shaping sequences requires running the learning algorithm many times and thus is generally much more computationally demanding than the conventional reinforcement learning approaches that rely on training directly in the target task. Therefore, the practical significance of the proposed method is limited. Nevertheless, this contribution can be considered as a proof of concept that qualitative improvement of agent's performance is possible solely by adaptively modifying selected aspects of the environment it operates in.

Computational effort

Moreover, once the shaping sequence is synthesized it can be reused many times for arbitrary number of training episodes and potentially also for different learning algorithms. For instance, training tasks from the best shaping sequences could be used as a means of evaluating the fitness of policies in evolutionary learning algorithms. On the other hand, it would be interesting to verify whether shaping distributions that were found successful in the previous chapter would prove useful for another learning algorithm, such as temporal difference learning considered here.

Reusing shaping sequences

9

CONCLUSIONS

The central motivation behind this thesis can be rephrased with the following questions. Given a reinforcement learning algorithm and a target task for which a decision-making policy is to be learned:

- How and to what extent can we improve learning effectiveness (in terms of the performance on the target task) without changing the algorithm itself?
- What type of domain knowledge is required to increase the learning effectiveness?

In this thesis, we have attempted to answer these questions by referring to the concept of shaping borrowed from behavioral psychology. Rather than modifying the learning algorithm, the shaping approach consists in changing the training environment and exposing the learner to the more informative training experience than that available directly in the target task. By doing so we expected to provide an easier and faster path to learning, allowing given algorithm to reach the desirable areas in solution space.

The take-away message from this dissertation is that by appropriately changing the training environments it is possible to significantly improve the learning process realized by off-the-shelf reinforcement learning methods. In particular, we have employed different types of neural networks to represent policies and two classes of model-free reinforcement learning algorithms, namely, evolutionary learning and temporal difference learning. We have empirically demonstrated that the proposed shaping methods can significantly improve the effectiveness of these algorithms in three nontrivial reinforcement learning domains. Importantly, none of the introduced shaping methods required manual construction of training tasks. Particularly, the coevolutionary shaping approach was the most autonomous in this respect. This stays in sharp contrast with most of the previous approaches to shaping which relied on human supervision and extensive knowledge of problem domain.

The original shaping approach applied in animal training concerns simplifying the task that is too difficult to be learned directly. However, the experiments conducted in game-based domains reveal that in order to gain general game-playing abilities it is actually beneficial to focus on more difficult opponents rather than the easier ones. This observation follows the intuitive belief that we can learn more from a skillful teacher than from a beginner.

9.1 CONTRIBUTIONS

The main contributions of this thesis may be summarized as follows:

- Presentation of the unified shaping framework which embraces existing shaping approaches and outlines the role of shaping with respect to the basic reinforcement learning framework. We introduced a class of test-based reinforcement learning problems and discussed how approaching such problems with coevolutionary algorithms conforms to our notion of shaping.
  [Chapter 4]

- Demonstration of the synergy between evolutionary search performed by coevolution and gradient-based search carried out by self-play temporal difference learning. By hybridizing these two forms of shaping, the proposed method of coevolutionary temporal difference learning outperformed its both constituents in two game-playing domains. In particular, we show that using temporal difference learning is crucial to support coevolution in highly-dimensional search spaces.
  [Chapter 6]

- Demonstration of discrepancies between policies' assessments obtained using various performance measures. In particular, we showed that coevolutionary shaping methods exposed learners to richer training experience, that allowed them to achieve higher generalization performance than that of learners trained in a single environment. Moreover, we demonstrated that attaining much higher performance on demanding environments may be not sufficient to gain a substantial advantage in terms of the expected utility in the entire spectrum of possible tasks.
  [Chapters 6 and 7]

- Introduction and formalization of the measure of reinforcement learning task difficulty and the notion of difficulty distribution.
  [Chapter 7]

- Design and verification of a set of difficulty-based shaping methods that provide training tasks from a precomputed task pool according to either static or dynamic difficulty distribution. Most of them were able to significantly improve the results achieved by the conventional unshaped reinforcement learning approach. Additionally, we analyzed the coevolutionary shaping methods that do not require creating difficulty-based task pool but instead were able to autonomously discover the useful training environments on the go. Moreover, the introduced measure of task difficulty allowed us to inspect the difficulty distribution of the training tasks provided by coevolution.
  [Chapter 7]

- Formalization of the problem of optimal shaping task sequence. This leads to mapping the original learning problem of optimizing an agent's policy into a dual problem of finding the best input for the given learning algorithm, while maintaining the ultimate goal of learning.
  [Chapter 8]

- Design and verification of a dedicated coevolutionary algorithm for the problem of optimal shaping task sequence. The experimental results demonstrate that temporal difference learning from pre-selected training experience found by this algorithm can be significantly more effective than learning from a raw unshaped experience.
  [Chapter 8]

- Implementation of the proposed shaping methods as a common software framework. The framework is easy to extend and allows flexible experiment definition. The software integrates with a well-known ECJ system and, thereby, may be helpful for many ECJ users.

9.2 FUTURE WORK

The work presented in this thesis may be extended in many directions. Let us point out a few of them in the following list:

- A hybrid method of coevolutionary temporal difference learning presented in Chapter **6** relies on a single-population coevolutionary algorithm and self-play temporal difference learning. An extension of interest might be to employ two-population coevolution coupled with cross-population temporal difference learning. Evolving training environments in a separate population would allow for more flexible shaping schemes such as those applied by difficulty-based shaping methods introduced in Chapter 7.

- The performance assessments presented in Chapters 6 and 7 demonstrate that some policies perform better in easier tasks while the others excel in more demanding environments. Aggregating the policy performance to a single scalar value results in losing the information about these characteristics. This points to the need of more detailed, multi-criteria performance assessment that could illustrate how a given policy copes with tasks of various difficulty. Importantly, such multi-objectivization could be exploited not only to evaluate an already learned policy but also to drive the evolutionary learning process.

- The shaping methods proposed in Chapters 7 and 8 provided training tasks that were supposed to facilitate specific reinforcement learning algorithms. An interesting area for future work is to investigate whether training difficulty distributions or concrete shaping task sequences that were found useful in the context of a given learning algorithm could be successfully reused to improve the effectiveness of another algorithm. This would confirm that identified training tasks are not algorithm-specific but represent a general knowledge about the problem structure and its underlying objectives.

A

STATISTICAL SIGNIFICANCE

A.1 OTHELLO OPPONENT DOMAIN

	overlapped(50, 70, 2, 30)	uniform(50, 90)	cyclic(50, 90, 2, 10)	overlapped(50, 80, 5, 15)	cyclic(50, 90, 10, 5)	overlapped(50, 70, 10, 20)	staged(50, 90, 2)	staged(50, 90, 10)	staged(50, 90, 20)
uniform(50, 90)	1.000								
cyclic(50, 90, 2, 10)	1.000	1.000							
overlapped(50, 80, 5, 15)	1.000	1.000	1.000						
cyclic(50, 90, 10, 5)	0.920	1.000	1.000	1.000					
overlapped(50, 70, 10, 20)	0.920	1.000	1.000	1.000	1.000				
staged(50, 90, 2)	0.124	0.374	1.000	1.000	1.000	1.000			
staged(50, 90, 10)	0.056	0.130	0.374	0.722	1.000	1.000	1.000		
staged(50, 90, 20)	0.000	0.000	0.001	0.002	0.024	0.093	0.186	0.561	
unshaped	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.135

Table A.1: **Multi-stage** shaping methods compared in the Othello opponent domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

	uniform(50, 90)	performance(50, 90, 4, 40)	performance(50, 90, 4, 30)	performance(50, 90, 4, 20)	performance(50, 90, 4, 10)	distinctions(50, 90, 2)	distinctions(50, 90, 4)	staged(50, 90, 4)	distinctions(50, 90, 8)
performance(50, 90, 4, 40)	1.000								
performance(50, 90, 4, 30)	1.000	1.000							
performance(50, 90, 4, 20)	1.000	1.000	1.000						
performance(50, 90, 4, 10)	1.000	1.000	1.000	1.000					
distinctions(50, 90, 2)	0.455	1.000	1.000	1.000	1.000				
distinctions(50, 90, 4)	0.079	0.431	0.431	0.479	1.000	1.000			
staged(50, 90, 4)	0.001	0.006	0.006	0.005	0.039	0.321	0.928		
distinctions(50, 90, 8)	0.000	0.002	0.003	0.003	0.010	0.101	0.379	1.000	
unshaped	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.006	0.075

Table A.2: **Hyper-heuristic** shaping methods compared in the Othello opponent domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

	overlapped(50,70,2,30)	uniform(50,90)	uniform(50,70)	coev-task(25,50) rate=10.0	coev-diff(25,50) rate=40.0	coev-diff(25,50) rate=20.0	coev-diff(25,50) rate=10.0	coev-diff(25+25) rate=4.0	coev-diff(25,50) rate=4.0
uniform(50,90)	1.000								
uniform(50,70)	0.620	1.000							
coev-task(25,50) rate=10.0	0.265	0.620	1.000						
coev-diff(25,50) rate=40.0	0.093	0.180	0.845	1.000					
coev-diff(25,50) rate=20.0	0.029	0.080	0.620	1.000	1.000				
coev-diff(25,50) rate=10.0	0.000	0.000	0.000	0.004	0.011	0.163			
coev-diff(25+25) rate=4.0	0.000	0.000	0.000	0.005	0.010	0.121	1.000		
coev-diff(25,50) rate=4.0	0.000	0.000	0.000	0.000	0.000	0.021	1.000	1.000	
unshaped	0.000	0.000	0.000	0.000	0.000	0.000	0.620	0.822	1.000

Table A.3: **Coevolutionary** shaping methods compared in the Othello opponent domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

A.2 OTHELLO INITIAL STATE DOMAIN

	triangular(60,70,95)	uniform(65,85)	uniform(70,80)	normal(75,5)	triangular(65,70,85)	normal(70,10)	unshaped	triangular(70,80,95)	uniform(50,90)
uniform(65,85)	1.000								
uniform(70,80)	0.900	1.000							
normal(75,5)	0.605	1.000	1.000						
triangular(65,70,85)	0.605	1.000	1.000	1.000					
normal(70,10)	0.330	1.000	1.000	1.000	1.000				
unshaped	0.053	0.310	1.000	1.000	1.000	1.000			
triangular(70,80,95)	0.001	0.020	0.176	0.330	0.511	0.563	1.000		
uniform(50,90)	0.000	0.000	0.005	0.008	0.020	0.009	0.238	1.000	
normal(85,5)	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.191	1.000

Table A.4: **Single-stage** shaping methods compared in the Othello initial state domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

	overlapped(60,80,4,10)	overlapped(55,75,25,10)	cyclic(65,85,4,50)	staged(65,85,4)	uniform(65,85)	cyclic(65,85,4,10)	overlapped(65,85,4,10)	unshaped	staged(65,85,20)
overlapped(55,75,25,10)	1.000								
cyclic(65,85,4,50)	0.040	0.166							
staged(65,85,4)	0.066	0.166	1.000						
uniform(65,85)	0.016	0.053	1.000	1.000					
cyclic(65,85,4,10)	0.001	0.007	0.667	1.000	1.000				
overlapped(65,85,4,10)	0.000	0.000	0.021	0.112	0.293	0.667			
unshaped	0.000	0.000	0.010	0.046	0.167	0.421	1.000		
staged(65,85,20)	0.000	0.000	0.001	0.006	0.046	0.128	1.000	1.000	
staged(60,90,5)	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.002	0.032

Table A.5: **Multi-stage** shaping methods compared in the Othello initial state domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

	coev-diff rate=10.0	overlapped(60, 80, 4, 10)	coev-diff rate=5.0	staged(65, 85, 4)	triangular(60, 70, 95)	uniform(65, 85)	coev-diff rate=20.0	unshaped	coev-task depth-rate=5
overlapped(60, 80, 4, 10)	1.000								
coev-diff rate=5.0	1.000	1.000							
staged(65, 85, 4)	0.020	0.048	0.072						
triangular(60, 70, 95)	0.009	0.026	0.048	1.000					
uniform(65, 85)	0.004	0.013	0.031	1.000	1.000				
coev-diff rate=20.0	0.000	0.000	0.000	0.138	0.138	0.487			
unshaped	0.000	0.000	0.000	0.031	0.030	0.131	1.000		
coev-task depth-rate=5	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.003	
coev-task move-rate=5	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table A.6: **Coevolutionary** shaping methods compared in the Othello initial state domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

A.3 POLE BALANCING DYNAMICS DOMAIN

	uniform(60, 95)	uniform(70, 95)	uniform(50, 95)	uniform(70, 90)	uniform(60, 90)	uniform(50, 90)	unshaped	uniform(50, 80)
uniform(70, 95)	1.000							
uniform(50, 95)	1.000	1.000						
uniform(70, 90)	0.000	0.000	0.000					
uniform(60, 90)	0.000	0.000	0.000	0.856				
uniform(50, 90)	0.000	0.000	0.000	0.856	1.000			
unshaped	0.000	0.000	0.000	0.262	0.662	0.662		
uniform(50, 80)	0.000	0.000	0.000	0.000	0.000	0.000	0.003	
uniform(60, 80)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.108

Table A.7: **Single-stage** shaping methods compared in the pole balancing dynamics domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

	uniform(60, 95)	uniform(70, 95)	cevo-diff rate=20.0	uniform(50, 95)	cevo-diff rate=10.0	cevo-diff rate=5.0
uniform(70, 95)	1.000					
cevo-diff rate=20.0	1.000	1.000				
uniform(50, 95)	1.000	1.000	1.000			
cevo-diff rate=10.0	0.012	0.013	0.042	0.013		
cevo-diff rate=5.0	0.010	0.010	0.040	0.013	1.000	
unshaped	0.000	0.000	0.000	0.000	0.000	0.003

Table A.8: **Coevolutionary** shaping methods compared in the pole balancing dynamics domain. Table shows p-values obtained in one-sided pairwise Mann-Whitney test with Holm correction.

BIBLIOGRAPHY

[1] David Ackley and Michael Littman. Interactions Between Learning and Evolution. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 487–509. Addison-Wesley, Redwood City, CA, 1992.

[2] Riad Akrour, Marc Schoenauer, and Michele Sebag. Preference-Based Policy Learning. In *Machine Learning and Knowledge Discovery in Databases*, volume 6911 of *Lecture Notes in Computer Science*, pages 12–27. Springer Berlin Heidelberg, 2011.

[3] Charles W. Anderson. Learning to Control an Inverted Pendulum Using Neural Networks. *IEEE Control Systems Magazine*, 9 (3):31–37, 1989.

[4] Charles W. Anderson and W. Thomas Miller. A Challenging Set of Control Problems. In *Neural Networks for Control*, pages 475–508. MIT Press, Cambridge, MA, USA, 1990.

[5] Peter J. Angeline and Jordan B. Pollack. Competitive Environments Evolve Better Solutions for Complex Tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 264–270, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[6] Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[7] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[8] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning. *Machine Learning*, 23(2-3):279–303, 1996.

[9] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, 1997.

[10] Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel. Evolutionary Computation: Comments on the History and Current State. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, 1997.

[11] Michael Bain and Claude Sammut. A Framework for Behavioural Cloning. In *Machine Intelligence 15, Intelligent Agents*, pages 103–129, Oxford, UK, 1999. Oxford University.

[12] Andrew G. Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[13] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):835–846, 1983.

[14] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to Play Chess Using Temporal Differences. *Machine Learning*, 40(3):243–263, 2000.

[15] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[16] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[17] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1 (1):3–52, 2002.

[18] Kevin J. Binkley, Ken Seehart, and Masafumi Hagiwara. A Study of Artificial Neural Network Architectures for Othello Evaluation Functions. *Transactions of the Japanese Society for Artificial Intelligence*, 22(5):461–471, 2007.

[19] Alan D. Blair and Jordan B. Pollack. What makes a good coevolutionary learning environment. *Australian Journal of Intelligent Information Processing Systems*, 4(3/4):166–175, 1997.

[20] Woodrow W. Bledsoe and Iben Browning. Pattern Recognition and Reading by Machine. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 225–232, New York, NY, USA, 1959. ACM.

[21] Bruno Bouzy and Tristan Cazenave. Computer Go: An AI Oriented Survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[22] Bruno Bouzy and Bernard Helmstetter. Monte Carlo Go Developments. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *Advances in Computer Games conference (ACG-10), Graz 2003*, pages 159–174. Kluwer, 2003.

[23] Richard Bozulich. *The Go Player's Almanac*. Ishi Press, Tokyo, 1992.

[24] Anthony Bucci. *Emergent Geometric Organization and Informative Dimensions in Coevolutionary Algorithms*. PhD thesis, Waltham, MA, USA, 2007.

[25] Anthony Bucci and Jordan B. Pollack. A Mathematical Framework for the Study of Coevolution. In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Proceedings of the Seventh Workshop on Foundations of Genetic Algorithms*, pages 221–236. Morgan Kaufmann, 2002.

[26] Edmund K. Burke, Michel Gendreau, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[27] Michael Buro. Logistello: A Strong Learning Othello Program. In *19th Annual Conference Gesellschaft für Klassifikation e.V.*, 1995.

[28] Lucian Buşoniu, Robert Babuška, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Boca Raton, Florida, 2010.

[29] Kumar Chellapilla and David B. Fogel. Evolving an Expert Checkers Playing Program Without Using Human Expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428, 2001.

[30] Siang Yew Chong, Mei K. Tan, and Jonathon D. White. Observing the Evolution of Neural Networks Learning to Play the Game of Othello. *IEEE Transactions on Evolutionary Computation*, 9(3):240–251, 2005.

[31] Siang Yew Chong, P. Tino, and Xin Yao. Measuring Generalization Performance in Coevolutionary Learning. *IEEE Transactions on Evolutionary Computation*, 12(4):479–505, 2008.

[32] Siang Yew Chong, Peter Tino, and Xin Yao. Relationship Between Generalization and Diversity in Coevolutionary Learning. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):214–232, 2009.

[33] Siang Yew Chong, Peter Tino, Day Chyi Ku, and Xin Yao. Improving Generalization Performance in Co-Evolutionary Learning. *IEEE Transactions on Evolutionary Computation*, 16(1):70–85, 2012.

[34] Anders Lyhne Christensen and Marco Dorigo. Incremental Evolution of Robot Controllers for a Highly Integrated Task. In *Proceedings of the 9th International Conference on From Animals to Animats: Simulation of Adaptive Behavior*, SAB'06, pages 473–484, Berlin, Heidelberg, 2006. Springer-Verlag.

[35] Paweł Cichosz. *Systemy uczące się*. Wydawnictwa Naukowo-Techniczne, 2007.

[36] Adrian F. Clark, editor. *Proceedings of the British Machine Vision Conference 1997, BMVC 1997, University of Essex, UK, 1997*, 1997. British Machine Vision Association.

[37] Robert H. Crites and Andrew G. Barto. Elevator Group Control Using Multiple Reinforcement Learning Agents. *Machine Learning*, 33(2-3):235–262, 1998.

[38] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2 (4):303–314, 1989.

[39] Paul J. Darwen. Why Co-Evolution Beats Temporal Difference Learning at Backgammon for a Linear Architecture, but not a Non-Linear Architecture. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001)*, pages 1003–1010, Piscataway, NJ, USA, 2001. IEEE Press.

[40] Edwin D. de Jong. The MaxSolve Algorithm for Coevolution. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 483–489, New York, NY, USA, 2005. ACM.

[41] Edwin D. de Jong and Jordan B. Pollack. Ideal Evaluation from Coevolution. *Evolutionary Computation*, 12(2):159–192, 2004.

[42] Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18, 2011.

[43] Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[44] Alexander Dilger and Hannah Geyer. Are Three Points for a Win Really Better Than Two? A Comparison of German Soccer League and Cup Games. *Journal of Sports Economics*, 10(3):305–318, 2009.

[45] Kevin R. Dixon, Richard J. Malak, and Pradeep K. Khosla. Incorporating Prior Knowledge and Previously Learned Information into Reinforcement Learning. Technical report, Institute for Complex Engineered Systems, Carnegie Mellon University, 2000.

[46] Stephen Dominic, Darrell Whitley, and Charles W. Anderson. Genetic Reinforcement Learning for Neural Networks. In *Proceedings of the International Joint Conference on Neural Networks - IJCNN 91*, pages 71–76, Piscataway, NJ, USA, 1991. IEEE.

[47] Marco Dorigo and Marco Colombetti. Robot Shaping: Developing Autonomous Agents through Learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[48] John R. Dormand and Peter J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.

[49] Adam Dziuk and Risto Miikkulainen. Creating Intelligent Agents through Shaping of Coevolution. In Alice E. Smith, editor, *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1077–1083, New Orleans, LA, USA, 2011. IEEE Press.

[50] Agoston E. Eiben and Selmar K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[51] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003.

[52] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego — An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[53] Susan L. Epstein. Toward an Ideal Trainer. *Machine Learning*, 15 (3):251–277, 1994.

[54] Tom Erez and William D Smart. What does Shaping Mean for Computational Reinforcement Learning? In *7th IEEE International Conference on Development and Learning, ICDL 2008.*, pages 215–219. IEEE, 2008.

[55] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[56] Sevan G. Ficici. *Solution Concepts in Coevolutionary Algorithms*. PhD thesis, Waltham, MA, USA, 2004.

[57] Sevan G. Ficici and Jordan B. Pollack. Challenges in Coevolutionary Learning: Arms-race Dynamics, Open-endedness, and Medicocre Stable States. In *Proceedings of the Sixth International Conference on Artificial Life*, ALIFE, pages 238–247, Cambridge, MA, USA, 1998. MIT Press.

[58] Sevan G. Ficici and Jordan B. Pollack. Pareto Optimality in Coevolutionary Learning. In *Proceedings of the 6th European Conference on Advances in Artificial Life*, ECAL '01, pages 316–325, London, UK, 2001. Springer-Verlag.

[59] Sevan G. Ficici and Jordan B. Pollack. A Game-theoretic Memory Mechanism for Coevolution. In *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation: PartI*, GECCO'03, pages 286–297, Berlin, Heidelberg, 2003. Springer-Verlag.

[60] Dario Floreano and Stefano Nolfi. God Save the Red Queen! Competition in Co-evolutionary Robotics. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 398–406, San Fransisco, CA, USA, 1997. Morgan Kaufmann.

[61] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[62] David B. Fogel. Using Evolutionary Programming to Create Neural Networks that are Capable of Playing Tic-Tac-Toe. In *Proceedings of IEEE International Conference on Neural Networks*, volume 2, pages 875–880, San Francisco, CA, USA, 1993. IEEE.

[63] David B. Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[64] David Fotland. Static Eye Analysis in "The Many Faces of Go". *ICGA Journal*, 25(4):203–210, 2002.

[65] Johannes Fürnkranz and Miroslav Kubat, editors. *Machines That Learn to Play Games*. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[66] Shlomo Geva and Joaquin Sitte. A Cartpole Experiment Benchmark for Trainable Controllers. *IEEE Control Systems*, 13(5):40–51, 1993.

[67] Imran Ghory. Reinforcement Learning in Board Games. Technical Report CSTR-04-004, Department of Computer Science, University of Bristol, 2004.

[68] David E. Goldberg. Simple Genetic Algorithms and the Minimal Deceptive Problem. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence, pages 74–88. Pitman, London, UK, 1987.

[69] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[70] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated Neural Evolution Through Cooperatively Coevolved Synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[71] Faustino J. Gomez. *Robust Nonlinear Control through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2003.

[72] Faustino J. Gomez and Risto Miikkulainen. Incremental Evolution Of Complex General Behavior. *Adaptive Behavior*, (5):317–342, 1997.

[73] Faustino J. Gomez and Risto Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, pages 1356–1361, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[74] Faustino J. Gomez and Risto Miikkulainen. Active Guidance for a Finless Rocket Using Neuroevolution. In *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation: PartII*, GECCO'03, pages 2084–2095, Berlin, Heidelberg, 2003. Springer-Verlag.

[75] Faustino J. Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient Non-linear Control Through Neuroevolution. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 654–662, Berlin, Heidelberg, 2006. Springer-Verlag.

[76] Vijaykumar Gullapalli and Andrew G. Barto. Shaping as a Method for Accelerating Reinforcement Learning. In *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, pages 554–559, 1992.

[77] Nikolaus Hansen and Andreas Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[78] Elvin O. Harbin. *Games of Many Nations*. Abingdon Press; Arco Publishers, 1955.

[79] Inman Harvey, Phil Husbands, and Dave Cliff. Seeing the Light: Artificial Evolution, Real Vision. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior : From*

Animals to Animats 3: From Animals to Animats 3, SAB94, pages 392–401, Cambridge, MA, USA, 1994. MIT Press.

[80] Goro Hasegawa and Maxine Brady. *How to Win at Othello*. Let Me Read Book. Jove Publications, 1977.

[81] Ami Hauptman and Moshe Sipper. Emergence of Complex Strategies in the Evolution of Chess Endgame Players. *Advances in Complex Systems*, 10:35–59, 2007.

[82] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[83] Verena Heidrich-Meisner and Christian Igel. Hoeffding and Bernstein Races for Selecting Policies in Evolutionary Direct Policy Search. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 401–408, New York, NY, USA, 2009. ACM.

[84] Bernhard Hengst. Discovering Hierarchy in Reinforcement Learning with HEXQ. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 243–250, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[85] William Daniel Hillis. Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure. *Physica D*, 42 (1-3):228–234, 1990.

[86] Geoffrey E. Hinton and Steven J. Nowlan. How Learning Can Guide Evolution. *Complex systems*, 1(3):495–502, 1987.

[87] Jerry L. Hintze and Ray D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184, 1998.

[88] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[89] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[90] Christian Igel. Neuroevolution for Reinforcement Learning using Evolution Strategies. In *Proceedings of the 2003 Congress on Evolutionary Computation*, volume 4, pages 2588–2595, 2003.

[91] Wojciech Jaśkowski. *Algorithms for Test-Based Problems*. PhD thesis, Institute of Computing Science, Poznan University of Technology, Poznan, Poland, 2011.

[92] Wojciech Jaśkowski and Krzysztof Krawiec. Formal Analysis, Hardness, and Algorithms for Extracting Internal Structure of Test-Based Problems. *Evolutionary Computation*, 19(4):639–671, 2011.

[93] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Winning Ant Wars: Evolving a Human-competitive Game Strategy Using Fitnessless Selection. In *Proceedings of the 11th European Conference on Genetic Programming*, EuroGP'08, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[94] Wojciech Jaśkowski, Paweł Liskowski, Marcin G. Szubert, and Krzysztof Krawiec. Improving Coevolution by Random Sampling. In *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO '13, pages 1141–1148, New York, NY, USA, 2013. ACM.

[95] Yaochu Jin. A Comprehensive Survey of Fitness Approximation in Evolutionary Computation. *Soft Computing Journal*, 9(1):3–12, 2005.

[96] George Johnson. To Test a Powerful Computer, Play an Ancient Game. *The New York Times*, 1997.

[97] Hugues Juillé. *Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm*. PhD thesis, Waltham, MA, USA, 1999.

[98] Hugues Juillé and Jordan B. Pollack. Coevolutionary Learning: A Case Study. In Jude W. Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998)*, pages 251–259, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[99] Hugues Juille and Jordan B. Pollack. Coevolving the Ideal Trainer: Application to the Discovery of Cellular Automata Rules. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 519–527, San Francisco, CA, USA, 1998. Morgan Kaufmann.

[100] Andrew Moore Justin Boyan. Generalization in Reinforcement Learning: Safely Approximating the Value Function. In *Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

[101] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4(1):237–285, 1996.

[102] K.-J. Kim, Heejin Choi, and Sung-Bae Cho. Hybrid of Evolution and Reinforcement Learning for Othello Players. In *IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 203–209, Honolulu, HI, 2007.

[103] Jérôme Kodjabachian and Jean-Arcady Meyer. Evolution and Development of Neural Controllers for Locomotion, Gradient-following, and Obstacle-avoidance in Artificial Insects. *IEEE Transactions on Neural Networks*, 9(5):796–812, 1998.

[104] Aleksander Kolcz and Nigel M. Allinson. N-tuple Regression Network. *Neural Networks*, 9(5):855–869, 1996.

[105] Rogier Koppejan and Shimon Whiteson. Neuroevolutionary reinforcement learning for generalized control of simulated helicopters. *Evolutionary Intelligence*, 4(4):219–241, 2011.

[106] Clifford Kotnik and Jugal K. Kalita. The Significance of Temporal-Difference Learning in Self-Play Training TD-Rummy versus EVO-rummy. In Tom Fawcett and Nina Mishra, editors, *Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003)*, pages 369–375, Washington, DC, USA, 2003. AAAI Press.

[107] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[108] Krzysztof Krawiec and Bir Bhanu. Visual Learning by Coevolutionary Feature Synthesis. *IEEE Transactions on Systems, Man, and Cybernetics–Part B*, 35(3):409–425, 2005.

[109] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch Reinforcement Learning. In Marco Wiering and Martijn Otterlo, editors, *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 45–73. Springer Berlin Heidelberg, 2012.

[110] Edward Lasker. *Go and Go-Moku: The Oriental Board Games*. Dover Publications, 1960.

[111] Adam Laud and Gerald DeJong. Reinforcement Learning and Shaping: Encouraging Intended Behaviors. In Claude Sammut and Achim G. Hoffmann, editors, *Proceedings of the Nineteenth International Conference on Machine Learning (ICML 2002)*, pages 355–362, University of New South Wales, Sydney, Australia, 2002. Morgan Kaufmann.

[112] Kai-Fu Lee and Sanjoy Mahajan. The Development of a World Class Othello Program. *Artificial Intelligence*, 43(1):21–36, 1990.

[113] Joel Lehman and Kenneth O. Stanley. Abandoning Objectives: Evolution through the Search for Novelty Alone. *Evolutionary Computation*, 19(2):189–223, 2011.

[114] Long Lin and Tom Mitchell. Memory Approaches to Reinforcement Learning in Non-Markovian Domains. Technical report, Pittsburgh, PA, USA, 1992.

[115] Alex Lubberts and Risto Miikkulainen. Co-Evolving a Go-Playing Neural Network. In *Coevolution: Turning Adaptive Algorithms Upon Themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

[116] Simon M. Lucas. Face recognition with the continuous n-tuple classifier. In Clark [36].

[117] Simon M. Lucas. Learning to Play Othello with N-tuple Systems. *Australian Journal of Intelligent Information Processing Systems, Special Issue on Game Technology*, 9(4):01–20, 2007.

[118] Simon M. Lucas. Neural Network Othello Competition. *SIGEVOlution*, 2(4):38–40, 2007.

[119] Simon M. Lucas and Ali Amiri. Recognition of chain-coded handwritten character images with scanning n-tuple method. *Electronics Letters*, 31(24):2088–2089, 2002.

[120] Simon M. Lucas and Thomas Philip Runarsson. Temporal Difference Learning Versus Co-Evolution for Acquiring Othello Position Evaluation. In Sushil J. Louis and Graham Kendall, editors, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games, CIG 2006*, pages 52–59. IEEE, 2006.

[121] Sean Luke. *The ECJ Owner's Manual – A User Manual for the ECJ Evolutionary Computation Library*, 2010.

[122] Richard Maclin and Jude W. Shavlik. Creating Advice-Taking Reinforcement Learners. *Machine Learning*, 22(1-3):251–281, 1996.

[123] Michael G. Madden and Tom Howley. Transfer of Experience Between Reinforcement Learning Environments with Progressive Difficulty. *Artificial Intelligence Review*, 21(3-4):375–398, 2004.

[124] Jacek Mańdziuk. *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*, volume 276 of *Studies in Computational Intelligence*. Springer, 2010.

[125] Edward P. Manning. Temporal Difference Learning of an Othello Evaluation Function for a Small Neural Network with

Shared Weights. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 216–223. IEEE, 2007.

[126] Edward P. Manning. Using Resource-Limited Nash Memory to Improve an Othello Evaluation Function. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):40–53, 2010.

[127] Maja J. Mataric. Reward Functions for Accelerated Learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 181–189, New Brunswick, NJ, USA, 1994. Morgan Kaufmann.

[128] Helmut A. Mayer. Board Representations for Neural Go Players Learning by Temporal Difference. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 183–188. IEEE, 2007.

[129] David Mechner. All Systems Go. *The Sciences*, 38(1):32–37, 1998.

[130] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. Springer-Verlag, London, UK, 1996.

[131] Donald Michie and Roger A. Chambers. BOXES: An Experiment in Adaptive Control. In *Machine Intelligence*. Oliver and Boyd, Edinburgh, UK, 1968.

[132] Geoffrey Miller and Dave Cliff. Co-Evolution of Pursuit and Evasion I: Biological and Game-Theoretic Foundations. Technical Report CSRP311, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1994.

[133] Marvin Minsky. Steps toward Artificial Intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[134] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[135] German A. Monroy, Kenneth O. Stanley, and Risto Miikkulainen. Coevolution of Neural Networks Using a Layered Pareto Archive. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 329–336, New York, NY, USA, 2006. ACM.

[136] Andrew W. Moore and Christopher G. Atkeson. Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time. *Machine Learning*, 13:103–130, 1993.

[137] David E. Moriarty and Risto Miikkulainen. Efficient Reinforcement Learning Through Symbiotic Evolution. *Machine Learning*, 22:11–32, 1996.

[138] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999.

[139] Pablo Moscato. New Ideas in Optimization. chapter Memetic Algorithms: A Short Introduction, pages 219–234. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[140] Jean-Baptiste Mouret and Stéphane Doncieux. Incremental Evolution of Animats' Behaviors as a Multi-objective Optimization. In Minoru Asada, John C. T. Hallam, Jean-Arcady Meyer, and Jun Tani, editors, *From Animals to Animats 10, 10th International Conference on Simulation of Adaptive Behavior*, volume 5040 of *Lecture Notes in Computer Science*, pages 210–219, Osaka, Japan, 2008. Springer.

[141] Jean-Baptiste Mouret and Stéphane Doncieux. Overcoming the Bootstrap Problem in Evolutionary Robotics using Behavioral Diversity. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009*, pages 1161–1168, Trondheim, Norway, 2009. IEEE.

[142] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999)*, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann.

[143] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous Inverted Helicopter Flight via Reinforcement Learning. In Marcelo H. Ang Jr. and Oussama Khatib, editors, *Experimental Robotics IX, The 9th International Symposium on Experimental Robotics*, volume 21 of *Springer Tracts in Advanced Robotics*, pages 363–372. Springer, 2004.

[144] Stefano Nolfi and Dario Floreano. Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution? *Artificial Life*, 4(4):311–335, 1998.

[145] Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics. In R. A. Brooks and P. Maes, editors, *Artificial life IV: Proceedings of the 4th International Workshop on Artificial Life*, pages 190–197, Cambridge, MA, USA, 1994. MIT Press.

[146] Peter Nordin and Wolfgang Banzhaf. An On-line Method to Evolve Behavior and to Control a Miniature Robot in Real

Time with Genetic Programming. *Adaptive Behavior*, 5(2):107–140, 1996.

[147] Liviu Panait and Sean Luke. A Comparison Of Two Competitive Fitness Functions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 503–511, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[148] Gary B. Parker. The Incremental Evolution of Gaits for Hexapod Robots. In Lee Spector, Erik D. Goodman, Annie Wu, W.B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 1114–1121, San Francisco, CA, USA, 2001. Morgan Kaufmann.

[149] David S. Parlett. *The Oxford history of board games*. Oxford University Press, Oxford, 1999.

[150] Jing Peng and Ronald J. Williams. Incremental Multi-step Q-learning. *Machine Learning*, 22(1-3):283–290, 1996.

[151] Gail B. Peterson. A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82(3):317–328, 2004.

[152] Jordan B. Pollack and Alan D. Blair. Co-Evolution in the Successful Learning of Backgammon Strategy. *Machine Learning*, 32 (3):225–240, 1998.

[153] Dean A. Pomerleau. Efficient Training of Artificial Neural Networks for Autonomous Navigation. *Neural Computation*, 3(1): 88–97, 1991.

[154] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. de Jong. Coevolutionary Principles. In Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 987–1033. Springer, 2012.

[155] Mitchell A. Potter and Kenneth A. De Jong. Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[156] Doina Precup, Richard S. Sutton, and Sanjoy Dasgupta. Off-Policy Temporal Difference Learning with Function Approximation. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 417–424, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[157] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[158] Emmanuel Rachelson, Franois Schnitzler, Louis Wehenkel, and Damien Ernst. Optimal Sample Selection for Batch-Mode Reinforcement Learning. In Joaquim Filipe and Ana L. N. Fred, editors, *ICAART 2011 - Proceedings of the 3rd International Conference on Agents and Artificial Intelligence*, volume 1 - Artificial Intelligence, pages 41–50, Rome, Italy, 2011. SciTePress.

[159] Jette Randløv. Shaping in Reinforcement Learning by Changing the Physics of the Problem. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 767–774, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[160] Jette Randløv and Preben Alstrøm. Learning to Drive a Bicycle using Reinforcement Learning and Shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471. Morgan Kaufmann, San Francisco, CA, 1998.

[161] Richard Rohwer and Michał Morciniec. A Theoretical and Experimental Account of N-tuple Classifier Performance. *Neural Computation*, 8(3):629–642, 1996.

[162] Christopher D. Rosin and Richard K. Belew. New Methods for Competitive Coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[163] Gavin A. Rummery and Mahesan Niranjan. On-line Q-Learning using Connectionist Sytems. Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK, 1994.

[164] Thomas P. Runarsson and Simon M. Lucas. Co-evolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go. *IEEE Transactions on Evolutionary Computation*, 9(6):628–640, 2005.

[165] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[166] Claude Sammut and Donald Michie. Controlling a Black-Box Simulation of a Spacecraft. *AI Magazine*, 12(1):56–63, 1991.

[167] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to Fly. In Kerstin Dautenhahn and Chrystopher L. Nehaniv, editors, *Imitation in Animals and Artifacts*, pages 171–189. MIT Press, Cambridge, MA, USA, 2002.

[168] Spyridon Samothrakis, Simon M. Lucas, Thomas Philip Runarsson, and David Robles. Coevolving Game-Playing Agents: Measuring Performance and Intransitivities. *IEEE Transactions on Evolutionary Computation*, 17(2):213–226, 2013.

[169] Arthur L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 44(1):206–227, 1959.

[170] Arthur L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. II: Recent Progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[171] Jonathan Schaeffer and H. Jaap van den Herik. Games, Computers and Artificial Intelligence. *Artificial Intelligence*, 134(1-2): 1–8, 2002.

[172] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Learning to Evaluate Go Positions via Temporal Difference Methods. In Norio Baba and Lakhmi C. Jain, editors, *Computational Intelligence in Games*, volume 62 of *Studies in Fuzziness and Soft Computing*, chapter 4, pages 77–98. Springer Verlag, Berlin, 2001.

[173] Alan G. Schultz. Adapting the Evaluation Space to Improve Global Learning. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, ICGA, pages 158–165, San Diego, CA, USA, 1991. Morgan Kaufmann.

[174] Oliver G. Selfridge, Richard S. Sutton, and Andrew G. Barto. Training and Tracking in Robotics. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, IJCAI, pages 670–672, Los Angeles, CA, 1985. Morgan Kaufmann.

[175] William Shakespeare. *The Oxford Shakespeare: the complete works of William Shakespeare*. Oxford University Press, London, UK, 1914.

[176] Claude E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, Ser. 7, Vol. 41(314):256–275, 1950.

[177] Silver, David and Sutton, Richard S. and Müller, Martin. Sample-based Learning and Search with Permanent and Transient Memories. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 968–975, New York, NY, USA, 2008. ACM.

[178] Karl Sims. Evolving 3D Morphology and Behavior by Competition. *Artificial Life*, 1(4):353–372, 1994.

[179] Joshua A. Singer. Co-evolving a Neural-Net Evaluation Function for Othello by Combining Genetic Algorithms and Reinforcement Learning. In *Proceedings of the International Conference*

on Computational Science-Part II, ICCS '01, pages 377–389, London, UK, 2001. Springer-Verlag.

[180] Burrhus F. Skinner. *The behavior of organisms: An experimental analysis*. Appleton-Century, 1938.

[181] Burrhus F. Skinner. *Science and Human Behavior*. Macmillan, 1953.

[182] Matthijs Snel and Shimon Whiteson. Multi-Task Reinforcement Learning: Shaping and Feature Selection. In *Proceedings of the 9th European Conference on Recent Advances in Reinforcement Learning*, EWRL'11, pages 237–248, Berlin, Heidelberg, 2012. Springer-Verlag.

[183] Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, 2004.

[184] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[185] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time Neuroevolution in the NERO Video Game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[186] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[187] Richard S. Sutton. Introduction: The Challenge of Reinforcement Learning. *Machine Learning*, 8(3-4):225–227, 1992.

[188] Richard S. Sutton. Reinforcement Learning: Past, Present and Future. In *Selected Papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning*, SEAL'98, pages 195–197, London, UK, 1999. Springer-Verlag.

[189] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[190] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112 (1-2):181–211, 1999.

[191] Marcin G. Szubert. cECJ — Coevolutionary Computation in Java. http://www.cs.put.poznan.pl/mszubert/projects/cecj.html, 2010.

[192] Marcin G. Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Coevolutionary Temporal Difference Learning for Othello. In *Proceedings of the 5th International Conference on Computational Intelligence and Games*, CIG'09, pages 104–111. IEEE Press, 2009.

[193] Marcin G. Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning Board Evaluation Function for Othello by Hybridizing Coevolution with Temporal Difference Learning. *Control & Cybernetics*, 40(3):805–831, 2011.

[194] F. Tanaka and M. Yamamura. Multitask Reinforcement Learning on the Distribution of MDPs. In *Proceedings of the 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, volume 3, pages 1108–1113. IEEE, 2003.

[195] Matthew E. Taylor and Peter Stone. Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[196] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1321–1328, New York, NY, USA, 2006. ACM.

[197] Gerald Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8(3-4):257–277, 1992.

[198] Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[199] Edward L. Thorndike. Animal intelligence: An experimental study of thhe associative processes in animal. *Psychological Monographs: General and Applied*, 2(4):i–109, 1898.

[200] Edward L. Thorndike. *Animal Intelligence: Experimental Studies*. New York, The Macmillan Company, 1911.

[201] Sebastian Thrun and Anton Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In *Proceedings of the 1993 Connectionist Models Summer School*, pages 255–263. Lawrence Erlbaum, 1993.

[202] Sebastian B. Thrun. Efficient Exploration In Reinforcement Learning. Technical report, Pittsburgh, PA, USA, 1992.

[203] Sebastian B. Thrun. Learning to Play the Game of Chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, Cambridge, MA, 1995. MIT Press.

[204] Julian Togelius, Faustino J. Gomez, and Jürgen Schmidhuber. Learning What to Ignore: Memetic Climbing in Topology and Weight Space. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008*, pages 3274–3281, Hong Kong, China, 2008.

[205] Lisa Torrey and Jude Shavlik. Transfer Learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. 2010.

[206] Joseba Urzelai, Dario Floreano, Marco Dorigo, and Marco Colombetti. Incremental Robot Shaping. *Connection Science*, 10 (3-4):341–360, 1998.

[207] Sjoerd van den Dries and Marco A. Wiering. Neural-Fitted TD-Leaf Learning for Playing Othello With Structured Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 23(11):1701–1713, 2012.

[208] Nees Jan van Eck and Michiel van Wezel. Application of Reinforcement Learning to the Game of Othello. *Computers and Operations Research*, 35(6):1999–2017, 2008.

[209] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and Exploitation in Evolutionary Algorithms: A Survey. *ACM Computing Surveys*, 45(3):35:1–35:33, 2013.

[210] Shivakumar Viswanathan and Jordan B. Pollack. On the coevolutionary construction of learnable gradients. In *Proceedings of the 2005 AAAI Fall Symposium on Coevolutionary and Coadaptive Systems. AAAI Press*, 2005.

[211] W. Paul Vogt and Burke Johnson. *Dictionary of Statistics & Methodology: A Nontechnical Guide for the Social Sciences*. SAGE Publications, 4th edition, 2011.

[212] Lev S. Vygotsky. *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press, 1978.

[213] Lev S. Vygotsky. *Thought and language*. MIT Press, Cambridge, MA, 1986.

[214] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[215] Christopher J. C. H. Watkins and Peter Dayan. Q-Learning. *Machine Learning*, 8(3-4):279–292, 1992.

[216] Thomas Weise, Raymond Chiong, and Kē Táng. Evolutionary Optimization: Pitfalls and Booby Traps. *Journal of Computer Science and Technology (JCST)*, 27(5):907–936, 2012. Special Issue

on Evolutionary Computation, edited by Xin Yao and Pietro S. Oliveto.

[217] Shimon Whiteson. Evolutionary Computation for Reinforcement Learning. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*, pages 325–358. Springer, Berlin, Germany, 2012.

[218] Shimon Whiteson and Peter Stone. Evolutionary Function Approximation for Reinforcement Learning. *Journal of Machine Learning Research*, 7:877–917, 2006.

[219] Shimon Whiteson, Brian Tanner, Matthew E. Taylor, and Peter Stone. Generalized Domains for Empirical Evaluations in Reinforcement Learning. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML 2009): Workshop on Evaluation Methods for Machine Learning*, 2009.

[220] Shimon Whiteson, Brian Tanner, Matthew E. Taylor, and Peter Stone. Protecting Against Evaluation Overfitting in Empirical Reinforcement Learning. In *IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, pages 120–127. IEEE, 2011.

[221] Darrell Whitley, Stephen Dominic, Rajarshi Das, and Charles W. Anderson. Genetic Reinforcement Learning for Neurocontrol Problems. In *Genetic Algorithms for Machine Learning*, pages 103–128. Springer US, 1994.

[222] Bernard Widrow and Fred W. Smith. Pattern Recognizing Control Systems. In *Computer and Information Sciences: Collected Papers on Learning, Adaptation and Control in Information Systems*, pages 288–317, Washington, DC, USA, 1964. Spartan Books.

[223] Alexis P. Wieland. Evolving Neural Network Controllers for Unstable Systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 667–673. IEEE, IEEE, 1991.

[224] Marco A. Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam, 1999.

[225] Marco A. Wiering and Jürgen Schmidhuber. Fast Online Q(λ). *Machine Learning*, 33(1):105–115, 1998.

[226] Marco A. Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-Art*. Adaptation, Learning, and Optimization. Springer, 2012.

[227] Eric Wiewiora, Garrison W. Cottrell, and Charles Elkan. Principled Methods for Advising Reinforcement Learning Agents. In

Tom Fawcett and Nina Mishra, editors, *Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003)*, pages 792–799, Washington, DC, USA, 2003. AAAI Press.

[228] Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task Reinforcement Learning: a Hierarchical Bayesian Approach. In Zoubin Ghahramani, editor, *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML 2007)*, volume 227, pages 1015–1022, Corvallis, Oregon, USA, 2007. ACM.

[229] Jay F. Winkeler and B. S. Manjunath. Incremental Evolution in Genetic Programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 403–411, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[230] David J. Wood, Jerome S. Bruner, and Gail Ross. The Role of Tutoring in Problem Solving. *Journal of Child Psychiatry and Psychology*, 17(2):89–100, 1976.

[231] Xin Yao. Evolving Artificial Neural Networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[232] Taku Yoshioka, Shin Ishii, and Minoru Ito. Strategy Acquisition for the Game "Othello" Based on Reinforcement Learning. In Shiro Usui and Takashi Omori, editors, *Proceedings of the Fifth International Conference on Neural Information Processing, ICONIP98*, pages 841–844, Kitakyushu, Japan, 1998. IOA Press.

[233] Wei Zhang and Thomas G. Dietterich. A Reinforcement Learning Approach to Job-Shop Scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95*, IJCAI, pages 1114–1120, Montréal Québec, Canada, 1995. Morgan Kaufmann.

[234] Jean-Christophe Zufferey, Dario Floreano, Matthijs van Leeuwen, and Tancredi Merenda. Evolving Vision-Based Flying Robots. In *Proceedings of the Second International Workshop on Biologically Motivated Computer Vision*, BMCV '02, pages 592–600, London, UK, 2002. Springer-Verlag.