SCALING GENETIC PROGRAMMING TO CHALLENGING REINFORCEMENT TASKS THROUGH EMERGENT MODULARITY

by

Stephen Kelly

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
June 8 2018

Table of Contents

List of Tables

List of Figures

Abstract

Algorithms that learn through environmental interaction and delayed rewards, or reinforcement learning, increasingly face the challenge of scaling to dynamic, high-dimensional environments. Video games model these types of real-world decision-making and control scenarios while being simple enough to implement within experiments. This work demonstrates how emergent modularity and open-ended evolution allow genetic programming (GP) to discover strategies for difficult gaming scenarios while maintaining relatively low model complexity. Two related learning algorithms are considered: Policy Trees and Tangled Program Graphs (TPG).

In the case of Policy Trees, a methodology for transfer learning is proposed which specifically leverages both structural and behavioural modularity in the learner representation. The utility of the approach is empirically evaluated in two challenging task domains: RoboCup Soccer and Ms. Pac-Man. In RoboCup, decision-making policies are first evolved for simple subtasks and then reused within a policy hierarchy in order to learn the more complex task of Half-Field Offense. The same methodology is applied to Ms. Pac-Man, in which case the use of task-agnostic diversity maintenance enables the automatic discovery of suitable sub-policies, removing the need for a prior human-specified task decomposition. In both task domains, the final GP decision-making policies reach state-of-the-art levels of play while being significantly less complex than solutions from temporal difference methods and neuroevolution.

TPG takes a more open-ended approach to modularity, emphasizing the ability to adaptively complexify policies through interaction with the task environment. The challenging Atari video game environment is used to show that this approach builds decision-making policies that broadly match the quality of several deep learning methods while being several orders of magnitude less computationally demanding, both in terms of sample efficiency and model complexity. Finally, the approach is capable of evolving solutions to multiple game titles simultaneously with no additional computational cost. In this case, agent behaviours for an individual game as well as single agents capable of playing up to 5 games emerge from the same evolutionary run.

List of Abbreviations Used

ADF Automatically Defined Function

AVF Adapted Visual Field

CC Cooperative Coevolution

GA Genetic Algorithm

GP Genetic Programming

HRL Hierarchical Reinforcement Learning

MTRL Multi-Task Reinforcement Learning

RL Reinforcement Learning

SBB Symbiotic Bid-Based Genetic Programming

TD Temporal Difference

TPG Tangled Program Graph

Chapter 1

Introduction

1.1 Machine Learning and Artificial Behavioural Agents

Living organisms interact with their environment through an open-ended sequence of observations and actions. The organism observes its surroundings and responds by making decisions and taking actions that affect the world. Most real-world decision-making problems can be characterized as an environmental interaction of this nature. An organism's *behaviour* is defined by how the sequence of observations, actions, and environmental changes plays out over time. Adaptation is critical. That is, the term behaviour only applies to an organism who's sensory/motor response systems have been adapted through environmental interaction [23].

Reinforcement learning (RL) is a common means of applying machine learning to build artificial behavioural agents. RL can be understood as learning how to act in an environment, that is, how to map situations to actions in the pursuit of a pre-defined objective [135]. A solution in RL is represented by an agent that develops a decision-making policy through direct interaction with the task environment. Interactions in RL are typically episodic, beginning with the agent situated in a start state defined by the task environment. From there, the agent observes the environment via sensory inputs, takes an action based on the observation, and receives feedback in the form of a reward signal. The process repeats in a loop until a task end state is encountered or the episode ends for another reason, such as a time constraint. The end state provides the final reward signal that characterizes the quality of the policy, or the degree of success/failure, Figure 1.1. The policy's objective is therefore to select actions that maximize this long-term reward. The potential applications for RL are vast and diverse, from autonomous robotics [78] to interaction design in video games [138] and network security [93]. The breadth of these applications has motivated RL researchers to design frameworks that are general enough to be applied to a variety of environments without extensive parameter tuning.

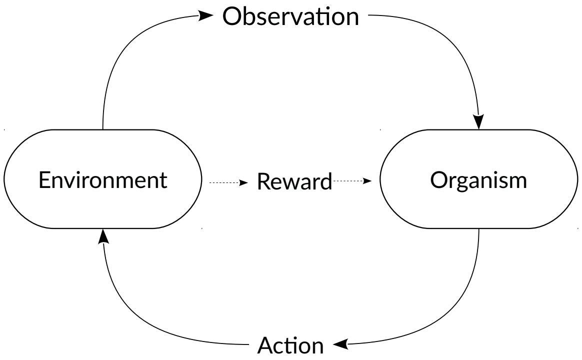


Figure 1.1: Inspired by living organisms, artificial behavioural agents adapt through interaction with their environment. Feedback from the environment (reward signal) is only informative after many interactions. The agent's objective is to maximize its long-term cumulative reward.

1.2 The Scaling Problem

In real-world applications of RL, the agent is likely to observe the environment through a high-dimensional sensory interface (e.g. a video camera). However, scaling to high-dimensional state observations presents a significant challenge for machine learning, and RL in particular. Each individual input from the sensory interface, or *state variable*, may have a large (potentially infinite) number of possible values. Thus, as the number of state variables increases, there is a significant increase in the number of environmental observations required for the agent to gain the breadth of experience required to build a strong policy. This is broadly referred to as the *curse of dimensionality* (Section 1.4 of [16]). The temporal nature of RL introduces additional challenges. In particular, complete information about the environment is not always available from a single observation (i.e the environment is *partial observable*) and delayed rewards are common, requiring the agent to make thousands of decisions before receiving enough feedback to assess the quality of its behaviour [63]. If the agent could simultaneously learn a decision-making policy *and* determine which state variables to use versus what is irrelevant to the task at hand, it could potentially mitigate the computational cost of processing a large number of training samples. This is the nature of the RL algorithm proposed in this thesis.

Video Games

Video games provide an interesting test domain for scalable RL. In particular, they cover a diverse range of dynamic task environments that are designed to be challenging for humans, all through a common high-dimensional visual interface, or the game screen, e.g. [14]. Developing behavioural agents, or non-player characters (NPC), for video games through reinforcement learning is of broad significance to game AI in general [163], with specific importance to three classes of end user: (1) From the perspective of the *game player*, highly-skilled, believable agent behaviours represent interesting game content and thus enhance the gaming experience. A believable agent behaviour implies that the agent's decision-making should be plausible. It is increasingly unacceptable to rely on cheats or global information. Moreover, agents need to be 'proficient' in order to be believable. The more proficient/believable a behaviour, the more immersive a gaming experience potentially is; (2) From the perspective of the *game designer*, NPCs that exhibit human-level competence can be used to simulate playthroughs in newly created game environments, automating aspects of quality control in game design. Furthermore, automated testing of game content is essential to procedural content generation (i.e. game design by AI); (3) From the perspective of the *AI researcher*, games represent an interesting benchmarking environment characterized by high-dimensional spatial-temporal information. Moreover, the environments are often non-stationary and subject to partial observability. As such, games represent a particularly rich combination of challenges for RL.

Artificial Behavioural Agents and Generalization

With regard to constructing behavioural agents for video games through machine learning, the concept of generalization has been studied primarily from three related but unique perspectives:

- *Domain-Independent AI* is concerned with developing learning algorithms that can be applied to a variety of task domains with minimal prior knowledge and no task-dependent parameter tuning. To date, this has been the focus of studies in the Atari 2600 environment, where several approaches have demonstrated an ability to build policies for multiple game titles using the same learning framework [100, 14, 49, 152, 104]. In these works, a game-specific policy is developed from scratch for each game title. However, task transfer has also been employed in the Atari task [22], in which case the agent reuses experience gained in one or more *source* game titles to improve learning under a single *target* title. The Term 'General Video Game Playing' is often used when describing Domain-Independent AI. However, there is an important distinction between these terms which we acknowledge below.

- *General Video Game Playing* is concerned with building agents that are capable of playing multiple games which they have *never seen before* [112, 86]. That is, no off-line experience / training with a specific game environment takes place prior to evaluation on that game. Thus, while learning is possible during gameplay, a prior level of general game-playing competence is required.

- *Multi-Task Learning* is concerned with discovering agent behaviours for multiple tasks simultaneously [142, 109], resulting in multiple task-specific policies and/or a single policy that is capable of managing multiple tasks (e.g. playing multiple game titles) at a high skill level. Multi-task learning in sequential decision-making environments is currently seen as an important step toward developing artificial general intelligence, i.e. algorithms capable of demonstrating human levels of intelligent behaviour [62, 41]. Indeed, a game-based formulation of the Turing test has been proposed in which the play of artificial agents are compared to that of humans [54].

This thesis specifically targets Domain-Independent AI and Multi-Task Learning in a variety of challenging game environments.

1.3 On the Significance of Ecology and Bottom-Up, Behaviour-Based Development

The concept of behaviour is derived from observing natural systems. With regard to modeling behaviour in artificial agents, the central focus of this thesis, it is not surprising that biologically-inspired mechanisms have proven useful. Specifically, artificial neural networks and genetic algorithms have been central to the study of artificial behavioural agents, for example, from the work of Valentino Braitenberg

[19] to the current popularity of Deep Learning [100] and Neuro-evolution [127]. The work herein adds to the latter contemporary approaches to modeling behavioural agents, and makes a novel contribution through revisiting and implementing some fundamental ideas in artificial behavioural systems:

- The *ecological approach* to understanding perception and behaviour [42] emphasizes the notion of environmental situatedness, or the idea that an agent/organism observes and acts within its environment, and relates to the environment on a specific spatio-temporal scale rather than through a complete, abstracted description of the world that might be easily understood by an external observer [27]. A core tenant of the ecological approach is that action and perception are fundamentally linked [27], which implies that a system of perception, as a fundamental component of behaviour, is also an adapted property.

- Early work in behavioural robotics by Rodney Brooks emphasized the importance of *behavioural decomposition*. Brooks proposed the subsumption architecture [23], an artificial behavioural system that is incrementally constructed from multiple interconnected modules, each representing a functionally independent *learned* behaviour. In contrast to a system designed through functional decomposition (i.e. distinct system components for vision, locomotion, etc.), behavioural modularity implies that the complexity of the artificial agent is not derived from the complexity of the world model, but from the interaction among multiple interconnected behavioural modules within a situated agent, itself in continuous interaction with the outside world.

Taken together, these points suggest that an artificial behavioural agent is essentially a *Complex System*, or a "collection of diverse, connected, interdependent entities whose behaviour is determined by rules, which may adapt, but need not. The interconnections of these entities often produce phenomenon [*sic*] that are more than the parts. These phenomenon [*sic*] are called emergent."[108]

This work proposes a genetic programming (GP) framework that emphasizes emergent behavioural modularity to addresses challenges in scaling RL to real-world tasks while maintaining minimal model complexity.

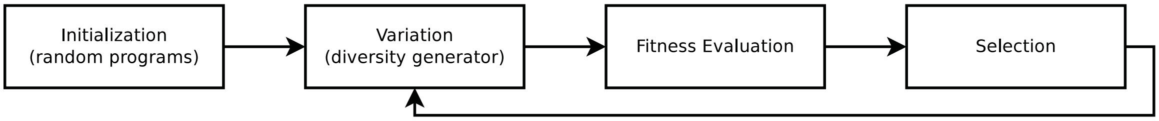


Figure 1.2: Basic workflow of a genetic algorithm. Genetic *programming* implies that the individuals being evolved are computer programs.

1.4 Genetic Algorithms

As a biologically-inspired approach to machine learning, a Genetic Algorithm (GA) generalizes the basic principles of biological evolution in order to discover incrementally more sophisticated prediction models. In this work, adopting a GA as the underlying adaptive mechanism is partially motivated by a general sense of wonder at the elegance and sophistication of organisms that have emerged from biological evolution. Throughout this thesis, the term *organism* is not only used for biological entities, but also to describe artificial agents in cases where the role of self-organization should be emphasized. For example, the nature of GAs allows the development process to begin with the simplest agent representations and then incrementally complexify based on interaction with the environment, simultaneously learning an effective decision-making policy and adapting the complexity of the model based on the requirements of the task.

In a basic generational GA, a population of organisms are randomly initialized and then iteratively developed over a sequence of generations. In each generation, the organisms are evaluated on a task and assigned a fitness score, the worst performing individuals are deleted, and the remaining organisms are sampled and modified to form the next generation. As in Darwinian evolution, the driving force of development in GAs is the selection-variation loop, Figure 1.2.

Banzhaf [9] discusses how the selection phase in a GA implies *downward causation* in the development of organisms. That is, the high-level interaction between an organism and its environment determines how lower-level attributes of the organism (e.g. code, weights, or other representation-specific components) evolve[1]. GAs

[1] For completeness, it is important to note that *bottom-up* causation, or the mapping from low-level attributes of the organism (e.g. code, weights) to high level attributes such as behaviour, plays an equally important role. However, such mappings are deterministic in this thesis. That is, assuming a consistent start state and a deterministic environment, an organism will always exhibit the same behaviour. As such, less attention is paid to bottom-up causation in this work.

present a practical starting point for the open-ended development of complex structures, where both the nature of an organism's behaviour and its structural complexity are emergent properties adapted through the combination of environmental interaction and the selection-variation loop. The concept of *emergence* is applied here to indicate that neither properties are explicitly selected for by the GA. That is, while the GA is designed to generate organisms with increasingly higher fitness, nothing is specified a priori regarding *how* an organism might achieve a high fitness score or the level of complexity necessary. A versatile representation for organisms is critical for emergence in this respect. This thesis proposes a novel representation which emphasizes the role of emergent modularity [107] (and by extension hierarchy) in developing strong yet efficient agent behaviours.

1.4.1 Genetic Programming

Genetic programming (GP) is the special case of a genetic algorithm where the individual members of the evolving population are programs [79]. A program's basic functionality is to solve a problem by processing input and predicting the best output to maximize an objective function. Real-world inputs and objective functions are often noisy and a program is typically required to generalize its operation to input data not seen during development/training. Under such uncertainty, a program's input-output mapping is an approximation, or prediction model [20]. In GP, the mechanism for adaptation takes the form of a search over the space of possible models. Thus, GP is a form of *model building*. This is distinct from other types of machine learning which complement a pre-existing model or objective function by searching for an optimal setting for the model's variables, i.e. *parameter optimization*. By definition, models contain more structure than points in multi-dimensional space, and as a result it has been suggested that model building might be less prone to learning-and-forgetting [88]. Multi-Task Learning represents a scenario in which multiple, potentially unrelated tasks are learned by the same model (See Section 1.2). Thus, the model requires a mechanism to avoid forgetting one task while learning another [77]. The empirical study in Section 9.6.6 demonstrates how the GP representation proposed in this work avoids forgetting by building multiple task-specific modules into the structure of the learned model.

More broadly speaking, there are several advantages in adopting GP. In particular, programs are capable of representing arbitrarily complex and non-linear functions, and minimal prior domain knowledge is required when selecting the programs' function set. Moreover, GP represents an embedded approach to state variable selection. That is, the system is capable of determining which state variables to use as part of the learning process.

The principal disadvantage of GP is that it is often considered computationally expensive, primarily due to the relatively high cost of evaluating the objective function. There is a significant cost in maintaining a population of candidate solutions, many (if not all) of which must be evaluated in the task domain prior to every application of selection and variation operators (See Figure 1.2) (the cost of evaluations is explained in more detail when discussing cost function in the next section). GP can also be prone to code bloat, where the complexity of programs increases without improving the quality of the policies they define. That is, the complexity of organisms becomes disengaged from the requirements of the task environment [6, 124]. Efficient policy representation in GP is a key factor in addressing the scaling problem, and adaptively scaling the complexity of organisms in concert with the requirements of the environment is one of the primary focuses of this thesis.

As in the design of any machine learning algorithm, numerous decisions influence the overall performance and complexity of a GP system. These design questions can be categorized by three fundamental considerations: representation, cost function, and credit assignment. While such categories are not independent and design decisions often involve trade-offs, the following subsections provide an overview of each category and give a brief justification for the decisions made in this work.

1.4.2 Representation

Representation defines the structure of organisms, including basic elements and the relations or rules that govern how elements may interact and/or be organized into complex structures. In the context of this work, individual solutions are composed of multiple programs working together. As such, representation includes a specification for the type of programs that will be evolved, linear register machines in this case [20], and a specification for how multiple programs combine to form an overall solution, all of which is captured by an organism's low-level encoding, or *genotype*. Given the set of possible solutions defined by a particular representation, a search process is tasked with finding the best one(s) relative to the problem at hand. Thus, a good representation should be general enough to define complete solutions for a wide variety of problems without unnecessary complexity *and* support an efficient search process. Section 1.5 provides overview of the specific representation considered in this thesis, with Chapters 4, 5, and 8 providing complete details.

1.4.3 Cost Function

The cost function represents a measure of the quality of an organism's decision-making policy relative to the task objective(s). As such, the cost function quantifies aspects of an organism's *phenotype*, or the observable characteristics of the organism as it interacts with the task environment. In a GA, the cost function is used to assign organisms with a fitness value. Individuals with higher fitness produce more offspring, thus directing the search over successive generations to more desirable regions of the search space. In RL tasks, raw measure of a policy's quality is the reward signal received from the environment at the end of an episode. Unfortunately, each episode of environmental interaction may involve thousands of decision steps, and thus obtaining this feedback can be computationally expensive. Furthermore, noise in the environment (discussed with respect to specific tasks in Chapters 3 and 7) implies that each policy must be evaluated in multiple episodes, with the cost function derived from the combined outcome. Finally, the raw episode outcome received from the environment may be augmented with heuristics in order to direct the search. A common example of this in GAs is diversity regularization, a method of maintaining population diversity by defining each policy's fitness as a combination of raw environmental outcome and some measure of uniqueness relative to the rest of the current population, discussed in detail in Section 5.4.1.

1.4.4 Credit Assignment

Credit assignment is the mechanism used to modify candidate solutions relative to information obtained through the cost function. In sequential decision-making problems, the task environment may provide the agent with a reward in response to each action taken (each time-step or interaction step throughout the episode). However, it is often difficult to determine which specific decision(s) led to ultimate success or failure. For example, even actions with a neutral or negative step-wise reward may ultimately contribute to a successful outcome. This is known as the temporal credit assignment problem [136, 55]. The problem is addressed differently by methods that perform a learning update relative to each decision and the immediate reward within the temporal sequence, or *ontogenetic* learning (e.g Temporal Difference learning, TD(λ) [136]), and cases such as GP, in which an organism's decision-making policy is evaluated as whole based on the final episode outcome only, or *phylogenetic* learning. In effect, decision-level credit in GP is applied implicitly, since policies that make better decisions will receive higher fitness and produce more offspring, thus evolution directs the search in favour of policies that make individual decisions that contribute to a positive overall outcome. As such, the issue of temporal credit assignment is effectively managed through the downward causation implicit in GP (See Section 1.4). In the context of model building with GP, each learning update effectively creates a new model (e.g. by selection and variation operators in the GA), and thus the search process is performed over the space of possible models (decision-making policies) within a particular representation. Under RL tasks this approach is known as *policy search*.

The relative merits of ontogenetic and phylogenetic learning for sequential decision-making tasks has been the subject of debate [11], and which method is superior for a particular problem remains an open question, with arguments supporting the advantages of both phylogenetic [102] and ontogenetic [136, 135] methods. While no argument is made one way or the other here, this work can be seen as an empirical example of the strengths of phylogenetic, GP-based RL.

1.5 Approach

This thesis is concerned with two related but unique representations for developing hierarchical decision-making policies through evolutionary policy search: *Policy Trees* and *Tangled Program Graphs*. Key research contributions are demonstrated through two extensive empirical evaluations, highlighting the relative merits of each representation in relation to current state-of-the-art RL algorithms.

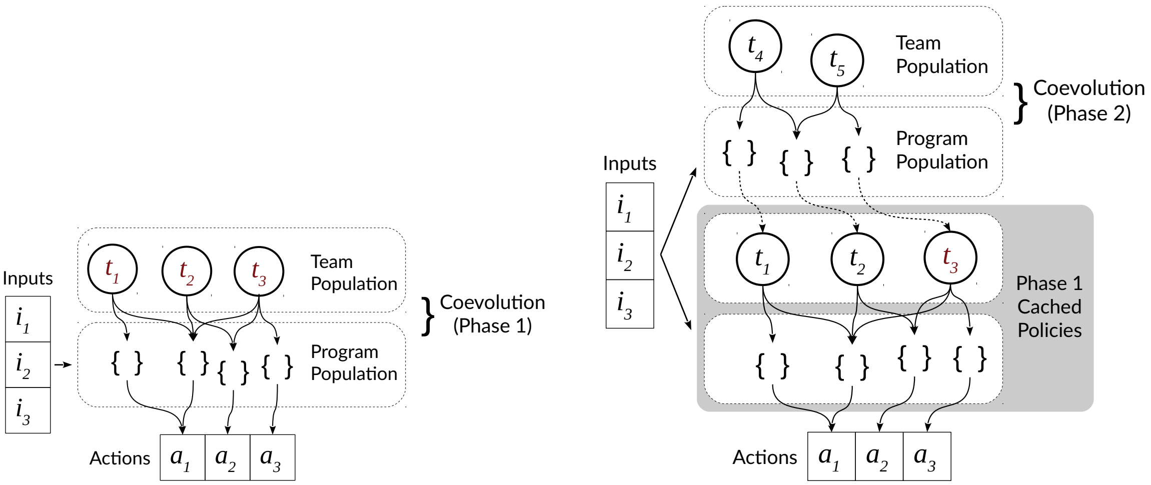
1.5.1 Teams of Programs and Policy Trees

The representations considered in this work are extensions of the well-established Symbiotic Bid-Based (SBB) algorithm for evolving teams of programs [91]. SBB has been shown to build strong policies for a variety of reinforcement learning tasks [35, 75, 92, 70, 71], owing primarily to two key innovations:

1. Solutions are represented by a team of programs, in which each program defines a context for deploying a single action. In this context, programs can be understood as value functions for state/action pairs. That is, for each decision made by a team, each team member (program) processes sensor input (all or part of the current environmental observation represented as a vector of real values) and produces a single real-valued output, or *bid*. The team then deploys the action associated with the highest-bidding program. The number and complement of programs per team, as well as the bidding behaviour of each program, are evolved properties, Figure 1.3(a);

2. A single team of programs represents the smallest stand-alone decision-making entity. More complex, hierarchical decision-making agents may be constructed over two independent phases of evolution, where the first phase produces a library of diverse, specialist teams and the second phase attempts to build more general policies by reusing the library. The method of hierarchically organizing teams through code reuse is an example of what Watson and Pollack call *compositional evolution*[158], or the evolution of complex structures that combine multiple subcomponents which were previously adapted independently as stand-alone behaviours. In the case of SBB, symbiotic coevolution provides the underlying mechanism for building teams of programs *and* hierarchically organizing multiple teams into a *policy tree*, Figure 1.3(b).

The explicitly modular nature of SBB is partly motivated by the intuition that (automatic) problem decomposition is an important learning skill for artificial agents, just as it is for humans. Each program learns a unique mapping from state to value, or bid. Since the team will consider all its members (programs) and select the action associated with the single highest bidder, the programs within a team collectively represent a form of lateral problem decomposition. Conversely, policy trees, constructed



(a) (b)

Figure 1.3: Overview of Policy Trees. Teams and programs are stored in separate populations and coevolved. A single team, which is simply a variable length list of pointers to members of the program population, represents the smallest standalone decision-making entity in SBB (a). Multiple team/program population pairs can be stacked and developed bottom-up over two phases of evolution (b) to form a policy tree. In the case of hierarchically stacked populations, or policy trees, programs above Phase 1 learn state/action mappings in which the 'action' refers to a previously evolved (and archived) team. In other words, programs in Phase 2 reuse code from a previous evolutionary run, learning new contexts in which to deploy a cached team of programs.

from the bottom up over independent phases of evolution, represent a form of vertical problem decomposition.

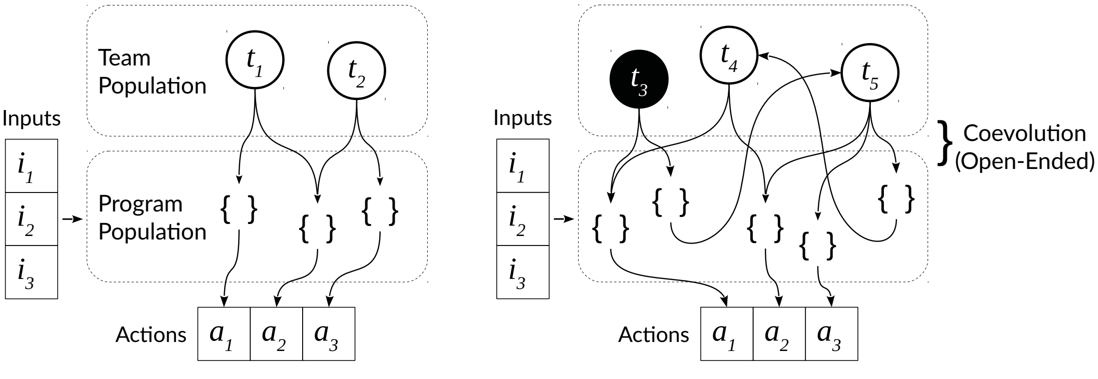
1.5.2 Research contribution 1: A Methodology for Transfer Learning Using Policy Trees

The first research contribution of this thesis is to demonstrate how vertical decomposition in SBB Policy Trees can be extended to support task transfer, discovering strategies for difficult gaming scenarios while maintaining relatively low model complexity. Critical factors in the proposed approach are illustrated through an in-depth study in two challenging task domains: RoboCup Soccer and Ms. Pac-Man. In RoboCup, it is shown that policies initially evolved for simple subtasks (Phase 1 of evolution, Figure 1.3(b)) can be reused, with no additional training or transfer function, in order to improve learning in the complex Half Field Offense (HFO) task (Phase 2 of evolution, Figure 1.3(b)). It is then shown how the same approach to code reuse can be applied directly in Ms. Pac-Man. In the latter case, the use of task-agnostic diversity maintenance removes the need to explicitly identify suitable subtasks a priori. The resulting GP policies achieve state-of-the-art levels of play in HFO and surpass scores previously reported in the Ms. Pac-Man literature, while employing less domain knowledge during training. Moreover, it is shown that modularity plays an important role in the development of *efficient* decision-making policies, which are shown to be significantly less complex than state-of-the-art solutions in both domains. In addition, special attention is paid to a pair of task-agnostic diversity maintenance techniques and their importance to the development of strong policies is empirically demonstrated[2]. This work is published in [74], ©2017 IEEE. Excerpts are used in this thesis by permission.

1.5.3 Research contribution 2: Addressing the Scaling Problem in GP Through Emergent Modularity

The principal disadvantage of the Policy Tree approach stems from the fact that each 'path' through the tree (of teams) is of equal depth. Thus, a team at the root describes a policy through a path defined in terms of teams at *all* prior levels of the tree. Unfortunately, as the tree depth increases it becomes increasingly difficult to disambiguate the relation between state and action, as new teams have to operate 'through' all previous policy levels. If teams could instead be inter-related through a graph structure, the potential to retain clarity between state and action could be retained. Furthermore, while effective, the approach to constructing policy trees described above requires that some critical decisions be made a priori: (1) How many phases of evolution, or levels in the hierarchy, might be useful for a particular problem? This also implies defining an appropriate diversity maintenance scheme and/or manually decomposing the task into useful source tasks for phase 1 of evolution. This assumes that the relevant diversity for a given phase is self evident. However, as the capacity of a policy improves, the type of diversity and useful task decomposition might radically change as new parts of the environment are encountered; (2) What computational budget (generation or evaluation limit) should be granted to each phase of evolution? While such decisions could be partially automated through

[2] Preliminary work on the subject of diversity maintenance can be found in [70, 69, 71]



(a) Initial policies

(b) Evolved policies

Figure 1.4: Overview of Tangled Program Graphs. Decision-making in each time step (frame) begins at the root team and follows the edge with the winning program bid (output) until an atomic action is reached. The initial population contains only single-team polices (a). Multi-team graphs emerge as evolution progresses (b).

the use of heuristics and competitive coevolution [122], an alternative solution might be to use emergent modularity [107] to adaptively complexify policies through interaction with the task environment. This idea motivates the second hierarchical representation proposed and investigated in this thesis: Emergent Tangled Program Graphs (TPG).

In TPG, evolution begins with a population of simple teams, Figure 1.4(a), which are then further developed by adding, removing, and modifying individual programs. TPG extends the SBB approach to team GP to enable emergent *behavioural* modularity from a single cycle of evolution by adaptively recombining multiple teams into variably deep/wide directed graph structures, or *policy graphs*, Figure 1.4(b). The behaviour of each program, complement of programs per team, complement of teams per graph, and the connectivity within each graph are all emergent properties of an open-ended evolutionary process. The benefits of this approach are twofold:

1. A single policy graph may eventually evolve to include hundreds of teams, where each represents a simple, specialized behaviour (Figure 1.4(b)). However, mapping a state observation to an action requires traversing only *one* path through the graph from root (team) to leaf (action). Thus, the representation is capable of compartmentalizing many behaviours and recalling only those relevant to the current environmental conditions. This allows TPG to scale to complex, high-dimensional task environments while maintaining a relatively low computational

cost per decision.

2. The programs in each team will collectively index a small, unique subset of the state space. As multi-team policy graphs emerge, only specific regions of the state space that are important for decision-making will be indexed by the graph as a whole. Thus, emergent modularity allows the policy to simultaneously decompose the task spatially *and* behaviorally, detecting important regions of the state space and optimizing the decisions made in different regions. This minimizes the requirement for *a priori* crafting task-specific features, and lets TGP perform both feature construction and policy discovery simultaneously.

The RoboCup and Ms. Pac-Man tasks employed for the study of Policy Trees both assume relatively low-dimensional, human-engineered inputs (e.g. distance and angle measurements) and both tasks are manually decomposable into simpler subtasks, allowing for intuitive formulation of task transfer. The second large empirical study in this thesis investigates the utility of TPG under the challenging problem of scaling RL to dynamic, high-dimensional, and partially observable environments which are not easily decomposable a priori. Significant attention is being paid to frameworks from deep learning, which scale to high-dimensional data by decomposing the task through multi-layered neural networks. While effective, the representation is complex and computationally demanding. In the challenging Atari video game environment, TPG is compared with several deep reinforcement learning frameworks as well as more traditional reinforcement learning frameworks based on a priori engineered inputs. Results indicate that the proposed approach matches the quality of deep learning while being a minimum of three orders of magnitude simpler with respect to model complexity. This results in real-time operation of the champion RL agent without recourse to specialized hardware support. Moreover, the approach is capable of evolving solutions to multiple game titles simultaneously (i.e. Multi-Task Learning) with no additional computational cost. In this case, agent behaviours for an individual game as well as single agents capable of playing *up to 5* games emerge from the same evolutionary run [72, 73].

1.6 Chapter by Chapter Thesis Outline

This thesis is organized as follows:

Chapter 2 presents the methodological foundations supporting the evolution of Policy Trees and Tangled Program Graphs. Specific topics reviewed include Hierarchical Reinforcement Learning (HRL), Cooperative Coevolution and Modularity, Transfer Learning, and Diversity Maintenance.

Chapter 3 introduces the RoboCup and Ms. Pac-Man task domains and explains why they are interesting benchmark environments for scalable policy search and transfer learning in particular.

Chapter 4 describes the Symbiotic Bid-Based (SBB) algorithm, detailing the representation for teams of programs. SBB is the starting point for both Policy Trees (Chapter 5) and Tangled Program Graphs (Chapter 8).

Chapter 5 describes the algorithm for code reuse through Policy Trees. Emphasis is placed on how the architecture is extended to support transfer learning and the approach taken for defining task-agnostic diversity measures.

Chapter 6 documents an empirical evaluation of transfer learning using Policy Trees in the RoboCup and Ms. Pac-Man task environments. A direct comparison is made with Sara (RoboCup) and Neuro-evolution (Ms. Pac-Man) and Policy Trees are shown to produce highly competitive policies while being much more efficient to deploy post-training.

Chapter 7 describes the Arcade Learning Environment (ALE), a suite of Atari 2600 video games equipped with an interface for benchmarking high-dimensional *visual* reinforcement learning agents. This chapter also reviews recent approaches to developing both single-task and multi-task Atari game playing agents through deep learning, neuro-evolution, and temporal difference learning methods.

Chapter 8 describes the Emergent Tangled Program Graph (TPG) algorithm, detailing how SBB is extended to support open-ended evolution of *policy graphs*, and why this development makes sense in the context of high-dimensional, visual reinforcement learning environments.

Chapter 9 documents an empirical evaluation of TPG in the Arcade Learning Environment, highlighting TPG's capability to build both single-task and multi-task policies that are competitive with results from a variety of deep learning approaches.

More importantly, TPG policies are shown to be significantly more efficient to train and deploy post-training.

Chapter 10 concludes the thesis, summarizes the research contributions, and looks toward future work.

Chapter 2

Foundations

This chapter presents a review of the fundamental concepts and building blocks supporting the algorithms considered in this thesis. It is by no means an exhaustive account of research surrounding any particular concept, but rather an introductory overview of the fundamentals, with references to research that specifically informed or inspired the work herein.

2.1 Modularity and Hierarchical Models

In this thesis, the term modularity is primarily used to describe a system that is incrementally constructed from multiple subsystems which were initially developed independently, or more specifically *compositional evolution* [158]. An overarching goal in this research is to propose how two critical properties of such a system can be automated: 1) The identification of stable building blocks, or subsystems; and 2) Establishing the nature of the interaction among subsystems within a hierarchical decision-making policy, or *module interdependence*.

Relative to the first property to be automated, i.e. discovery of stable building blocks, Herbert Simon [119] suggested that the presence of stable intermediate structures speeds up evolution by providing building blocks from which increasingly complex hierarchies may be constructed. Put simply, Simon points out that if a complex system is built from structurally modular building blocks, its development is less likely to require a restart from scratch should an error be introduced during construction (See Simon's famous Watchmaker's Parable for an illustrative example of this concept). In other words, modularity helps promote stability in an evolving organism, preventing a particular genome from being a "House of Cards" [76] in which a single variation might bring it tumbling down. Ultimately, Simon's suggestion is that modular systems are more *evolvable*, that is, more capable of continuously discovering new organisms with higher fitness than their parents. This theory has been investigated widely among evolutionary biologists [162, 106]. Indeed, the concept has motivated a breadth of cross-disciplinary research between evolutionary biology and computer science, with the potential to address fundamental problems in both disciplines [156].

As for the second property to be automated, or the nature of interaction among subsystems, Watson *et al.* [158] emphasize that structural modularity (i.e. structural complexity encapsulated such that dependencies between subsystems are weaker than dependencies within subsystems) does not imply *independence* of subsystems. Specifically, functional interdependence among subsystems is critical for hierarchies in which all levels of organization are meaningful. In the simplest sense, if we assume that an organism's incremental acquisition of modules leads to an incremental improvement in fitness, then a relationship between modularity and evolvability is obvious. However, Watson points out that simply accumulating multiple building blocks into an aggregate system does not capture the full potential of modularity. Although he does not use the term, he ultimately expresses that module interdependence is essential for *emergence* because without interdependence, a hierarchy of subsystems is nothing more than the sum of its parts (See the analysis of Herbert Simon's lock picking example [158] for a clear illustration of the importance of module interdependence). Watson proposes a formal definition for module interdependence, and argues that systems with strong module interdependence are evolvable under certain conditions, namely composition evolution [29].

Regarding the specific approach to GP teaming employed within this work, the above characterization of modularity implies that a single team of programs represents the simplest possible subsystem, or module. That is, a team represents the simplest stand-alone entity capable of being developed independently[1]. From this starting point, this thesis investigates two approaches to building artificial behavioural agents through compositional evolution, or the automated organization of multiple teams into increasingly complex systems. The first approach, Policy Trees, emphasizes support for scaling to complex tasks through transfer learning (Section 2.5). In this context, diversity maintenance (Section 2.3) helps to promote module interdependence by ensuring subsystems are unique *partial* solutions. The second approach,

[1] This perspective does not discount the modularity inherent *within* a team of programs, it simply defines the module boundaries at a higher level of abstraction, i.e. stand-alone decision-making entities.

Tangled Program Graphs, introduces a specialized mutation operator and additional representational rules which allow for a more open-ended form of compositional evolution (Section 2.3.4). In both cases, task domains are assumed in which the general goal is to identify an agent capable of operating under RL and Multi-Task Learning in high-dimensional environments.

2.2 Modular Genetic Programming

Modular architectures are a recurring theme in GP, primarily motivated by the observation that many problems can be solved more efficiently by decomposing them into sub-problems which may be solved independently, and then assembling the solutions for sub-problems hierarchically to form an overall solution. Early approaches, for example Koza's Automatically Defined Functions (ADFs) [80], explored various methods of supporting automatic encapsulation of code in subroutines which could then be reused within the evolving programs. Initially, the structure of subroutines (e.g. the number and type of arguments accepted) were defined a priori and could not change during evolution. Furthermore, the appropriate number of subroutines for a particular problem was guessed a priori and specific to a single evolving program, that is, a single program could reuse its own subroutines but there was no sharing of code between programs unless crossover at the root of a subroutine's tree happened to occur. Furthermore, ADFs force evolution to begin from the most complex structures, i.e. initial individuals have a full complement of ADFs. This biases the nature of emergent and self-organizing properties. Koza himself documented the issue of ADFs making it more difficult to find solutions to tasks than without (Chapter 5 in [80]). Indeed, ADFs are no silver bullet, for example, including ADFs for the relatively simple truck reversal task completely failed to identify solutions (Appendix A in [68]). Extensive prior input with respect to solution modularity is problematic because it constrains evolution to a potentially erroneous division of labour [107]. As such, subsequent methodologies aimed to minimize the amount of prior knowledge required to define the modularity appropriate for a particular task. Less constrained approaches were proposed by Koza [81] and others, for example Coevolving Functions in GP [2], Adaptive Representations through Learning [115], as well as Tag-Based Modules [125] all found ways of leveraging modularity to speed up evolution and/or scale GP to more complex tasks. In proposing their Module Acquisition framework [7], Angeline and Pollack articulate a perspective on modularity to which this thesis subscribes, specifically, they emphasize that the complexity and "modularizations" of the representation can emerge through direct interaction between the organism (solution) and its environment (problem).

2.2.1 Teaming and Cooperative Coevolution

Early modular representations for GP focused primarily on evolving single-program solutions. The principal benefits of modularity were in: 1) encapsulating and protecting partial solutions during evolution and; 2) supporting code reuse (i.e. sharing of common knowledge). By contrast to the single-program approach, solutions in this thesis are composed of multiple cooperating programs, i.e. teams of programs [20]. This implies that solutions (teams) are evolved from multiple interacting, coadapted subcomponents (programs), i.e cooperative coevolution (CC) [114]. CC maintains the benefits of modularity already identified, while also directly supporting *automatic* problem decomposition, i.e. the automated discovery of building blocks. In this work, the approach to team development takes inspiration from biological symbiosis [52] in order to explicitly balance functional and structural modularity within evolving organisms. The remainder of this section elaborates on this perspective of modularity in CC, beginning with a brief motivation for adopting a symbiotic cooperative coevolutionary model.

2.2.2 On the Utility of Symbiosis

Early formulations for CC made use of knowledge regarding the number of subcomponents required to form a solution, with each subcomponent being associated with a separate population. For example, given a prior specification for a neural network topology, then different populations might be associated with each weight in the network [44, 82]. A heuristic is now necessary for choosing representatives from each population, potentially differentiating between group fitness (e.g., fitness of the resulting neural network) versus fitness of individual subcomponents. In the context of GP teaming, separate populations could be associated with each program [149], assuming that knowledge is available for defining how many programs should participate within a team. However, the issue of selection heuristics again appears, where this can be performed at the group or subcomponent (program) level. Indeed, alternating between the two appears to provide robust teams under multi-class classification tasks [149].

Symbiosis represents the group and subcomponent using independent populations (host and symbiont respectively). Two works in particular have used such a framework to support CC: 1) SANE (Symbiotic, Adaptive Neuro-Evolution), an architecture for evolving weights (symbiont) and neural networks (host) [101], and; 2) SBB, where programs are symbionts and teams of programs are hosts [90]. In both cases the host population is where a complete organism is expressed (and where fitness evaluation takes place), but in terms of pointers to members of the symbiont population. In effect a combinatorial search is being performed by the host population for 'good' symbiont combinations. If host individuals assume a variable length representation (as in [90]), then the number of subcomponents is also evolved. Thus, broadly speaking, the symbiotic approach to CC automates aspects of building collective decision-making entities that may previously have been fixed a priori. The composition of a group behaviour is now an entirely evolved property.

2.2.3 Functional Modularity

In the (symbiotic) team GP employed in this thesis, each program defines the context for *one* atomic action and an inter-program bidding mechanism resolves which action to associate with a given observation. Thus, in order to form a complete solution, a team must identify multiple programs which collectively solve the problem by each learning the context (i.e. a mapping from environmental observation to scalar bid value) for their respective action. This amounts to a forced division of labour in which each program serves a unique function within the group behaviour. In other words, some degree of *functional* modularity/specialization is explicitly required in order for a team to represent a complete solution and achieve a high fitness score. As such, some amount of problem decomposition is guaranteed and selection can now identify effective group behaviours through fitness expressed at the team level only[2]. Such a mechanism conforms to the idea that organisms must be evaluated as a whole, rather than independent modules, in order to evolve cooperation [149, 91, 161]. However, evolving cooperative behaviours is not *that* easy. Balancing the need to develop specialization (a property of a subcomponent) *and* cooperation (a property of a collection of subcomponents) is where diversity maintenance and neutrality also potentially have parts to play (Sections 2.3 and 5.4).

2.2.4 Structural modularity

In the team GP employed here, a team is simply a collection of pointers to its member programs. As such, the structural complexity within a program's code is more tightly coupled than the structural complexity linking programs within the team, or *structural* modularity. This is significant because the impact of both team and program variation operators is now isolated to particular structural modules, resulting in less likelihood of disruption to all the functional properties of a policy during variation [156].

In short, functional modularity supports automatic problem decomposition, making it possible for credit assignment to identify useful 'modules' and associate them with specific contexts, i.e. the interaction between specialization and modularity [39]. Structural modularity implies that partial solutions can be automatically encapsulated and protected during evolution, as long as they prove useful within a group behaviour.

2.3 Diversity

All evolutionary computation frameworks include some form of diversity generating mechanism to ensure the continued production of novel organisms (Figure 1.2). In the simplest case, this takes the form of a mutation operator introducing errors/variation into the reproduction of organisms such that children are slightly different than their parent(s). However, additional effort is often required in order to ensure that mutation (or other diversity generating operators) results in new organisms that are

[2] It is worth noting that attempts have also been made to define fitness at the program as well as the team level [149, 161], requiring task-specific definitions for fitness at both the individual and group levels.

measurably different (in a phenotypic and / or genotypic sense) form what exists in the current population. In particular, explicit maintenance of a diverse solution population is important to ensure a thorough search of the solution space. Indeed, diversity maintenance is a well known factor in preventing GP from getting stuck in local optima, i.e. avoiding premature convergence [113, 24]. Furthermore, if a policy will ultimately be represented by a group behaviour, then diversity maintenance might be critical in developing individual group members that complement each other in a group setting, that is, producing individuals that succeed / fail in different ways [57, 24]. This section identifies three generic approaches to diversity maintenance, any of which could potentially be used in combination: competitive coevolution, regularization, and multiple populations.

2.3.1 Competitive coevolution

Competitive coevolution attempts to search for training cases (e.g. game configurations) that in some way 'discriminate' between the performance of policies without resulting in disengagement [59, 25]. Naturally, a balance needs to be struck between rewarding generalists (those policies that solve the 'easy' training cases) versus specialists (those that solve 'difficult' training cases). Indeed, depending on the representation (for policies), it might not be possible to resolve the two into a single policy [118]. Unfortunately, in complex tasks individual training cases are often much less informative, for example, due to noise in sensors and actuators or other forms of stochasticity in the environment. Thus, two policies with similar or identical strategies may achieve different levels of success, even if they start from the same initial environmental configuration.

2.3.2 Regularization

Diversity maintenance through regularization implies that the fitness function incorporates discounting for properties other than goal directed task performance. Indeed, at one extreme is novelty *as* the objective [85, 28, 34]. In the development of Policy Trees (Chapter 5), two task-agnostic diversity mechanisms are assumed when evolving solutions to subtasks. These are motivated in part by the intuition that both structural diversity *and* behavioural diversity are likely to promote specialization. Indeed, it has been previously suggested that GP applied to complex problems is likely to benefit from multiple diversity mechanisms [24, 34]. The specific method adopted in this work for maintaining *structural* diversity is inspired by the approach taken in [97] for promoting diversity under a neuro-evolutionary setting. In addition, promoting *behavioural* diversity, or diversity in the observable characteristics of policies as they interact with the environment, has been shown to significantly improve development in evolutionary robotics [103]. Moreover, the approach taken to combining multiple diversity measures also has an impact on the development of modularity. For example, instead of all forms of regularization being present all the time, switching between different objectives can potentially be beneficial [67, 110].

2.3.3 Multi-Population Frameworks

Finally, in the case of Island formulations for multi-population frameworks, the underlying motivation is to provide programs with 'independent errors' (in their behavioural traits) [57] such that they complement each other within a group behaviour. However, the issue of how to identify the optimal group of individuals post training needs to be explicitly addressed. Moreover, it has been noted that providing more explicit mechanisms to maintain diversity (such as regularization, different performance objectives or CC) may provide a more direct method for maintaining diversity (e.g. [149]).

2.3.4 Open-Ended Evolution

While diversity maintenance is the mechanism ensuring that evolving population(s) maintain a rich variety of organisms (structurally and/or behaviourally), open-ended evolution looks at the bigger picture by imagining how a GA might support the continual, unbounded production of novelty over time [10]. Stepney and Hoverd introduce the concept by noting that variation operators applied to an organism's genotype (e.g mutation, crossover) typically generate a constant supply of "new things" (i.e. variations on a theme). Open-ended evolution extends this, implying the production of "new kinds of things" (new species perhaps?) or even "new kinds of new kinds of things" (major evolutionary transitions, radical novelty) [128]. This characterization suggests a critical factor for open-endedness, namely that an open-ended model is one that allows for the emergence of increasingly complex organisms. However, continuous production of novelty does not imply a continuous/monotonic increase in complexity, it simply creates the conditions under which increasingly complex/novel organisms *may* emerge [147].

A body of research has developed over the past 20 years which explores open-ended evolution in significant detail, defining terminology and outlining guidelines for future research [147, 10, 128]. In particular, [10] identifies GP as a fertile area for further study of open-ended evolution. Indeed, there seems to be a lot of potential to draw from the wealth of (primarily theoretical) research on open-endedness when contemplating (bio-inspired) engineering of complex systems. As such, the remainder of this section makes a case for how the GP representations proposed and studied in this thesis fall under the scope of open-ended evolution.

The work in this thesis is concerned with open-ended evolution as it pertains to one form of emergent novelty/complexity in particular, that of a major evolutionary transition [95], or the hierarchical coordination of a number of previously autonomous individuals into a new organism. Thus, individual organisms that previously interacted with their environment and reproduced autonomously transition to interact and reproduce collectively as a whole. Major evolutionary transitions of this nature are classified as *emergent events* in the literature [10], and the capability to produce such events is a marker for open-ended evolution. In this work, Symbiosis (Section 2.2.2) provides the mechanism under which a major transition is simulated and automated within the GA. However, the two representations explored in this work, Policy Trees and Tangled Program Graphs, approach the simulation of major transitions in critically different ways. A high-level discussion of these differences is provided next with illustrations in Figures 2.1 and 2.2. Note that in both cases, the procedure for decision-making (mapping state to action) begins at the root team and follows one directed path to an atomic action. Algorithm details regarding organism development and decision-making are provided in Chapters 4, 5, and 8.

In the case of Policy Trees, only one major transition event takes place, Figure 2.1. Furthermore, a computational budget for the development of each level in the hierarchy (pre and post transition) is chosen a priori, as is the transition point, i.e. the generation at which *all* autonomous organisms (teams) are absorbed by a higher level

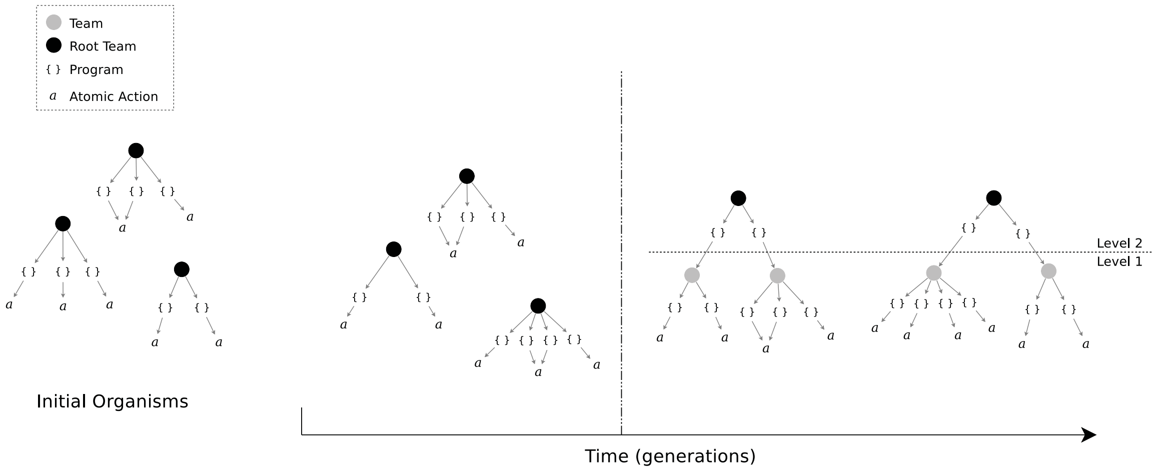


Figure 2.1: Illustration of a major transition in the Policy Tree representation. The transition is planned and parameterized as part of the representation. That is, teams are initialized as autonomous, non-hierarchical entities. Then, at some fixed point during evolution (vertical dotted line), new teams are initialized and a two-level hierarchical relationship between the new and existing entities is enforced. Specifically, teams at the new level must define policies entirely through reuse of the cached team behaviours from the first phase of evolution.

organism. In terms of the open-endedness of the system, this amounts to an implementation shortcut or "cheat" which limits the generality of the model [10]. However, the benefit of this shortcut is that transfer learning is now facilitated. Specifically, lower-level teams can be trained in the *source task* environment and then reused (after the evolutionary transition) by higher level teams learning a more general policy in the more challenging *target task* environment [3]. It will later be suggested (Chapter 5) that population diversity maintenance is important under the first phase of evolution in order to develop a diverse range of 'building block' organisms for later reuse [24, 71]. One implication of this is that it must be possible to explicitly encounter a sufficient range of such building block organisms. Earlier research had demonstrated this using competitive coevolution as the diversity mechanism, i.e. start configurations of the task were also evolved. Specific results using this approach include:

- Policy Trees were capable of generalizing the truck reversal task (with additional hidden wall) for an arbitrary number of start points, whereas ADFs could not [88].

[3] If required, training of lower-level teams in multiple source tasks can be done in parallel. This is demonstrated in Chapter 6

- Policy Trees were also sufficient for matching results by A* for solving the Acrobot Handstand task and generalizing this to arbitrary start points [35].
- Policy Trees could generalize the Pin Ball task [75].

On the other hand, when the start conditions for the task cannot be controlled and/or the task is stochastic, the competitive coevolutionary approach is no longer effective (i.e. organisms are ranked on the basis of their performance on each start condition, but tasks with stochastic properties disrupt the quality of such a ranking).

In the case of Tangled Program Graphs, an unbounded number of hierarchical transitions are fully automated, Figure 2.2. No decisions are made a priori regarding the point at which transitions appear, and TPG allows hierarchical organisms to grow and shrink. That is, increasingly complex hierarchies can emerge at any time and, assuming the increased complexity results in some behavioural advantage, survive indefinitely. However, complex organisms may also incrementally break apart when they are not stable. Thus, organisms at various stages of development (levels of hierarchical complexity) can exists in the same population. Furthermore, unlike Policy Trees, teams at any level may reuse lower-level team behaviours and/or access atomic actions directly, and the interaction among entities is not strictly top-down. Finally, modularity is abundant on a variety of levels. For example, multiple groups of densely interconnected entities interact via less-dense “interface” connections [10] (Red lines in Figure 2.2). For example, groups A, B, and C in Figure 2.2 all represent emergent modular structures of varying complexity (Section 9.5.3). In short, by contrast with a policy tree, interconnecting teams in a graph structure allows for a much richer, modular organization of entities and amounts to a much more open-ended approach to model building.

While TPG extends the open-endedness of Policy Trees by allowing for emergent events (major transitions) without recourse to a pre-coded shortcut, it is still by no means an entirely open-ended system. For example, emergent events in TPG are explicitly *anticipated and captured* [10] by prior structural rules (a special mutation operator in the case of TPG, Chapter 8). A more open-ended approach might include a means of continuously modifying/adapting structural rules to generate unforeseen emergent events. Stepney and Hoverd take a step in this direction through their work on self-modifying and reflective systems [128]. In the specific context of GP,

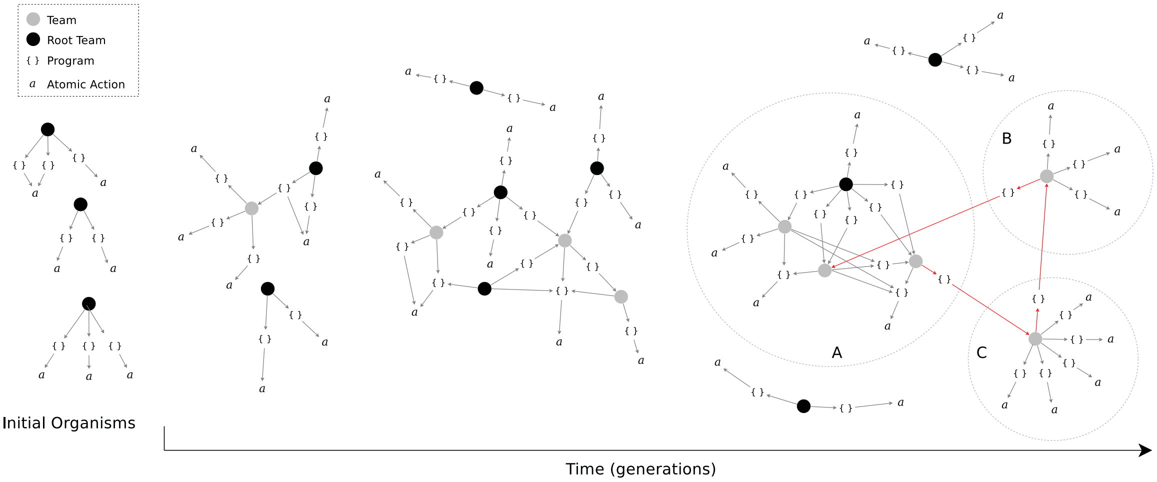


Figure 2.2: Illustration of major transitions and levels of modularity in the Tangled Program Graph representation. See text for details. This figure is intended to contextualize emergent complexity in TPG policies, an implemented GP system, relative to the levels of a theoretical open-ended system illustrated in Figure 2 of [10]. The potential richness of structural diversity and levels of organization in TPG as compared to Policy Trees (Figure 2.1) is apparent.

the idea of concurrently evolving models and diversity-generating mechanisms has a long history, for example, Meta-Genetic Programming [36, 66], Autoconstructive Evolution [126], and SMART operators [148]. However, little is known about how these approaches might lead to emergent novelty/complexity events. In fact, is was noted that maintaining diversity in systems with evolving genetic operators requires special attention [36, 126].

2.4 Hierarchical Reinforcement Learning

The principal goal of Hierarchical Reinforcement Learning (HRL) is to allow sequential decision-making agents to reason at levels of abstraction above the raw sensory inputs and atomic actions assumed by the agent-environment interface. As such, HRL represents an approach to scaling RL by mitigating the effect of the curse of dimensionality [13]. Two types of abstraction are generally considered:

1. *State abstraction* implies that an agent is able to ignore inputs/state variables not relevant to the task at hand [131, 5], which potentially improves learning efficiency by simplifying the input space [60]. Discovering redundancies in this

fashion is distinct from learning a mapping from the complete raw input space to a lower-resolution input representation, as in deep learning [100]. A learned state abstraction of this sort undoubtedly improves learning but does not capitalize on the efficiencies associated with learning when to ignore particular state variables.

2. *Temporal abstraction* implies that the policy can encapsulate a sequence of actions as a single abstract action, or *option* [137]. Options are typically associated with specific subgoals. A higher-level decision-maker is now required in order to map environmental situations to options. Once identified, the selected option assumes control until its (sub)goal state is reached. If a suitable task decomposition is known a priori, multiple options can be trained individually to solve the (human-specified) subgoals and then reused within a fixed hierarchy to solve a more complex task [31]. However, subgoals can also be identified automatically, leading to the automated discovery of options and potentially also learning the hierarchical policy [129, 96, 32, 33].

In considering HRL from the perspective of learning in biological organisms, i.e. the study of brain and behaviour, recent work has placed emphasis on temporal abstractions, and in particular the importance of automatic subgoal and option discovery, where this also implies the emergence of hierarchical structure [18]. From a psychological perspective, the role of "intrinsic motivation", or "curiosity" [116], in motivating an agent's exploration of action-sequences not explicitly rewarded by the environment has been suggested as a way of kick-starting option discovery. Indeed, the concept has shown promise when implemented within HRL [83]. Evolution is also assumed to play a role in the development of hierarchically organized behaviour in animals [47], and several hybrid TD/GA methods have appeared in the HRL literature. For example, evolutionary methods have been proposed which search for the useful (intermediate) reward functions in order for TD methods to reach the overall goal more efficiently, or "shaping rewards" [37]. More generally, an evolutionary search could be performed over the space of possible rewards in order to support continual, systematic exploration [120]. Elfwing et. al. also demonstrate that once subtasks have been defined and learned using TD methods, GP can be employed to evolve option hierarchies [38].

Regarding the specific representations assumed in this thesis, recall that a team of programs represents the simplest stand-alone mapping from state to atomic action, i.e. a stand-alone behaviour. The programs indexed by the team perform their own state variable selection, with individual programs typically making use of a small proportion of available inputs (See Section 4.2, Algorithm 1). As such, teams are capable of identifying redundant inputs and discovering regions of the state space for which the same atomic action applies. In other words, teams directly support state abstraction. Furthermore, a single team need not define a holistic policy over the complete state space. Rather, policies are composed of multiple teams combined hierarchically, each representing a unique state abstraction. Diversity maintenance provides the mechanism for persistent exploration of potentially useful state abstractions, i.e. diverse team behaviours. An argument will be made in Chapter 9 regarding TPG's additional support for temporal abstraction. Finally, Dietterich [31] lists several desirable characteristics of an HRL system, two of which are addressed directly by this thesis: 1) State/temporal abstractions learned under a specific task environment should be re-usable within the hierarchy of policies learning a different but related task; 2) The hierarchy itself should be learned.

2.5 Transfer Learning

Many tasks are too complex to be approached without some amount of decomposition into simpler subtasks. Transfer learning in RL refers to the use of solutions from easier *source tasks* in an attempt to solve a more difficult *target task* [143, 142]. The key issues in transfer learning can be summarized in terms of: 1) what knowledge is transferred; 2) how to implement the knowledge transfer between tasks that may differ with respect to sensors, actions, and objectives, and; 3) How to quantify the utility of transfer. As such, at least from an evolutionary computation perspective, transfer learning in RL has a broad heritage. In particular we identify incremental evolution [43] and layered learning [130] as having similar motivation and sharing the same broad issues.

Incremental evolution assumes that the source tasks follow a sequence (or chain [17]) in which each source task is incrementally more complex than the last. Naturally, it is only possible to solve the first source task in the sequence tabula rasa.[4] Moreover, the issue of what to transfer was defined in terms of a single champion individual that is then used to seed the population for the next source task. Thus, variation operators applied to the genotype of the champion are deemed to provide a suitable starting point for generating content for the next population. Variations on this theme might carry the content of the population 'as is' between consecutive source tasks (e.g., [1]) or transfer multiple champion individuals [56]. Implicit in this process is the approach for addressing the issue of 'how to transfer', i.e. the a priori sequence of source tasks. Moreover, there is sufficient similarity between sensors and actions for the champion individual(s) to directly carry over between consecutive source tasks.

Layered learning [130] is a more recent development and as such captures a wider range of mechanisms in which task transfer might appear. Specifically, rather than assume that source tasks represent a sequence of explicitly related tasks of incremental complexity, layered learning may define source tasks that are initially independent. For each source task a separate training environment with unique states and / or actions may appear. At some point, transfer is accomplished through a gating or switching policy that joins independent solutions to each source task, where such a switching policy might take the form of a prior decision tree (no learning involved) or could also be evolved [159].

The final development that task transfer brings is to consider what happens when some source task action(s) and / or sensors are not present or have altered meaning in the target task [143, 142]. Layered learning assumes that solutions to source tasks do not require any reinterpretation by a later target task. Conversely, policies from source tasks may have incomplete partial alignment, each covering a portion of the sensors and actions available, but jointly still not covering the complete sensor / actuator space of the target task. A learned intertask mapping is one method of addressing this issue [146, 140]. In addition, Hyper-NEAT with a "Bird's Eye View" [154] has been proposed as a way to evolve generalized state representations such that an agent can transfer to more complex tasks without additional learning, assuming the objective remains the same.

In both incremental and layered learning, entire (autonomous) decision-making

[4] Solving a task 'tabula rasa' implies the direct application of RL to a task without any attempt to provide any a priori decomposition into simpler task(s).

policies are transfered from the source to target task, where they either continue adaptation under the more challenging task environment (incremental evolution) or are reused as-is within a hierarchical switching policy (layered learning). Both approaches are adopted separately in this thesis to address distinct goals relative to the utility of transfer. In the empirical evaluation of Policy Trees (Chapter 6), a form or layered learning is used to build behavioural agents for tasks in which Policy Trees would otherwise fail to reach state-of-the-art levels of performance. In the empirical evaluation of Tangled Program Graphs (Chapter 9), a form of incremental evolution is used to enable Multi-Task Learning, i.e. the focus is on building *general* behavioural agents that are capable of performing multiple tasks at state-of-the-art levels of performance.

2.6 Summary

This chapter has outlined methodological building blocks supporting the design and analysis of algorithms within this thesis. Specifically, the proposed representations emphasize modular, hierarchical decision-making agents developed through cooperative coevolution. Code reuse and transfer learning provide the basis for scaling to complex tasks (Chapters 5 and 6) and enabling multi-task learning (Chapters 8 and 9, Section 9.6). In order to include tasks with stochastic and / or little control over initial state, multiple forms of diversity maintenance through genotypic / phenotypic properties will be emphasized (Section 5.4). Moreover, a general push toward open-endedness supports continual exploration of the search space and emergent complexity (Section 9.5.3). The agents' ability to reason at different levels of abstraction is consistent with formulations for HRL, which informs the analysis of their *adapted* sensory interface (Section 9.5.4) and temporal task decomposition (Section 9.5.7).

Chapter 3

Domain Descriptions: RoboCup and Ms. Pac-Man

3.1 Overview

RoboCup 2D Simulated Soccer [134] and Ms. Pac-Man [117] are two widely-used video game domains for evaluating reinforcement learning systems. Both are representative of high-dimensional, dynamic, and stochastic sequential decision-making environments that are significantly more challenging than traditional control-style RL benchmarks (e.g. Multi-Pole Pendulum [44], Acrobot, Mountain Car [135]). This chapter provides a detailed description of each domain and motivates their use as benchmark environments for addressing the problem of scaling to complex tasks through transfer learning. In both RoboCup and Ms. Pac-Man, the learning agent interacts with the environment via a human-crafted sensory interface that includes only pre-processed state variables expected to be useful for decision-making. These include things like distance and angle measurements to other game entities and global information such as the number of opponents present at any given time (Ms. Pac-Man). This is in contrast to the *visual* RL domain described in Chapter 7, in which the agent interacts directly via raw pixel content of the video game screen. Given the simplified input space, the primary challenge in defining strong decision-making policies for RoboCup and Ms. Pac-Man lies in developing multiple modes of behaviour within a single agent. For example, a strong soccer-playing agent requires ball-handling *and* scoring capabilities, while a strong Ms. Pac-Man agent must know how to evade opponents (ghosts) when they are threats and hunt them when they become edible (i.e. sources for a large number of points). Both domains have seen interest from different sets of machine learning researchers. The RoboCup simulated soccer task has frequently been used for investigations into task transfer and multi-agent systems [65, 143]. The Ms. Pac-Man environment has been used for IEEE CEC Competitions and has therefore a range of different approaches demonstrated (Monte Carlo Tree Search [111, 4], Neuro-evolution [117]). The availability of previous results over a cross-section of machine learning methods will facilitate empirical comparison with the methodology proposed in this thesis (Chapter 6).

3.2 RoboCup Domain Description

In RoboCup 2D Soccer, researchers have primarily focused on subtasks of the full game, such as Keepaway [133], in order to more directly assess the specific contributions of various innovations while maintaining essential aspects of the soccer task domain [134, 132, 65, 97, 160, 64, 154]. These tasks are of general relevance to RL because they are:

1. multi-agent, implying that different agents will need to decide on which role they will need to prioritize at different parts of the game;
2. highly dynamic and stochastic;
3. utilizing real-valued state variables; and
4. can be parameterized to represent a family of incrementally more difficult tasks.

It is the latter property that has seen their recommendation as a benchmark for assessing the utility of RL algorithms in general and task transfer in particular [143, 132, 65]. We focus on three specific subtasks: Goal Scoring, Keepaway, and Half Field Offense (HFO).

In **Keepaway**, a team of K *keepers* tries to maintain possession of the ball for as long as possible while an opposing team of $K - 1$ *takers* attempt to gain possession [132]. The game ends if the ball is kicked out of bounds or captured by the takers, at which point each keeper receives the game duration in milliseconds as a reward signal. Thus, the task objective is for keepers to learn a policy to maximize the length of play against the takers, which follow an a priori strategy.[1]

This work specifically targets a 4 versus 3 (4v3), or $K = 4$, version of the task. The version of 4v3 Keepaway in this work is played on a 60×60 meter field, instead of the more common 25×25 meter bounding box, as the ultimate objective is the

[1] The use of an a priori strategy leads to constant benchmarking practices. The default strategy for takers in the Keepaway task is simple: the two takers closest to the ball head directly for it and attempt to gain possession, while the third taker attempts to block passes.

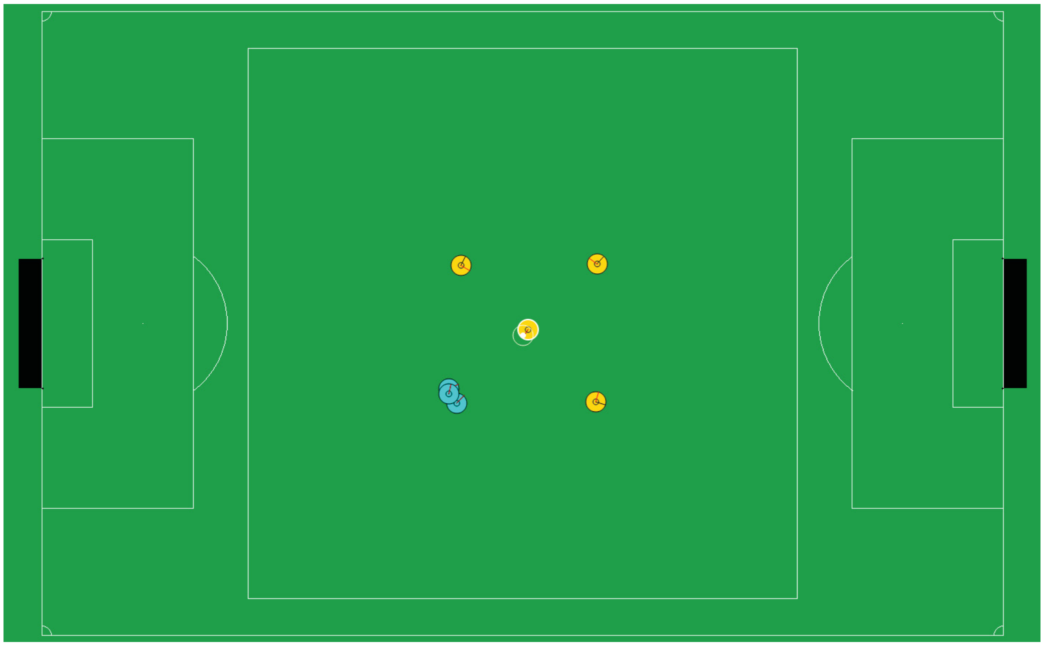


Figure 3.1: Initial positions for keepaway players at the beginning of an episode. Keepers are yellow, takers are blue. Keeper in centre has possession of the ball. White square marks the pay area. See Figure 3.4 and Appendix B, Table B.1 for a description of player state observations.

HFO task as opposed to Keepaway.[2] The game is initialized with keepers in three corners and the centre of a 15 × 15 meter square at centre field, the ball placed near one keeper, and the takers in the remaining corner, Figure 3.1. Keepers learn to deploy a set of domain-specific macro-actions $\{$*Hold,Pass(k)*$\}$, at discrete time steps of 100 milliseconds, Figure 3.2. Each macro-action may last more than one time step, and control is returned to the player when the macro-action terminates. A keeper in possession of the ball has the option of either $Hold$ or $Pass(k)$, where k indexes one of its $K-1$ teammates. Keepers not in possession of the ball assume the *GetOpen* macro-action.

At each time step, keepers receive state information in the form of 11 (egocentric) real-valued distance and angle sensor inputs (state variables) that describe the location of other players on the field (Appendix B, Table B.1). The sensors are noisy, thus players must cope with inaccurate state information. Furthermore, the players'

[2] Generalization between different Keepaway field sizes has been demonstrated [70, 154]. However, it is more difficult to evolve behaviours for smaller fields, where this comes at a considerable computational cost.

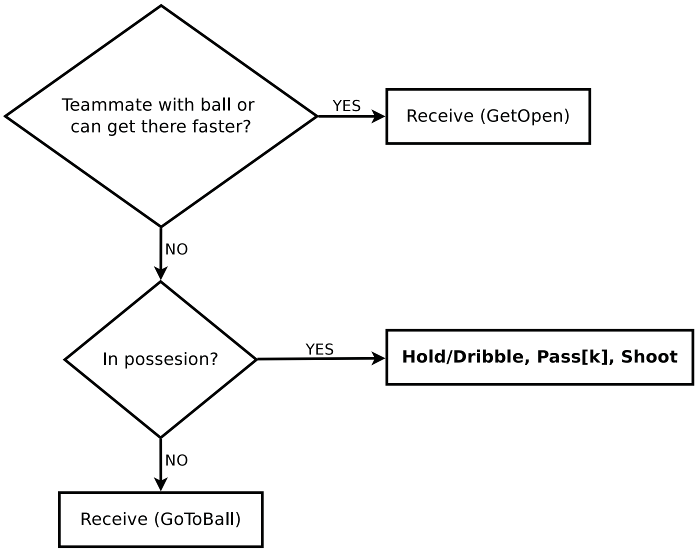

Teammate with ball or can get there faster?
YES
Receive (GetOpen)
NO
In possesion?
YES
Hold/Dribble, Pass[k], Shoot
NO
Receive (GoToBall)

Figure 3.2: Decision tree assumed by RoboCup agents. At each timestep, the agent in possession of the ball selects one macro-action from $\{Hold, Pass(k)\}$ (Keepaway) or $\{Dribble, Pass(k), Shoot\}$ (Scoring/HFO). Players not in possession follow either GetOpen or GoToBall.

actuators are unreliable, often resulting in inaccurate passes and fumbled attempts to hold the ball[3]. The presence of noisy sensors and actuators makes credit assignment particularly difficult and represents one of the issues specifically addressed by the proposed approach.

Half Field Offense is another subtask of RoboCup Soccer with significantly more complexity than Keepaway [65]. In HFO, a team of offense players tries to manoeuvre the ball past a defending team and around the goalie in order to score. A game ends if:

1. the ball is kicked out of bounds,
2. the ball is captured by an outfield defense player,
3. the ball is intercepted by the goalie, or
4. a goal is scored.

[3] The level of noise in state variables is parameterized in the soccer server configuration (Appendix A, Table A.1). In general, the reliability of observations w.r.t. game entities decreases proportional to their distance from the agent. Actuator noise does not replace a particular discrete action with another, but affects the outcome of the selected action. For example, if the agent chooses action *pass to player 2*, the ball's trajectory may be imprecise, forcing player 2 to chase the ball or even fumbling the pass altogether.

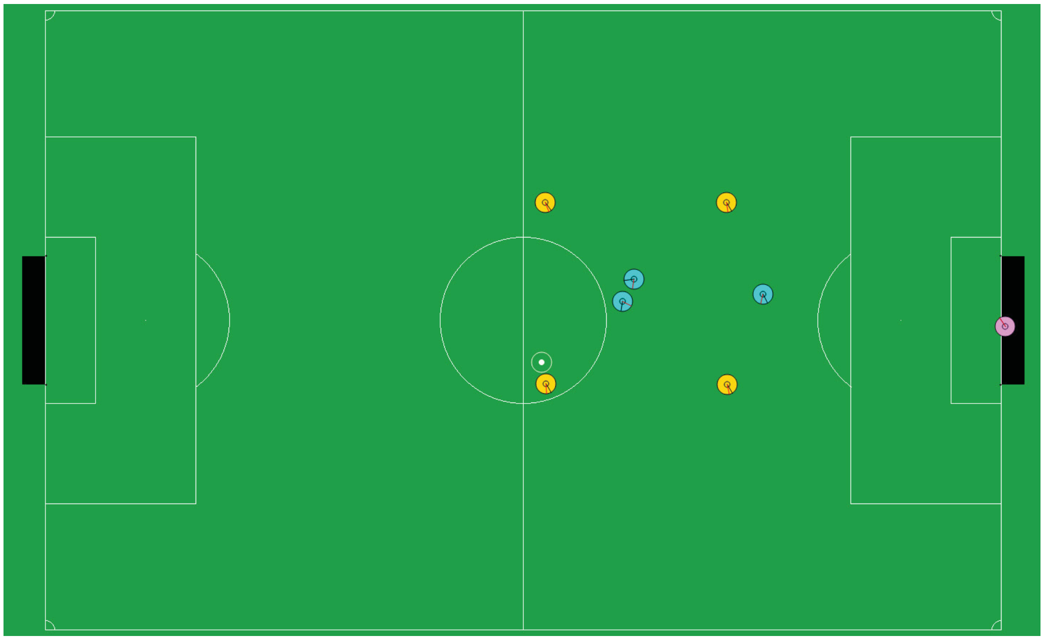


Figure 3.3: Initial positions for players at the beginning of an episode of Half-Field Offense. Offense players are yellow, defense players are blue, goalie is pink. The offense player at bottom left has possession of the ball. See Figure 3.4 and Appendix B, Table B.1 for a description of player state observations.

The defense team in HFO follow a pre-specified behaviour defined by the original task [65][4]. Each offense player receives one reward signal of either $\{0.8, 0.8, 0.9, 2\}$[5] for each of the above end-game conditions.

We are specifically interested in 4 versus 4 (4v4) HFO, in which the offense team has 4 players while the defense team is made up of 3 outfield defenders plus a goalie. HFO is played on one half of the full soccer field. All outfield players are initialized within a 15×15 meter square as in Keepaway, but in HFO the centre of the initialization box is stochastically placed somewhere on the field roughly 55 meters from the goal, Figure 3.3. Naturally, the goalie is initialized in the goal region.

Offense players in HFO learn to deploy a somewhat different set of macro-actions than in the Keepaway task. Their options at each time step are $\{$*Dribble, Pass(k), Shoot*$\}$. Thus, an offense player in possession of the ball has the option to either

[4] The behaviour of the defense team in HFO is the same as for the takers in Keepaway with the addition of a goalie to guard the net.

[5] This reward structure was obtained by adding 1 to the rewards used by Sarsa under the same task [65], thus defining reward in terms of positive values alone. We do not claim any optimality for these values from an evolutionary computation perspective.

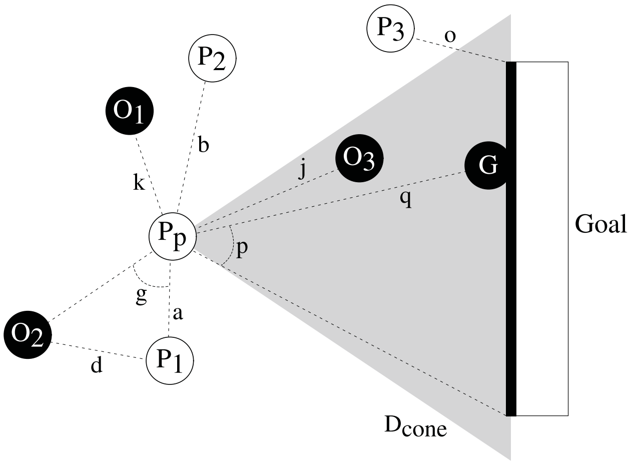


Figure 3.4: Example sensor inputs for the RoboCup environment. See descriptions in Appendix B, Table B.1. Sensors that reference the goal or goalie are specific to the Half-field Offense task. P_j denote players whereas P_p is the special case of the player from which egocentric measurements are made, G is the special case of the 'goalie' opponent, O_i denote opponents. Equivalent measurements are repeated for each player. Used by permission, ©2017 IEEE.

dribble towards the goal, pass to a teammate, or shooting on goal, Figure 3.2. As per the Keepaway task, each macro-action may last more than one time step and control is returned to the player when the macro-action terminates. The HFO task has the same 11 sensors available in Keepaway, plus 6 *additional* sensor inputs that summarize distance to the goal for each offense player and the angle of the maximum scoring window for the player with the ball, i.e. how open the net is (Figure 3.4 and Appendix B, Table B.1). As in Keepaway, all sensor inputs and actuators are unreliable. Readers are refereed to the original HFO paper for a more detailed definition of the task [65].

The **Goal Scoring** task is similar to the HFO environment except that the initial position of players is restricted to within 15 meters of the goal. While some passing and ball handling will still be necessary from this position, offense players no longer need to manoeuvre the ball down field and can therefore concentrate on scoring goals.

3.3 Related Work in RoboCup

Keepaway is the most widely used subtask of RoboCup soccer, representing a benchmark for both multi-agent and reinforcement learning / policy search in general

[133, 134, 98, 160]. Given that a decision maker is necessary for each keeper, reinforcement learning approaches have adopted a heterogeneous assignment of learners to keepers[6], where this is a function of the overhead in attempting to update a single function approximator w.r.t. multiple keepers [160]. Conversely, (evolutionary) policy search generally assumes a homogeneous assignment, where this is a reflection of the lack of specialization required in the keeper policies [157]. Homogeneous teams are developed in this thesis under all three RoboCup subtasks, primarily due to the simplicity of implementation and minimal computational cost relative to heterogeneous teams.

In applying value function optimization methods to learn Keepaway policies, the first obstacle to be addressed was how to formulate the task such that credit assignment mechanisms such as Sarsa(λ) could be applied [133, 134]. With this achieved, most emphasis has been on the type of function approximation used to model Q-values. Thus, function approximation based on tile coding [133, 134] has been superseded by the use of Radial Basis Functions [160] or kernel methods [61]. Several approaches to neuro-evolution have also been applied to the Keepaway task, including NEAT [145, 160] and EANT [98]. It is worth noting that all neuro-evolution schemes make extensive use of genotypic diversity for maintaining multiple species during a run.

Having established some benchmark capabilities for a variety of value function optimization and policy search methods, the complexity of RoboCup tasks subsequently motivated a shift in focus towards leveraging transfer learning in order to scale policy development. In the earliest of this work, the key to transfer was a hand-coded mapping between the differing state and action spaces of increasingly complex tasks, for example [144, 146]. Later work introduced various approaches to automating transfer, for example [154, 40], and even considered reusing experience from outside of the RoboCup domain [141].

In the case of GP, a layered learning approach has been adopted in the past to facilitate the incremental evolution of tree structured GP, with and without ADFs [48, 56]. However, these results are reported for a different soccer simulator (TeamBots) and hence different atomic actions. Layered learning, as a form of transfer learning,

[6] Heterogeneous teams imply that a unique policy is developed specifically for each player on a team. Conversely, homogeneous teams imply that the same policy controls all players.

assumes that the task undergoes some prior decomposition with training performed relative to the simpler tasks first. However, it was also necessary to enforce a prior discretization of the state variables (i.e., a simplification of the task) and limit the number of takers to 1 (i.e., 3-versus-1 keepaway).

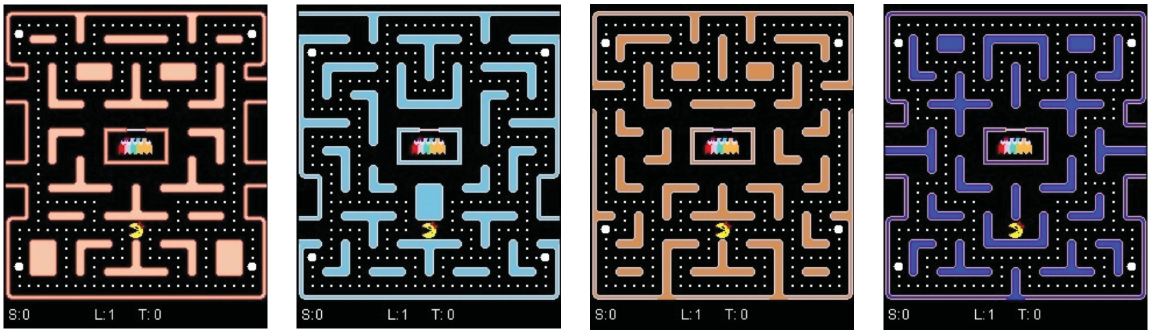
At the time of this writing, the HFO task has received significantly less attention than Keepaway. The most successful approach to HFO has been from the Sarsa(λ) value function method, where studies have explored the utility of inter-agent communication [65] and ad hoc teamwork [50, 12]. While not identical to HFO, Torrey et. al. [150] define a similar RoboCup task they call "Breakaway" for which they employ transfer learning from multiple simpler source tasks, specifically Keepaway and a custom "MoveDownField" subtask. Unlike the work in this thesis, their approach relies on a human-coded mapping which specifies the similarities between the source and target tasks.

Unfortunately, RoboCup has not received a significant amount of attention in very recent years, probably due to the programming overhead required to establish a reliable agent-environment interface and/or the computational costs associated with evaluation in multi-agent domains [50, 139]. However, notable works have appeared, for example, recent results from Deep Q-Learning for the keepaway task represent the best score reported to date [84]. Interestingly, the authors found that some aspects of the algorithm were over-kill, for example, a shallow network with just two hidden layers was sufficient. Tavafi's work on scaling GP to the full RoboCup 2D simulation task [139] is also notable. To do so, Tavafi first identified and hand-labeled examples of 'good' decision-making, or "snapshots", by observing the strategies of champion teams from the RoboCup 2016 competition. Evolution in his GP framework was divided into two phases. In the first phase, GP individuals were trained through supervised learning of snapshots, essentially learning to model a champion player's policy. In the second phase, the best individuals from the first phase were developed further through direct interaction with the task. In contrast to the state representation assumed in this thesis (Appendix B, Table B.1), Tavafi used high-level inputs that specifically encode 'good' and 'bad' states such as "weAreWinning" and "ballInDangerArea", i.e. a significant prior task simplification.

In this work, the goal is to scale the capability of Policy Trees to HFO through transfer learning. Agents will begin by learning Keepaway and Scoring *source* tasks separately, and then transfer this experience to Half Field Offense. Keepaway and Goal Scoring are clearly subtasks of HFO. For example, from the stochastic start position in HFO, offensive players essentially have to maintain possession and dribble towards the goal before trying to score. However, the ball-handling skills under HFO are different from Keepaway, requiring the players to maintain possession of the ball *and* create goal scoring opportunities. Furthermore, players that only have experience in the Scoring task, having only observed the environment near the goal region, will need to cope with larger distance and angle sensor readings under the HFO task. Thus, success in the HFO environment requires additional capabilities and reinterpretation of skills learned from the source tasks.

3.4 Ms. Pac-Man Domain Description

The Ms. Pac-Man simulation assumed in this work is a close approximation of the 1982 arcade version of the game, which is one of the most popular video games of all time. The objective is to identify a policy for guiding Ms. Pac-Man through a series of 4 mazes, each scattered with multiple regular pills and 4 power pills, Figure 3.5. When Ms. Pac-Man finds a pill, she eats it, gaining 10 points per regular pill and 50 points per power pill. When all pills have been eaten, Ms. Pac-Man advances to the next maze. However, the game includes 4 opponents, or ghosts, which emerge sequentially from a Lair in the centre of each maze and pursue Ms. Pac-Man. If a ghost touches her, she loses a life. Ghosts behave non-deterministically, necessitating a responsive strategy to out-maneuver them, rather than simply memorizing an effective trajectory through the maze. Normally, Ms. Pac-Man and the ghosts move at the same speed. However, when Ms. Pac-Man eats a power pill, all ghosts become edible and reduce their speed by 50%, making it possible for Ms. Pac-Man to catch and eat them. The 1^{st}, 2^{nd}, 3^{rd}, and 4^{th} ghosts eaten are worth 200, 400, 800, and 1600 points, respectively. Ms. Pac-Man is therefore a predator-prey scenario that requires the agent to switch between multiple modes of behaviour throughout an episode. The agent must assume the role of predator *and* prey at different times, actively pursuing ghosts when they are edible and evading ghosts when they are threats [117]. Multi-modal behaviour of this nature is similar to the the ball-handling and scoring



(a) Maze 1 (b) Maze 2 (c) Maze 3 (d) Maze 4

Figure 3.5: 4 unique mazes and initial positions for Ms. Pac-Man.

requirement in the Half Field Offense subtask of RoboCup soccer. However, unlike HFO, multiple mode transitions are explicitly required in Ms. Pac-Man. This, coupled with the fact that a single game in Ms. Pac-Man requires thousands of decisions from the agent, places additional burden on credit assignment and makes the task particularly difficult.

In each time step, there are a total of 95 sensors: 7 non-directional sensors (e.g. number of regular pills left in maze, number of edible ghosts, remaining edible ghost time) and 22 directed sensors for each of the 4 directions (e.g. distances to the 1^{st}, 2^{nd}, 3^{rd}, and 4^{th} closest ghosts, whether each ghost is approaching, distance to nearest power pill and maze junction). Tables C.1 and C.2 in Appendix C list all undirected and directed sensors respectively. For a more detailed description of each sensor, see 'Conflict Sensors' in [117].

3.5 Related Work in Ms. Pac-Man

Best results on this task to date have all benefited from incorporating some form of task-specific bias. Specific examples include assumptions regarding the optimal amount of modularity to include under modular neuro-evolution [117], or providing appropriate end game tactics for use under Monte-Carlo Tree Search [111]. Schrum and Miikkulainen [117] provide two key insights into learning Ms. Pac-Man agents:

1. Ms. Pac-Man can be formulated as a multi-objective problem in which maximizing the number of pills and ghosts eaten are treated as distinct objectives, and

2. Explicitly encouraging modularity in the learning representation facilitates the discovery of multi-modal behaviour.

The prior task decomposition required by the former insight suggests that transfer learning may be applicable to this domain. Furthermore, the requirement for modularity is perfectly suited to the team GP method proposed in this work.

Similar to decomposing the objective space a priori, various GP methods have made effective use of prior game knowledge in configuring the environment. Alhejali and Lucas [3] designed multiple tightly-constrained game scenarios, or training camps, in which the agent was trained relative to specific objectives prior to attempting to learn a more general controller. Their GP also employed high-level sensors, such as those directly alerting Ms. Pac-Man to dangerous states, and high-level actions that, for example, may assume control for multiple time steps and lead Ms. Pac-Man out of dangerous states. However, GP restricted to low-level actions and sensors has also been successful in this task [21]. Low-level sensors include only general information like 'distance to the nearest pill', and thus do not imply a specific interpretation by the agent, ie. 'good' and 'bad' states. Low-level actions imply that the Ms. Pac-Man controller simply selects a direction to move (UP, DOWN, LEFT, or RIGHT) in each time step of the game. Brandstetter *et al.* [21] demonstrated how a GP individual with low-level actions and sensors can be designed to evaluate and rate each direction, then simply move in the direction with maximum rating. Our work extends this to the case in which polices consist of multiple cooperative GP individuals, or teams. As such, the experiments in this thesis assume low-level actions and sensors for Ms. Pac-Man.

In this thesis, Ms. Pac-Man is framed as a suitable case for transfer learning in which the 2 broad objectives of the game, eating pills and eating ghosts, are treated as source tasks with simple reward signals equal to the number of pills/ghost eaten, with overall game score as the target task. While both source objectives must be maximized to achieve high scores, the relative benefit in pursuing one objective over the other needs to be managed effectively by the agent under the target task. For example, eating pills too aggressively would result in clearing the maze before all the ghosts are eaten, missing out on significant scoring opportunities, while eating ghosts too aggressively leaves Ms. Pac-Man vulnerable to threat ghosts as she clears pills from difficult regions of the maze. As such, behaviours developed under the source task(s) must be adapted, or re-contextualized under the target environment. Furthermore, Ms. Pac-Man requires specific skills for evading threat ghosts and luring both threat and edible ghosts into disadvantaged positions, neither of which are explicitly rewarded in either the source or target objective functions.

3.6 Summary

This chapter has aimed to provide a complete but concise overview of the two task domains used to evaluate Policy Trees in Chapter 6, paying specific attention to why they present interesting benchmarks for the application of transfer learning. Both domain simulators have complex dynamics and extensive procedural rules defining gameplay and opponent behaviours. However, since the subject of this research is primarily bio-inspired engineering of complex systems as opposed to (video) game theory, an exhaustive description of gameplay is unnecessary for understanding the contributions herein. Further details for Robocup Soccer and Ms. Pac-Man are available from [130, 65] and [117] respectively.

Chapter 4

Algorithm Description: Teams of Programs

4.1 Overview

The two representations considered in this thesis, Policy Trees and Tangled Program Graphs, provide different approaches to constructing hierarchical decision-making policies by developing the interdependence between multiple teams of programs. In both cases, evolution begins with policies in their simplest form, i.e. each organism/agent is a single team of programs as defined by the SBB algorithm [88]. This chapter provides a technical specification for individual programs, the group decision-making process within a team (i.e. the process for a team to map a state observation to an action), as well as initialization and variation procedures for both team and program. These are the aspects of team GP that are common to both Policy Trees and TPG, and will therefore be established generically here and referenced later with respect to the differences between the two approaches. In particular, Policy Trees and TPG have slightly different rules with respect to applying variation operators at the team-level, resulting in the development of significantly different kinds of structural modularity and hierarchy (See Chapter 2, Figures 2.1 and 2.2). Furthermore, team development in Policy Trees and TPG is driven by significantly different overall training algorithms. Thus, from the perspective of evolutionary model building, this chapter establishes the variation operators and representations for individual programs and teams or programs (i.e. the GP)[1] while Chapters 5 and 8 detail the higher-level procedure for building (hierarchical) models through interaction with the task environment (i.e. the GA).

[1] The approach to GP teaming described in this chapter is nearly identical to that in [88] and is reproduced here with slight modification to accommodate a more general approach to the hierarchical construction of policies.

4.2 Teams of Programs

The two basic entities of this model, teams and programs, are stored in separate populations and coevolved (See Figure 1.3(a)). A team is simply a set of pointers to members of the program population, while programs represent value functions for state/action pairs. Thus, each program's role within the team is to define a unique *context* for one discrete action. In this work, programs are linear register machines [20] (See Algorithm 1)[2]. In order to map a state observation to an action in sequential decision-making tasks, each program in the team will execute relative to the current state, $\vec{s}(t)$, and return a single real valued 'bid', i.e. the content of register $R[0]$ after execution. The team then deploys the action of the program with the highest output, or the *winning bid.* [3]

Algorithm 1 Example program in which execution is sequential. Programs may include two-argument instructions of the form $R[i] \leftarrow R[x] \circ R[y]$ in which $\circ \in \{+, -, \times, \div\}$; single-argument instructions of the form $R[i] \leftarrow \circ(R[y])$ in which $\circ \in \{cos, ln, exp\}$; and a conditional statement of the the form IF $(R[i] < R[y])$ THEN $R[i] \leftarrow -R[i]$. $R[i]$ is a reference to an internal register, while $R[x]$ and $R[y]$ may reference internal registers or state variables (sensor inputs). All registers are set to 0 prior to program execution. If the result of an operation is undefined, e.g., division by zero, then zero is stored in the destination register. Determining which of the available state variables are actually used in the program, as well as the number of instructions and their operations, are both emergent properties of the evolutionary process.

```
$R[0] \leftarrow R[0] - R[3]$
$R[1] \leftarrow R[0] \div R[7]$
$R[1] \leftarrow Log(R[1])$
IF $(R[0] < R[1])$ THEN $R[0] \leftarrow -R[0]$
RETURN $R[0]$
```

4.2.1 Initialization

A two-part process is assumed for initializing the team and program populations:

[2] Any GP representation could be employed, the important innovation is that context and action are represented independently.

[3] If programs were not organized into teams, in which case all programs within the same population would compete for the right to suggest their action, it is very likely that degenerate individuals (programs that bid high for every state), would disrupt otherwise effective bidding strategies [89]

1. New teams are created such that each team contains two new programs (Program initialization is detailed in Section 4.3). The program actions are uniformly selected from an action set $\mathcal{A}'$[4] under the constraint that each program (within the same team) has a *different* action. Thus, after part one of the initialization process, the program population is exactly twice the size of the team population, Figure 4.1(a)

2. The target size for each new team is selected with uniform probability from $[2, ..., \omega]$, where ω is the initial maximum team size. A program *mixing* heuristic is then applied to add new team-program pointers such that each program is (initially) a member of roughly the same number of teams. Specifically, for each new team, the mixing procedure uniformly selects two programs at a time from the set of programs *not* already part of the team, adds whichever program is currently a member of the fewest teams, and repeats the process until the new team reaches its target size, Figure 4.1(b).

Thus, after part two, the size of the program population remains at twice the size of the team population, but each program potentially appears in multiple teams. This is significant because programs that appear in multiple teams are evaluated in multiple group behaviours, giving the evolutionary search a better chance of discovering a particular group in which they prove useful.

4.2.2 Variation

Variation of existing genetic material is the mechanism through which novel organisms are generated within evolving populations. In this work, team variation operators are asexual and take the form of a set of mutation operators incrementally applied to a team and some of its programs. GP has been demonstrated without any crossover (i.e. sexual recombination of individuals) and empirically found to perform as well as GP with crossover if multiple forms of mutation are included [26]. The forms of mutation adopted in this work facilitate the exchange of programs between teams.

[4] When referring to the set of actions available to programs, the convention in this thesis is to use $\mathcal{A}$ to denote the set of atomic actions assumed by the task environment, and $\mathcal{A}'$ to denote a generic action set. Depending on the context, $\mathcal{A}'$ will either be equivalent to $\mathcal{A}$ or, in the case of team hierarchies, $\mathcal{A}'$ may refer to a set of teams.

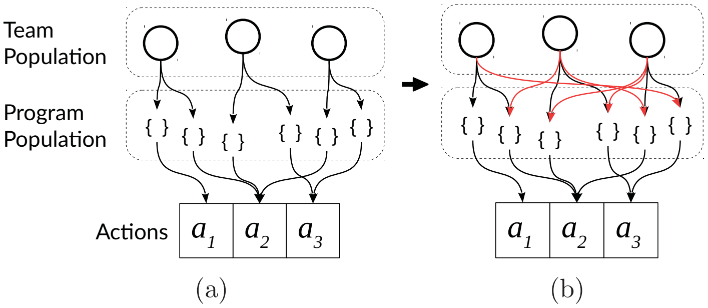


Figure 4.1: Two-step initialization procedure for team and program populations. Each team is initialized with two new programs, each associated with a different action (a). A program mixing heuristic is then applied to add new team-program pointers (red lines), distributing each program among multiple teams (b).

Hence, we are able to promote the circulation of 'useful' programs between parents and offspring, i.e. inheritance is supported.

In this work, the absence of a crossover operation implies that symbiosis (Section 2.2.2) is the only process through which genetic material from two independent organisms is combined and (hopefully) exploited such that the new organism is greater than the sum of parts. In particular, Chapters 5 and 8 will extend these variation algorithms to establish how mutation operators lead to *hierarchical* symbiosis, potentially combining multiple previously independent teams. In any case, the variation operators applied to a team tm_i assume the following sequence of operations:

1. Add program(s) to tm_i: Adding programs to teams implies that we sample from the set of programs available after deleting the worst teams and any 'orphaned' programs. Thus, the only programs left are those that are currently associated with the best performing teams. Such a process promotes the reuse of the more 'successful' programs.

2. Remove program(s) from tm_i: Provides a path for revisiting team program complement. That is to say, as programs are added to a team, redundancies might occur. In the worst case, hitchhiking programs can appear that never contribute a winning bid.

3. Mutate program(s) within tm_i: Represents a process for maintaining diversity at the level of programs as opposed to teams.

The add, remove, and mutate program procedures are described in detail below. Table 4.1 provides a description of team variation parameters.

Adding or Removing Programs

Algorithm 2 defines the procedure for adding or removing a program from a team. Following the variation sequence above, at least one program will be added and one program will be removed from the team. That is, Algorithm 2 will be called twice, once with $\text{OP} = \textit{remove}$ and once with $\text{OP} = \textit{add}$.

Algorithm 2 Procedure for adding or removing a program to/from team tm_i. $\text{OP} \in \{add, remove\}$ defines which operation is performed. At least one program will be affected. Thereafter the likelihood of repeating the operation decreases by a factor of p_{ma}. P represents the current program population. The function $rand(0, 1)$ returns a uniformly selected real value in the interval $[0, 1.0)$. Parameters are listed in Table 4.1

```
procedure MutateTeam($tm_i$, OP)
    $b = 1$
    while $b > rand(0, 1)$ and $tm_i$ has between 3 and $\Omega$ programs do
        if $\text{OP} = add$ then
            uniformly sample $p_i \in P : p_i \notin tm_i$
            add pointer from $tm_i$ to $p_i$
            $b = b \times p_{ma}$
        else
            uniformly sample $p_i \in tm_i$
            remove pointer from $tm_i$ to $p_i$
            $b = b \times p_{md}$
        end if
    end while
end procedure
```

Mutating a Program

Algorithm 3 defines the procedure for mutating a program within a team. Again, at least one program will be affected. Every program in the team is considered for mutation with probability p_{mm}. The program is cloned prior to mutation and the cloned program's bidding behaviour is modified (Section 4.3). The cloned program's action pointer is modified with probability p_{mn}. Cloning the program prior to mutation ensures that other teams in which the selected program may appear are unaffected by the mutation.

Algorithm 3 Procedure for mutating program(s) in team tm_i. Each program $p_i \in tm_i$ is considered in turn and mutated with probability p_{mm}. Note that programs are removed from the team and copied prior to mutation, where the *copy* is inserted back into the team and then modified (Lines 5 - 7). P is the current program population. When modifying a program's action, $\mathcal{A}'$ represents the action set from which to sample the new action. The function $rand(0, 1)$ returns a uniformly selected real value in the interval $[0, 1.0)$. The process for modifying bid programs, Line 8, is detailed in Algorithm 4. $action(p_i)$ refers to the action associated with program p_i. Parameters are listed in Table 4.1.

```
 1: procedure MutateProgram($tm_i$)
 2:     do
 3:         for all $p_i \in tm_i$ do
 4:             if $rand(0, 1) < p_{mm}$ then
 5:                 copy $p_i$ into $p'_i$
 6:                 remove pointer from $tm_i$ to $p_i$
 7:                 add pointer from $tm_i$ to $p'_i$
 8:                 mutate bid program of $p'_i$                ▷ See Algorithm 4
 9:                 if $rand(0, 1) < p_{mn}$ then
10:                     change $action(p'_i)$ to a uniformly selected action $a \in \mathcal{A}'$
11:                 end if
12:                 insert $p'_i$ into $P$
13:             end if
14:         end for
15:     while no program has been modified
16: end procedure
```

Table 4.1: Parameters associated with creating and modifying teams.

Parameter	Description
p_{md}	Probability of program deletion
p_{ma}	Probability of program addition
p_{mm}	Probability of program mutation (bidding behaviour)
p_{mn}	Probability of program mutation (action pointer)
ω	Maximum initial team size
Ω	Maximum team size

4.3 Linear GP Implementation

Programs in this work are linear register machines [20], Algorithm 1. Their implementation is nearly identical to previous work [88]. Implementation details from [88] are reproduced here to ensure the algorithm description in this thesis is self-contained and complete.

4.3.1 Encoding

Instructions are encoded as fixed bit strings with the following four fields:

1. *destination* (3 bits): encodes the register index x where the first operand is located and where the result of the operation is stored.

2. *source* (16 bits): encodes index y where the second operand is located. y may refer to either a register or an input depending on the instruction mode.

3. *mode* (1 bit): encodes whether index y is a reference to a register or input.

4. *operation* (3 bits): encodes the function to be applied to the values indexed by x and y.

A program is then defined as a linear sequence of instructions operating on inputs (state variables) and internal registers, Algorithm 1. Note that the caption of Algorithm 1 also provides the complete instruction set used in this thesis.

4.3.2 Initialization and Variation

New programs are created through a simple two-step process (parameters associated with creating and modifying programs are listed in Table 4.2):

1. Select the program size, or number of instructions, with uniform probability in $\{1, 2, ..., maxProgSize\}$.

2. Set each bit in the new program to a random value selected with uniform probability.

Algorithm 4 defines the procedure to modify the bidding behaviour of a program (bid mutation step in Line 8 of Algorithm 3).

Algorithm 4 Procedure for modifying the bidding behaviour of program p_i. The function $rand(0,1)$ returns a uniformly selected real value in the interval $[0, 1.0)$. Parameters are listed in Table 4.2

```
procedure MutateBidBehaviour($p_i$)
    if $p_i$ has more than one instruction and $rand(0,1) < p_{delete}$ then
        delete a uniformly selected instruction
    end if
    if $p_i$ has less than $maxProgSize$ instructions and $rand(0,1) < p_{add}$ then
        create new instruction $i_i$ with uniformly selected bit values
        insert $i_i$ into $p_i$ at uniformly selected location
    end if
    if $rand(0,1) < p_{mutate}$ then
        flip uniformly selected bit in $p_i$
    end if
    if $p_i$ has more than one instruction and $rand(0,1) < p_{swap}$ then
        swap the locations of two uniformly selected instructions
    end if
end procedure
```

4.3.3 Neutrality Test

When variation operators introduce changes to a program, there is no guarantee that the change will: 1) result in a behavioural change, and 2) even if a behavioural change results, it will be unique relative to the current set of programs. Point 1 is still useful as it results in the potential for multiple code changes to be incrementally built up before they appear, or neutral networks [20]. However, this can also result in wasted evaluation cycles because there is no functional difference relative to the parent. Given that fitness evaluation is expensive, we therefore test for behavioural uniqueness. Specifically, a set of the most recent state observations over all teams are retained in a global archive[5]. When a program is modified or a new program is created, its bid for each state in the archive is compared against the bid of every program in the current population. As long as all bid values from the new program are not within τ of all bids from any other program in the current population, the new program is accepted. If the new program fails the test, then the program variation operations (Algorithm 4) and neutrality test are repeated.

[5] Each state observation made by any team during evaluation is added to the archive with probability $p_{archive}$. The archive assumes a first-in-last-out structure with max size arc_{size}.

Table 4.2: Parameters associated with creating and modifying programs.

Parameter	Description
$numRegisters$	Number of registers
$maxProgSize$	Maximum program size
p_{delete}	Probability of instruction deletion
p_{add}	Probability of instruction addition
p_{mutate}	Probability of flipping a single bit
p_{swap}	Probability of swapping two instructions
$p_{archive}$	Probability of adding state observation to archive (Section 4.3.3)
arc_{size}	Size of global observation archive used for neutrality test
τ	Bid similarity threshold

4.3.4 Identification of Ineffective Code

In practice, $60 - 70\%$ of program instructions may have no effect on the program's output [20]. These 'introns' can be identified and marked prior to evaluation. As such, introns can be skipped during program execution, significantly reducing the computational cost of evaluating policies. The procedure for identifying introns is detailed in Algorithm 5. Note that introns are important for the development of programs and are never removed. For example, introns protect effective program code from potentially detrimental variation and support neutral variations, allowing variation operators to make several incremental modifications to a program before their (cumulative) affect on behaviour appears [20]. The procedure for identifying introns is detailed in Algorithm 5.

Algorithm 5 Algorithm for identifying instructions that have no effect on program output, or the content of register 0 after execution. The variable *ip* is an instruction pointer, $*ip$ denotes an instruction, and the program size is N. The procedures in lines 5 and 8 are used to extract the destination and source registers respectively, while line 7 tests if the y operand refers to a register. The introns are recorded in set I.

```
 1: procedure FindIntrons
 2:     $I = \emptyset$                              ▷ Set of introns.
 3:     $T = \{0\}$                                  ▷ Set of target registers.
 4:     for $ip = N - 1$ to 0 do
 5:         $x =$ GetDestination($*ip$)
 6:         if $x \in T$ then                        ▷ Instruction $*ip$ is effective.
 7:             if yIsRegister($*ip$) then
 8:                 $y =$ GetSource($*ip$)
 9:                 $T = T \cup \{y\}$
10:             end if
11:         else                                     ▷ Instruction $*ip$ is an intron.
12:             $I = I \cup ip$
13:         end if
14:     end for
15: end procedure
```

Chapter 5

Algorithm Description: Policy Trees

5.1 Overview

Having established a general specification for teams of programs in chapter 4, this chapter describes the hierarchical organization of multiple teams into *policy trees*. In particular, teams of programs are coevolved over two distinct phases of evolution. The first phase produces a library of diverse, specialist teams of limited capability (Phase 1, Figure 5.1). The second phase builds more general and robust policies by reusing the library, essentially building generalist strategies from multiple specialists (Phase 2, Figure 5.1). Thus, diversity maintenance is critical during the first phase of evolution to ensure the identification of a wide range of specialist behaviours.

This chapter proceeds as follows to describe the complete algorithm for developing policy trees. Section 5.2 defines how the team initialization and variation procedures are extended to support policy trees constructed over two phases of evolution. Section 5.3 defines the procedure for decision-making, or how policy trees interact with the task environment during evaluation. Section 5.4 details how results of evaluation are used to determine which policies will survive to reproduce, or the *selection* procedure. This is where diversity maintenance through fitness regularization plays a critical role. In particular, in phase 1 of evolution selection pressure is applied based on performance in the task *and* novelty with respect to other policies in the same population. Finally, Section 5.6 establishes how the entire GA procedure (Figure 1.2) is repeated over two phases of evolution to specifically enable transfer learning. In phase 1, multiple initializations run in parallel to develop solutions for multiple source tasks. In phase 2, a single GA develops hierarchical policy trees that reuse the solutions for source tasks to solve a more complex target task.

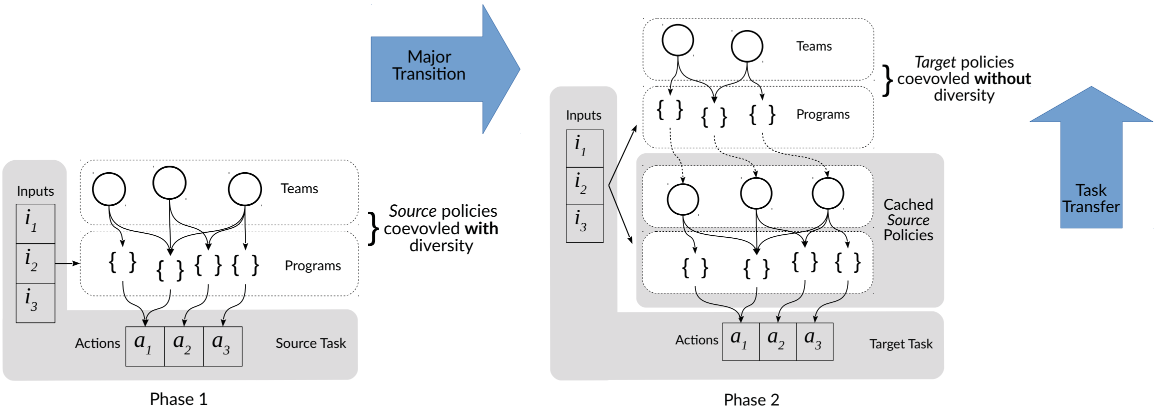


Figure 5.1: Illustration of the primary concepts in the development of policy trees: 1) Hierarchical policies are developed over two phases of evolution separated by a single (major) transition event; 2) Phase 1 evolves single-team policies with diversity maintenance, resulting in a library of specialist behaviours. Phase 2 evolves policy trees that reuse the library. Policy trees are evolved without diversity maintenance to encourage progress to the ultimate objective; 3) Transfer learning is enabled by increasing the task complexity between phases. Phase-1 policies learn a simpler source task while phase-2 policies learn a more difficult target task.

5.2 Initialization and Variation

From a structural perspective, the only difference between phase-1 and phase-2 policies is the action set available to programs. Thus, the generic team initialization and variation procedures (Sections 4.2.1 and 4.2.2) apply directly to both phases of the evolution of policy trees, where the action set $\mathcal{A}'$ for evolving programs is dependent on the phase. Specifically, programs in phase 1 are limited to atomic actions defined by the task environment. In phase 2, program actions refer exclusively to team behaviours evolved during phase 1. As such, the transition from phase 1 to phase 2 represents a major evolutionary transition from single-team to hierarchical, multi-team organisms, or policy trees (Figure 5.1).

5.3 Decision-Making in Policy Trees (Evaluation)

Decision-making in a policy tree follows the same bidding mechanism established in Section 4.2 for a single team of program, but in policy trees decision-making is a recursive procedure beginning at the root team and following one path through the tree (of teams) until an atomic action is reached, Algorithm 6. The process begins with the identification of a winning program in the root team relative to the current state observation. However, at this point the action is a previously evolved team as discovered during phase 1, i.e. a meta action. Thus, the meta action is now executed for the same state variables, with the winning program this time referencing an atomic action from the task environment. In sequential decision-making tasks, evaluation for the current policy begins in a *start state* defined by the task environment (See Chapter 3) and continues until an episodic *end state* is encountered i.e., for each new state of the task, the policy tree executes Algorithm 6 to select an atomic action, in each case potentially updating the state of the task. At the end of an episode, the task environment returns a final reward signal that characterizes the quality of the policy's decision-making during that episode. In order to mitigate the effect of noisy evaluations in stochastic tasks, the fitness of a policy is often defined by the mean reward across multiple evaluations in the task environment.

Algorithm 6 Generic process for selecting an atomic action with a single team of programs (phase 1) or through traversal of a policy tree (phase 2). tm_i is the current team (initially the root node in the case of a policy tree). $\vec{s}(t)$ is the vector of state variables representing the current environmental observation at time t. $\mathcal{A}$ is the set of atomic actions. First, all programs in tm_i are executed relative to the current state $\vec{s}(t)$ (Lines 2,3). The algorithm then identifies the winning program as that with the maximum bid (Line 5). If the winning program has an atomic action, the action is returned (Line 7). Otherwise, $action(p_i)$ is a reference to a previously evolved (source) policy/team, hence the process is called recursively on that team (Line 9).

```
 1: procedure SelectAction($tm_i, \vec{s}(t)$)
 2:     for all $p_i \in tm_i$ do
 3:         $bid(p_i) = exec(p_i, \vec{s}(t))$              ▷ run program on $\vec{s}(t)$ and save result
 4:     end for
 5:     $p_i^* = \arg_{p_i \in tm_i} \max[bid(p_i)]$          ▷ identify program with maximum bid
 6:     if $action(p_i^*) \in \mathcal{A}$ then
 7:         return $action(p_i^*)$                              ▷ atomic reached
 8:     else
 9:         return SelectAction($action(p_i), \vec{s}(t)$)          ▷ call to source policy
10:     end if
11: end procedure
```

5.4 Specialization and Diversity Maintenance

In order to promote population diversity, a policy's proficiency at interacting with the task environment (Section 5.3) as well as its *novelty* must factor into the selection process, where novelty refers to the degree of similarity between a policy and all other members of the *same* population. The intuition behind explicit diversity maintenance is that searching for *good* things as well as *different kinds* of things has two key benefits: 1) diversity helps prevent premature (team) convergence; and 2) when developing a library of reusable code, a diverse population represents a versatile toolbox for subsequent reuse [75] (See Phase 1, Figure 5.1). As such, two broad approaches to diversity maintenance are assumed in this work: regularization and multiple (source task) populations.

5.4.1 Fitness Regularization

Diversity through regularization implies that the teams are selected for policies that produce high rewards *and* exhibit unique qualities relative to all other members of the *same* population. Rather than assuming a single metric to quantify the similarity of two policies, this work adopts two different metrics and switches between them, selecting either metric with equal probability at each generation. Switching between dissimilar distance metrics: a) avoids introducing yet another scalar weighting parameter; b) has been shown to be as effective as combining metrics in a multi-objective formulation [34]; and, c) actively encourages the development of modularity [67, 110]. Such metrics are also ideally task-agnostic.

A **Program-Utility Distance Metric** is used to characterize diversity across the programs that are 'active' within a team, and as such defines diversity as a group property. A program is considered active if it contributes an action at least once during the life of a team. The distance between teams is summarized as the ratio of active programs common to both teams. Thus, the Program-Utility Distance between teams i and k is

$$dist(tm_i, tm_k) = 1 - \frac{Prog_{active}(tm_i) \cap Prog_{active}(tm_k)}{Prog_{active}(tm_i) \cup Prog_{active}(tm_k)} \tag{5.1}$$

where $Prog_{active}(tm_x)$ represents the set of active programs in team x. There is nothing task-specific in this metric.

A **Behavioural Distance Metric** characterizes a team by how it interacts with the environment. At each decision point in a game, the state and subsequent action taken by the team are recorded. Each state variable is discretized to $[0, 1, 2]$, or low, medium, high. Thus, for each training game a *profile* vector is recorded, $\vec{p} = [\{a(t_s), \vec{s}_d(t_s)\}, t_s \in T]$, where $a(t_s)$ is the atomic action taken, $\vec{s}_d(t_s)$ is the discretized state observation, and T represents every decision point in the associated game. Teams maintain a historical record of the profile $\vec{p}$ for every game played during training. We define $\vec{P}$ to be the concatenation of all profile vectors $\vec{p}$ in a team's historical record. Let $Z(\vec{P})$ be the compressed length[1] (in bytes) of profile vector $\vec{P}$. The behavioural distance between a pair of teams can now be summarized as the Normalized Compression Distance (NCD) [45] between their corresponding profiles:

$$NCD(\vec{P_i}, \vec{P_k}) = \frac{Z(\vec{P_i}\vec{P_k}) - min(Z(\vec{P_i}), Z(\vec{P_k}))}{max(Z(\vec{P_i}), Z(\vec{P_k}))} \tag{5.2}$$

where $Z(\vec{P_i}\vec{P_k})$ is the compressed length of the two profile vectors, $\vec{P_i}$ and $\vec{P_k}$, concatenated. NCD returns a real number between 0 and ≈ 1.2 that characterizes the difference between profile vectors. Equation (5.2) leverages the ability of compression algorithms to filter redundancies in data. For example, if $\vec{P_i}$ and $\vec{P_k}$ are very similar, then $Z(\vec{P_i}\vec{P_k}) \approx Z(\vec{P_i}) \approx Z(\vec{P_k})$, in which case $NCD(\vec{P_i}, \vec{P_k}) \approx 0$. NCD is informative even when comparing vectors that differ in length, which is important because each team's profile will contain a variety of episodes with different outcomes.

Balancing Fitness and Novelty can be achieved in several ways, including fitness sharing [92], linear weighted sum of fitness and novelty [28, 71], or multi-objective optimization [34]. In this work, a simple two-objective approach based on a Pareto dominance relation balances task fitness with novelty as measured using *one* of the above diversity measures. Thus, two objectives are identified:

- $Fit(tm_i)$ is the mean reward over all games in which tm_i has been evaluated.
- $Nov(tm_i)$ is the mean distance, as measured with either diversity metric (Eqn. (5.1) or (5.2)), between tm_i and the $knn = 15$ nearest neighbours in the same

[1] This work uses the bzip2 data compression library for calculating the length (in bytes) of a compressed sequence of integers.

population[2]. For each generation, we choose which diversity metric to assume throughout *that* generation with *equal* probability.

A team tm_i is said to dominate another team tm_j, if tm_i is better that tm_j in at least one objective and no worse in the others.[3] Finally, each team is ranked prior to selection, where we assume that more desirable teams will be dominated by fewer individuals:

$$Rank(tm_i) = 1 - \frac{Dom(tm_i)}{Pop_{size}} \tag{5.3}$$

where $Dom(tm_i)$ is the number of teams in the same population that dominate tm_i.

5.5 Overall Policy Tree Training Algorithm

Algorithm 7 describes the overall training algorithm for the development of policy trees, which applies to both phases of evolution. Parameters are listed in Table 5.1. The procedure begins by initializing $Pop_{size} - Pop_{gap}$ teams. This is followed immediately by the repeated application of variation operators inside the main training loop, creating new team offspring until the team population size reaches Pop_{size} (Lines 5 - 9). Next, all team policies are evaluated in the task environment, where each records the final reward signal and profile vector (Section 5.4.1) for each episode (Line 12). In phase 1, fitness regularization is then applied to rank teams, i.e diversity maintenance (Lines 14 - 16). In phase 2, teams are simply ranked by the mean final reward from all evaluation episodes. From there, policy development is driven by a generational GA such that a fixed number of the least desirable (lowest ranked) teams are deleted in each generation and replaced by the offspring of surviving teams. One implication of this selection process is that the GA is driven by group-level selection, i.e. the team is judged as a whole rather than by the performance of individual components (Section 1.5.1). As such, programs have no individual fitness. At each generation, orphaned programs – those that are no longer a member of any team – are assumed to be ineffective and deleted (Line 21).

[2] $knn = 15$ was selected as a reasonable value based on empirical tuning prior to the experiments described in this work.

[3] Formally, a solution a_i dominates another solution a_j if $\forall k[f_k(a_i) \geq f_k(a_j)] \wedge \exists k[f_k(a_i) > f_k(a_j)]$.

Algorithm 7 The overall Policy Tree training algorithm applied at each phase of evolution. T^t refers to the root-level team population at time t. P^t refers to the program population at time t. t_{max} is the number of generations for this phase. VARIATION is a reference to the team variation procedure detailed in Section 4.2.2. Policies are evaluated in t_{eval} episodes per generation, up to a max of l_{eval} episodes per lifetime. $numEval(tm_i)$ returns the number of evaluations for tm_i so far. When the cost of evaluations is high, these parameters can be set such that weak policies are identified and replaced early, while promising policies are verified with additional evaluations. Note that the program population is updated and modified implicitly through team variation operators (Line 8). When teams are deleted in Line 20, programs with no remaining team membership are also deleted (Line 21). *task* refers to the task environment.

```
 1: procedure TRAIN
 2:     $t = 0$
 3:     initialize $Pop_{size} - Pop_{gap}$ teams, add to $T^t$ (add new programs to $P^t$)
 4:     while $t \leq t_{max}$ do
 5:         while $T^t$ contains less than $Pop_{size}$ teams do                ▷ Variation
 6:             uniformly sample parent team $tm_i \in T^t$
 7:             copy $tm_i$ into $tm_j$
 8:             $tm'_j \leftarrow$ VARIATION($tm_j$)
 9:             add $tm'_j$ to $T^t$
10:         end while
11:         for all $tm_i \in T^t : numEval(tm_i) < l_{eval}$ do                ▷ Evaluation
12:             deploy $tm_i$ for $t_{eval}$ evaluations in $task$ (see Algorithm 6)
13:         end for
14:         if $phase == 1$ then                                  ▷ Diversity maintenance
15:             select either diversity metric with equal probability
16:             apply Pareto ranking to $T^t$ (see Section 5.4.1)
17:         else
18:             rank teams in $T^t$ by their mean episode outcome
19:         end if
20:         delete $Pop_{gap}$ lowest ranked teams from $T^t$                    ▷ Selection
21:         delete from $P^t$ programs that are not part of any team
22:         $t = t + 1$
23:     end while
24: end procedure
```

Another implication of this selection procedure relates to the exploration/exploitation trade-off common to search and optimization algorithms. That is, a balance needs to be struck between the resource spent exploring the search space versus resource spent exploiting solutions found so far [153]. In Algorithm 7 this balance is controlled by the Pop_{gap} parameter, which specifies the proportion of the team population (Pop_{size}) that is deleted in each generation to make room for new offspring. If Pop_{gap} is high, then the balance is shifted toward exploration, since more resource in each generation is spent producing offspring through application of the variation operators. On the other hand, the GA used here is elitist in that the least desirable teams are deterministically deleted (Line 20), i.e. a team is only deleted after variation operators have discovered something more desirable (from the perspective the cost function). As such, with the best performing teams protected, a relatively high Pop_{gap} is adopted in this work on the assumption that exploration leads to diversity, which is essential to constructing meaningful hierarchies.

Table 5.1: Parameters associated with the overall Policy Tree training procedure, Algorithm 7, in addition to parameters outlined in Chapter 4

Parameter	Description
Pop_{size}	Team population size
Pop_{gap}	Number of teams deleted and introduced in each generation
t_{max}	Number of generations
t_{eval}	Number of evaluation episodes per team in each generation
l_{eval}	Maximum number of evaluation episodes per team

5.6 Multiple Populations and Transfer Learning

Relative to previous work with SBB in RL tasks [35, 75, 92, 70, 71], the algorithm described in this chapter explicitly addresses how the bottom-up, incremental development of a policy tree provides natural opportunities for task transfer since each level of the hierarchy can be developed relative to different components of the task, e.g. different environments or objectives. Specifically, phase 1 identifies a diverse set of relatively simple ‘source’ policies while phase 2 learns how to reuse phase 1 source policies for solving a different, more complex target task. A user may have multiple candidate source tasks in mind, but no way of a priori prioritizing or selecting which source tasks to employ in practice. In this work it is assumed that all candidate source tasks will be developed in parallel during independent instances of phase-1 evolution. Note that in this case the parallel, multi-population model serves two purposes:

1. Parallel development of multiple source task policies.

2. Additional diversity maintenance since isolated populations are less likely to converge to similar solutions (Section 2.3).

Phase 2 will then identify which specific source policies to deploy and how. The complete procedure for Policy Tree development/training is depicted in Figure 5.2. The case of ‘Transfer’ implies that multiple different source tasks are learned in phase 1, where the alternative is to simply assume the target task throughout, or ‘Baseline’ in Figure 5.2. In effect, the Transfer case requires prior information to inform the identification of code for reuse. Conversely, the Baseline scenario assumes that diversity maintenance is sufficient for this purpose. The complete process is summarized as follows assuming two source tasks have been identified:

1. Phase 1: Coevolve N_1 independent, non-hierarchical team-program populations for source tasks A and B in parallel (Algorithm 7). The action set available to programs is the set of atomic actions from the (source) task environments.

2. Combine the independent team-program populations from phase 1 into a master set of policies for each source task, A and B.

3. Rank the policies in sets A and B based on mean training reward (i.e. no diversity regularization) and delete all but the *PopGap* best individuals from each set.

4. Combine sets A and B into a single pool of source policies. This pool now represents the library of source policies for reuse during phase 2. Source policies are not modified any further.

5. Phase 2: Coevolve N_2 independent, root team-program populations (i.e. policy trees) under the target task environment (Algorithm 7). The library of phase-1 source policies represents the action set available to phase-2 programs.

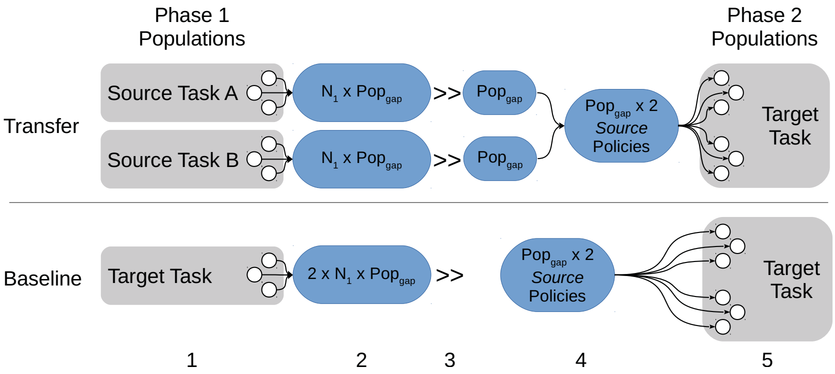


Figure 5.2: Selecting 'source' policies from Phase 1 of evolution for recombination at Phase 2. N_1 is the number of independent team-program population pairs coevolved in Phase 1 for each source task. Pop_{gap} is the final number of policies produced from each population. $\gg$ symbolizes the filtering process in which policies are ranked and selected based on average training reward. In the empirical evaluation of Transfer (Chapter 6), Phase 1 is parameterized such that Transfer and Baseline cases are allocated equivalent computational resources.

Note that in phase 2, the root team is essentially a switching policy that learns under what environmental situations to reuse policies from phase 1. No information is provided regarding which source policies were developed under which source task, nor any information regarding their quality or specialization. There is also no prior state or action mapping function used to define how the meaning of state variables and/or actions changes between source and target task. Thus, the nature of code reuse in phase 2 is developed entirely through environmental interaction.

5.7 Additional Domain-Specific Details

In RoboCup Soccer (Section 3.2), the process for a policy to map a state observation to a task-specific atomic action in each time step follows the generic procedure in Algorithm 6. This applies to the ego-centric state observation and task-specific macro actions available to the player currently in possession of the ball, while other players assume prior macro actions as per the decision tree in Figure 3.2. However, in Ms. Pac-Man the policy needs to evaluate each direction separately *then* decide in which direction to move (Section 3.4). Thus, decision-making is now a two-part process. First, the policy (tree) is used to map the state observation from each legal direction to one of two atomic actions in $\{Accept, Reject\}$. The winning bid value for each decision is also recorded. The policy then moves in the accepted direction with the highest bid. If no direction is accepted, the policy moves in the rejected direction with the lowest bid. In other words, the policy either moves in the direction that is most-accepted, or least-rejected. The process for a policy to select a move relative to each direction at any given time step is detailed in Algorithm 8.

Algorithm 8 Procedure to select a move for Ms. Pac-Man, given a team policy tm_i. D^L is the set of legal directions (not leading into a wall from the current location). Programs in source policies (Figure 5.1) assume one of two atomic actions from the action set $\mathcal{A} = \{Accept, Reject\}$. To select a move, the policy considers each legal direction in isolation, selecting an atomic action for each direction $d_i \in D^L$ (Lines 2 - 5). Note that the atomic action *and* the wining program bid are saved for each direction as $act(d_i)$ and $bid(d_i)$ respectively. The policy then moves in the accepted direction with the highest bid value (Line 7). If no directions are accepted, the policy moves in the rejected direction with the lowest bid (Line 9).

```
 1: procedure SelectMove($tm_i, D^L$)
 2:     for all $d_i \in D^L$ do
 3:         $\vec{s}(t) \leftarrow$ state variables for direction $d_i$
 4:         $\{act(d_i), bid(d_i)\} = \text{SelectAction}(tm_i, \vec{s}(t))$        ▷ Algorithm 6
 5:     end for
 6:     if $\exists d_i \in D^L : act(d_i) == Accept$ then
 7:         $Move \leftarrow \arg_{d_i \in D^L} \max(bid(d_i) : act(d_i) == Accept)$
 8:     else
 9:         $Move \leftarrow \arg_{d_i \in D^L} \min(bid(d_i) : act(d_i) == Reject)$
10:     end if
11:     return $Move$
12: end procedure
```

Chapter 6

Empirical Evaluation: Policy Trees

6.1 Overview

This chapter describes experiments carried out with the Policy Tree algorithm (Chapter 5) in the RoboCup and Ms Pac-Man task domain. The purpose of these experiments is to demonstrate how Policy Trees support transfer learning as a means of scaling to difficult task domains. As brief review, policy trees are constructed hierarchically over independent phases of evolution. First, a diverse group of lower-level source policies are evolved for related but *different* source task(s). A second phase of evolution is conducted to learn how to reuse the source policies under the target task by evolving a higher-level *switching policy.* Such a process enables policies from multiple source tasks to be recombined to solve the overall target task. Moreover, source tasks may be 'automatically' discovered through mechanisms such as behavioural diversity / novelty [85, 34].

In this work, the utility of supporting transfer learning through code reuse is initially demonstrated under the challenging multi-agent soccer domain of Half Field Offense (HFO). In Phase 1, policies are evolved under two independent source tasks: Keepaway (which evolves policies for retaining possession of the ball) and Goal Scoring. Phase 2 evolves a policy for the Half-Field Offense (HFO) target task by learning how to reuse a subset of the previously evolved behaviours, or a *switching policy* (Chapter 3 provides detailed source and target task descriptions). HFO is much closer to the full-scale RoboCup soccer task and requires one team to defend their goal and the other team to score (the offense team has no goal or goalie). The objective, state, and action spaces are not the same for source and target tasks, a further criterion for meaningful task transfer [143, 142]. Furthermore, the agent's sensors and actuators are noisy, making the problem highly stochastic and partially observable. The final policy tree achieves an equivalent level of performance as the current RL state-of-the-art, but at a fraction of the model complexity. Next, the identical Policy Tree framework is applied to the popular Ms. Pac-Man video game, where most previous research, including the current state-of-the-art learning algorithm [117], relies on a similar prior task decomposition into multiple source tasks (Chapter 3 provides details on the source and target tasks in Ms. Pac-Man). We demonstrate that hierarchical code reuse with diversity maintenance alone is sufficient to exceed the best results published to date. Additional decomposition through prior specification of source tasks improves on this. Furthermore, GP solutions are again shown to be simpler and more efficient than previous champions developed by neuro-evolution.

In summary, the scope of this chapter is to demonstrate the ability to learn high-quality behavioural agents for challenging game environments with minimal prior knowledge. The use of Policy Trees in constructing behaviours through task transfer implies that source task selection can be less constrained or even automated (see results under Ms. Pac-Man, Section 6.4), and multiple source tasks can be considered. As the policy tree evolves, evolution determines which source policies are of merit and which are not. Ultimately, source policies might have been originally developed from the perspective of say, goal scoring, but utilized for maintaining possession (this actually transpired in practice, see Section 6.6). Thus, pursuing the proposed approach to task transfer enables the discovery of effective/novel policies that would otherwise never have been possible.

Animations under both RoboCup and Ms. Pac-Man, which illustrate how the modular GP strategies support decision-making during gameplay, are included with this thesis (See files `PolicyTree-HFO.mp4` and `PolicyTree-MsPacMan.mp4`).

6.2 Experimental Setup

A representative comparator algorithm establishes a comparative measure of performance for each task domain as follows:

- **SarsaRBF**: The Sarsa Temporal Difference Method with Radial Basis Functions. SarsaRBF represents the current state-of-the-art in the HFO task [65].

- **MM-NEAT**: Modular Multi-objective NEAT with 2 modules, representing the current state-of-the-art in the Ms. Pac-Man task [117].

Table 6.1: Algorithms benchmarked in Section 6.2. SarsaRBF and HAND are comparison algorithms for HFO soccer. MM-NEAT is a comparison algorithm for Ms. Pac-Man. Shaded rows indicate experiments conducted only in HFO soccer, all other SBB treatments are tested in both domains. Used by permission, ©2017 IEEE.

Label	Description
SBB.TD	SBB with transfer and switching diversity maintenance
SBB.T	SBB with transfer, no diversity maintenance
SBB.D	SBB no transfer, with switching diversity maintenance
SBB.D.B	SBB no transfer, Behavioural diversity maintenance only
SBB.D.P	SBB no transfer, Prog-Utility diversity maintenance only
SBB	SBB no transfer, no diversity maintenance
SarsaRBF	Sarsa with Radial Basis Function Approximator
HAND	Hand-crafted HFO policy
MM-NEAT	Modular Multi-objective NEAT with 2 Modules

In both cases we assume the parameterization/source code from the original implementations.

Recall that the Policy Tree algorithm described in this thesis is an extension of the Symbiotic Bid-Based (SBB) approach to GP [88]. In naming the various extensions explored in this chapter, we refer to the original algorithm for building policy trees as simply SBB, while the addition of transfer learning is denoted by a ".T" extension, and the addition of diversity maintenance (as described in Section 5.4) is denoted with a ".D" extension[1]. This work establishes the utility of transfer learning and diversity maintenance independently and in combination. Table 6.1 summarizes all experimental cases. See Figure 5.2 for a illustration of the SBB model with and without transfer, and Section 5.4 for a detailed description of the approach to diversity maintenance proposed. Note that when no diversity is enforced, teams are selected based on training reward alone, or $Fit(h_i)$ in Section 5.4 (Balancing Fitness and Novelty).

Table 6.2 summarizes the parameterization assumed for SBB.T. Values specific to Ms. Pac-Man are in parentheses, while all other parameters are common to both domains. The maximum number of evaluations (e_{max}) is a function of the population size, number of generations, and number of evaluations for each individual per

[1] The results presented in this chapter are published in [74]. The naming scheme for policy tree variants in this thesis is consistent with that article.

Table 6.2: Parameterization of team and program populations. p_{mx} denotes a mutation operator in which: $x \in \{d, a\}$ are the prob. of deleting or adding a program respectively; $x \in \{m, n\}$ are the prob. of creating a new program or changing the program action respectively. e_{max} is the maximum number of evaluations over all populations (See Figure 5.2) per phase. Parameters for Ms. Pac-Man are in parentheses. Used by permission, ©2017 IEEE.

Team (GA) population			
Parameter	Value	Parameter	Value
N_1	10	N_2	20 (10)
e_{max} Phase 1	84,150 (90,900)	T	50 (3)
e_{max} Phase 2	56,700 (45,450)	e_{test}	1000 (100)
Pop_{size}	180	$Mgap$	50%
p_{md}, p_{ma}	0.7	ω/Ω	30
p_{mm}	0.2	p_{mn}	0.1
Program population			
numRegisters	8	*maxProgSize*	96
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

generation. These parameters can be modified based on the nature of the task. For example, the Goal Scoring source task was easier to learn than the Keepaway source task, and thus required fewer generations. The number of generations and number of evaluations for each individual per generation $(t_{eval})^2$ for a single phase in each task were as follows:

- Scoring: 60 generations, $t_{eval} = 25$
- Keepaway: 125 generations, $t_{eval} = 10$
- HFO: 125 generations, $t_{eval} = 10$
- All Ms. Pac-Man tasks: 100 generations, $t_{eval} = 10$

In order to keep the maximum number of evaluations constant over all experiments, SBB assumes the same parameterization as SBB.T *except* in the case of the Phase 1 generation limit. When multiple source tasks are not employed at phase 1 (Baseline in Figure 5.2), the generation limit is increased such that e_{max} for phase 1 is equivalent across all experiments in the respective task domain. In HFO, SarsaRBF

[2] Each individual was only evaluated in a single generation, thus $l_{eval} = t_{eval}$ (Table 5.1).

learning curves appeared to plateau after roughly 20,000 episodes, at which point training was stopped and policies saved for evaluation under test conditions.[3] MM-NEAT evolved a population of 100 individuals with 10 evaluations each over 200 generations, for a total of $100 \times 10 \times 200 = 200,000$ evaluations.

Due to the abundant sources of noise in RoboCup[4] and Ms. Pac-Man[5], post-training test games (e_{test}, Table 6.2) are required to provide an accurate measure of performance for the single champion policy from each run. However, the computational cost of evaluations precludes testing an entire population over a large number of games in order to identify such a champion. Thus, the best policy in any population is identified by first identifying the single policy with the highest *training* reward in each of the final T generations. If the policy with highest training reward has already been saved, the next-best policy is identified. These T unique cached policies are then tested in e_{test} games, and the highest scoring policy represents the population champion assumed for test.

6.3 Half Field Offense Test Performance

Figure 6.1 reports test performance for the four SBB configurations under the HFO task (first 6 rows of Table 6.1), along with SarsaRBF and the hand-crafted policy. It is apparent that both SarsaRBF and SBB.T reach similar levels of performance. Indeed, there is no statistical difference between SarsaRBF and either of the SBB.T distributions in Figure 6.1 (Mann-Whitney rank test, $P < 0.05$), as indicated by the Grey bracket across the top of the figure grouping these distributions. Diversity maintenance does not provide a significant benefit when task-transfer is used in the HFO task, nor is it harmful. The fact that diversity does not make a significant difference here is likely a further testament to the utility of knowledge transfer. Source policy reuse provides enough leverage to overshadow the contribution of diversity maintenance, with all SBB.T cases reaching state-of-the-art performance. Indeed, neither SBB case without transfer is able to reach a competitive level with SarsaRBF (SBB and SBB.D in Figure 6.1). However, diversity maintenance becomes important

[3] Under test, SarsaRBF selects actions strictly according to its value function (see [160]).

[4] Sensor and actuator noise, as well as random start conditions for takers, keepers, and ball in each game.

[5] Stochasticity in ghost behaviours and path-finding algorithms from which sensors are derived.

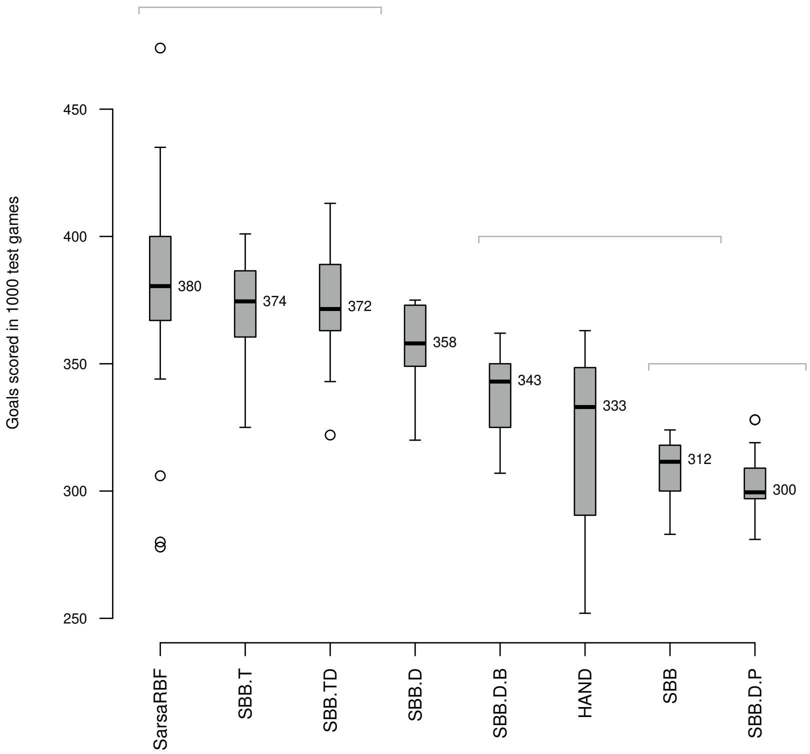


Figure 6.1: Post-training test results over 1000 games. SarsaRBF represents a current state-of-the-art reinforcement learner for the HFO task; SBB denotes hierarchical SBB cases as outlined in Table 6.1. HAND represents a third party hand-crafted (i.e. human-designed) HFO policy native to the HFO environment. Box plots summarize the quartile distribution over 20 independent runs. Grey bracket groups distributions with no statistically significant difference, $P > 0.05$ from Mann-Whitney rank test. Used by permission, ©2017 IEEE.

in the absence of task transfer, where only the SBB case with diversity maintenance (SBB.D) was able to outperform the hand-crafted policy. Finally, in order to test the utility of stochastically switching distance metrics within the diversity mechanism (Section 5.4), we performed one run with strictly behavioural diversity (SBB.D.B) and one run with strictly Program-Utility diversity (SBB.D.P). While standalone Behavioural Distance worked better than Program-Utility, stochastically switching between metrics proved important, as neither metric worked well enough alone to outperform the hand-crafted baseline.

6.4 Ms. Pac-Man Test Performance

Several constraints are placed on the Ms. Pac-Man environment in order to minimize the computational cost of evaluations during training. As such, Ms. Pac-Man is limited to a single life, a single visit to each of 4 mazes, and a maximum of 8000 time steps to complete each maze. A game ends when any of these limits expire. However, a much less-constrained version of Ms. Pac-Man is used to test the quality of learned behaviours post training. This version closely matches that used in the Ms. Pac-Man versus Ghosts competition (MPMvsG). Additional rules under the MPMvsG task are: 1) Ms. Pac-Man starts with 3 lives and gains an extra life after earning 10,000 points; 2) Completing the 4th maze returns Ms. Pac-Man to the first maze until each maze is visited 4 times, for a total of 16 levels; 3) the per-level time limit is 3000 time steps, but rather than being killed when time runs out, she receives half the score from the remaining pills and advances to the next level; and 4) Ms. Pac-Man has a time limit of 40ms to return an action in each time step. If an action is not returned in time, the action from the previous time step is assumed.

In keeping with the Ms. Pac-Man literature, we report the mean and max champion test scores for the four SBB configurations (first 4 rows of Table 6.1) under the MPMvsG task along with MM-NEAT, Figures 6.2 and 6.3 respectively. There is no statistical difference between the mean score for any of the SBB configurations (See the Grey bracket extending over the first 4 columns of Figure 6.2). However, both SBB cases *with* diversity maintenance significantly outperform MM-NEAT.[6] The fact that SBB manages to exceed the performance of MM-NEAT even without task transfer implies that prior task decomposition, or the separation of pill score and ghost score employed under SBB.T and in the multi-objective component of MM-NEAT, is unnecessary. However, diversity maintenance *is* critical for effective code reuse. That said, the combined task-transfer and switched diversity maintenance scheme is still the most consistent performing model.

The maximum scores for all SBB cases are significantly better than MM-NEAT, Figure 6.3. Again, diversity maintenance appears to be more important than task

[6] The lower Grey bracket in Figure 6.2 indicates no statistical difference between MM-NEAT and SBB treatments *without* diversity maintenance, but does not include SBB cases *with* diversity.

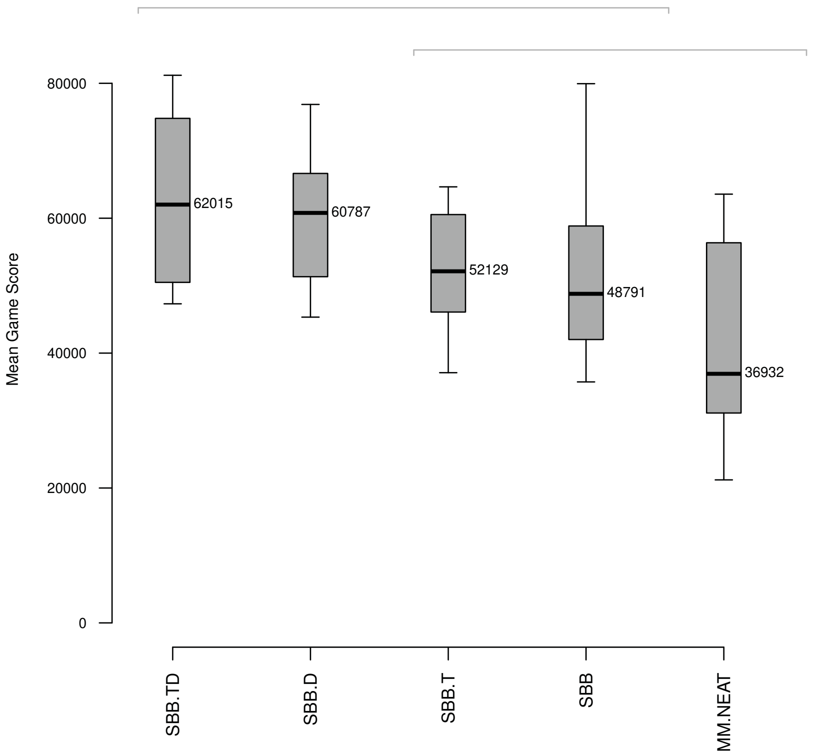


Figure 6.2: Mean post-training test scores over 100 games in the MPMvsG task. MM-NEAT represents a current state-of-the-art learner for the Ms. Pac-Man task; SBB denotes hierarchical SBB cases as outlined in Table 6.1. Box plots summarize the quartile distribution over 10 independent runs. Grey bracket groups distributions with no statistically significant difference, $P > 0.05$ from Mann-Whitney rank test. Used by permission, ©2017 IEEE.

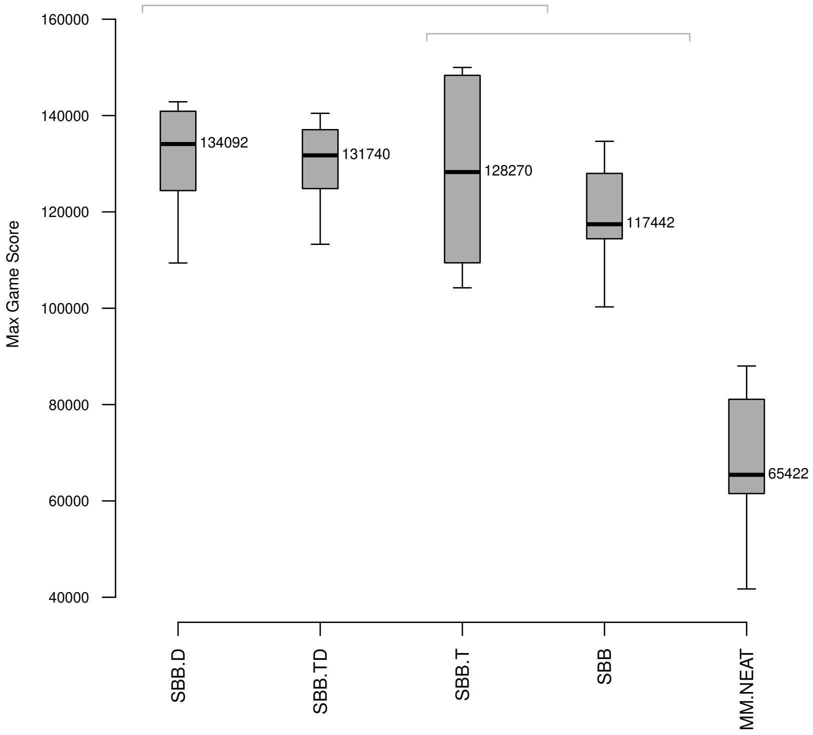


Figure 6.3: Max post-training test scores over 100 games in the MPMvsG task. MM-NEAT represents a current state-of-the-art learner for the Ms. Pac-Man task; SBB denotes hierarchical SBB cases as outlined in Table 6.1. Box plots summarize the quartile distribution over 10 independent runs. Grey bracket groups distributions with no statistically significant difference, $P > 0.05$ from Mann-Whitney rank test. Used by permission, ©2017 IEEE.

transfer in this domain, as only the cases with diversity (SBB.D and SBB.TD) produce better results than the SBB baseline. Note that the SBB.TD and SBB.D results are also competitive with policy discovery through Monte Carlo Tree Search (MCTS), i.e. a search process that requires considerably more task specific information than any formulation of SBB. Specifically, the corresponding average and max. post training performance for MCTS are 107,561 and 127,945 respectively [111]. Attempting to use GP to identify heuristics to guide MCTS resulted in average and best performance metrics of 32,641 and 62,630 [4], which is significantly lower than any SBB configuration.

6.5 Significance of Policy Tree Variants

There are a total of four Policy Tree configurations considered across both HFO and Ms. Pac-Man task domains (SBB.TD, SBB.D, SBB.T, SBB). The Friedman rank based non-parametric test may be employed to summarize the relative significance of the configurations [30]. Table 6.3 summarizes the initial ranks. The Friedman statistic for $k = 4$, $N = 2$ is $\chi^2 = 5.4$. Re-normalizing for the F-distribution provides $F_F = 9$. This is greater than the corresponding critical value of $F(3, 3) = 5.391$ at a p-value of 0.1; hence we are able to reject the null-hypothesis. Finally, applying the Nemenyi post hoc test defines the performance of any two classifiers as being significantly different if the average ranks differ by a critical difference (CD) defined as $q_\alpha(\frac{k(k+1)}{6N})^{\frac{1}{2}} = 2.96$ for $q_{0.1} = 2.291$. Thus, Policy Trees configured without transfer or diversity maintenance is consistently inferior to including both diversity and transfer.

Table 6.3: Rank based summary of SBB.TD, SBB.D, SBB.T, SBB Policy Tree configurations over each task. Used by permission, ©2017 IEEE.

Domain	SBB.TD	SBB.D	SBB.T	SBB
HFO	1	3	2	4
Ms.Pac-Man	1	2	3	4
Avg. Rank	1	2.5	2.5	4

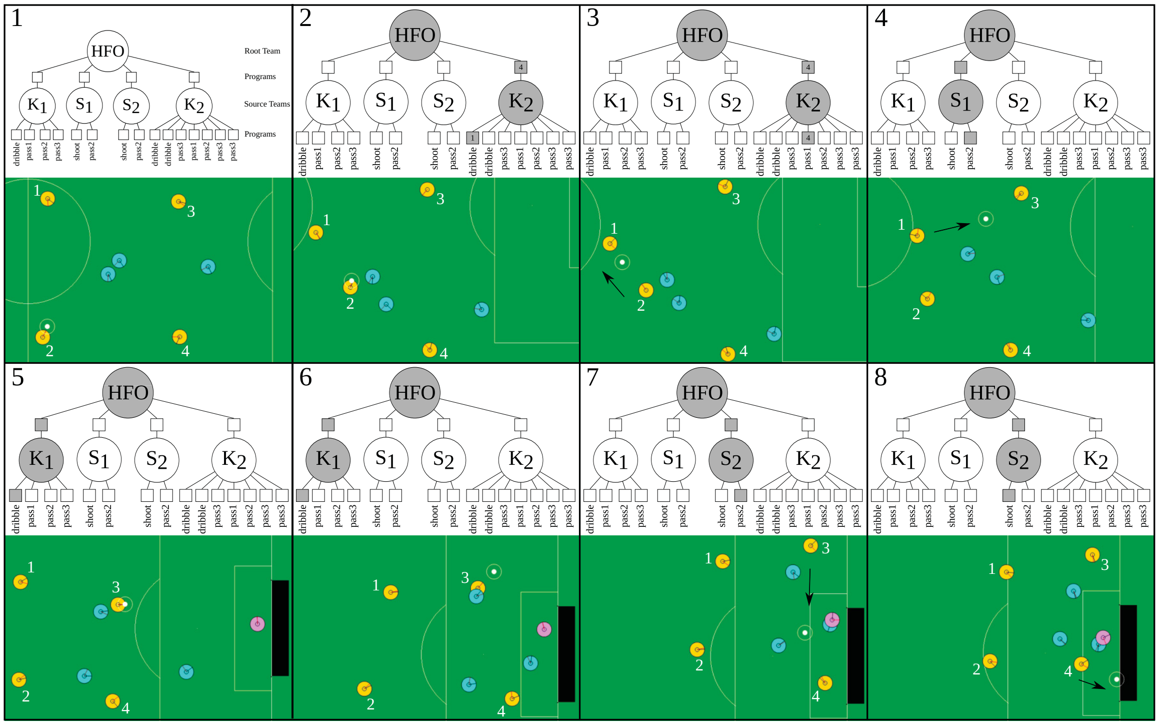


Figure 6.4: Step-by-step HFO game played by a champion policy tree from the SBB.TD experiment. This policy was able to win the game after only 7 'moves'. The policy controls yellow players (offense team) only. Opponents (defense team) are blue, and goalie is pink. Used by permission, ©2017 IEEE.

6.6 HFO Solution Analysis

In order to provide some insight into the behaviour of policy trees in the HFO task, we can analyze a single champion policy tree from the SBB.TD experiment, Figure 6.4. This individual scored in 394 of 1000 test games. The policy tree has 4 active programs at the root of the policy tree, 2 of which index Keepaway source policies (K_1, K_2) and 2 index Scoring source policies (S_1, S_2). There are a total of 15 active programs across all source policies, covering the full scope of domain-specific atomic actions. We evolved homogeneous teams, thus any offense player in possession of the ball is controlled by the same policy tree.

To understand the behaviour of this policy tree, we can trace the sequence of source policies and atomic actions deployed at each decision point during a specific HFO game, as shown in Figure 6.4. This particular game proceeded as per the following play-by-play, in which the components of the policy tree are explicitly annotated:

1. The game is initialized with offense player 2 in possession of the ball.

2. Program 4 has the highest bid in the top level HFO switching policy and thus deploys source policy K_2, where program 1 outbids the others to deploy the dribble atomic action.

3. Program 4 is again selected by the HFO switching policy, deploying K_2, where program 4 now has the highest bid. Offense player 2 thus passes to its closest teammate (pass1).

4. Player 1 immediately passes to its second closest teammate, this time using Scoring source policy S_1.

5. Player 3 dribbles toward the goal using K_1.

6. Player 3 loses control due to actuator noise, and is forced to chase the ball up to the top right corner of the field (with an opponent in close pursuit).

7. Player 3 passes down to player 4 (i.e., the second closest teammate).

8. Player 4 shoots and scores.

Clearly, this HFO policy interleaves different Keepaway and Scoring source policies while approaching the goal and achieving the winning shot. The high-level task decomposition does not follow the simplistic division of labour (ball handling and scoring) suggested by the source task configurations. Interestingly, at the lower level, K_1 and K_2 developed different contexts for the dribble atomic action, both of which proved useful. Similarly, S_1 and S_2 developed different contexts for the pass2 atomic action. Indeed, all programs in this policy produce a winning bid at least once over 25 games. An animation of this policy can be viewed in the file `PolicyTree-HFO.mp4` included with this thesis.

6.7 Ms. Pac-Man Solution Analysis

Figure 6.5 depicts a policy tree from the SBB.TD experimental case, which yielded the highest median score in the MPMvsG task (Figure 6.2). This policy includes 4 programs as part of the root team, 2 of which index source teams trained relative to the pill score objective (P_0 and P_1) while the other 2 index source teams trained on the ghost score objective (G_0 and G_1).

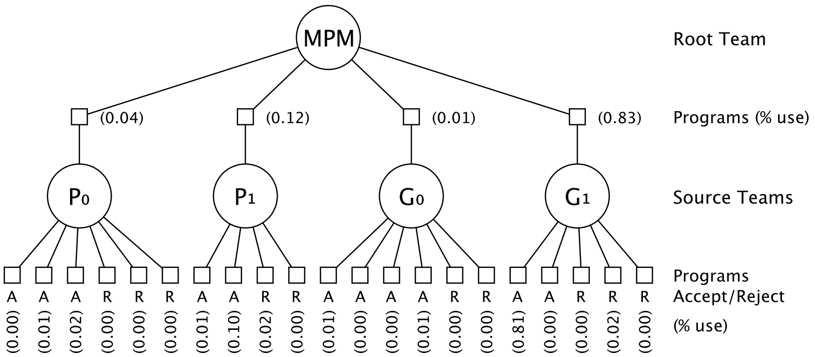


Figure 6.5: Example policy tree from the SBB.TD (Ms. Pac-Man) experimental case, which yielded the highest median scores in the MPMvsG task (See Figure 6.2). (% use) is the fraction of all time steps for which each program produced a winning bid under test. Used by permission, ©2017 IEEE.

The process for an SBB policy to follow a root-to-leaf path and ultimately select an action for Ms. Pac-Man is the same as under HFO (Section 6.6). However, the domain-specific atomic actions are different and the team needs to evaluate each direction separately *then* make a decision [117]. Thus, programs in a team collectively decide to accept or reject each direction, where each decision also includes a bid value. The policy then moves in the accepted direction with the highest bid. If no direction is accepted, the policy moves in the rejected direction with the lowest bid. In other words, the policy either moves in the direction that is most-accepted, or least-rejected. The process for a policy tree to select a move relative to the legal directions at any given time step is detailed in Chapter 5, Algorithm 8.

All source teams in the policy of Figure 6.5 have roughly a 50/50 mix of programs with the Accept and Reject atomic action. The frequency of use for each program, relative to a single game of MPMvsG, is shown in parentheses. Clearly, this policy favours source policy G_1 (used in 83% of decisions) while the other source teams are switched in to achieve specific behaviours. Interestingly, P_0 is often used when eating power pills and P_1 is often used at the moment ghosts are eaten. Both these behaviours are intuitively associated with the ghost objective, rather than the pill objective under which these source teams were developed. The highest degree of switching occurs in two specific circumstances: 1) when Ms. Pac-Man is oscillating back and forth to remain in the same position[7] while luring threat ghosts; and 2) when Ms. Pac-Man is in a dangerous situation, for example, with threat ghosts approaching from multiple directions. Frequent source team switching indicates a high degree of interaction and cooperation among source teams, which *collectively* achieve luring and escape behaviours for Ms. Pac-Man in this case. An animation of this policy playing the MPMvsG game is included with this thesis in the file `PolicyTree-MsPacMan.mp4`, in which Ms. Pac-Man survives to level 15 (of 16) and achieves a score of 100,480.

6.8 Comparison of Policy Tree Solution Complexity with SarsaRBF and MM-NEAT

All SBB policy trees are teams of programs working cooperatively within a hierarchical policy tree. The number of programs per team and specific complement of programs forming a team are all identified during evolution. Programs are simple linear register machines in which each instruction consists of a single arithmetic operator, function, or conditional statement (See Chapters 4 and 5). Thus, we can describe the complexity of a solution by counting the total number of program instructions over all programs referenced by a policy tree.

The policy tree in Figure 6.4 has a total of 19 programs (4 at the top level and 15 in the lower level), with a total of 880 instructions across all programs. However, the policy tree only follows one path from root to leaf node in order to suggest an action relative to the current state observation, which can be clearly seen in the policy animation (See `PolicyTree-HFO.mp4`). Indeed, while there was a median of 414 instructions in each SBB.TD policy for HFO, an average of only 116 instructions were executed in each time step during test games. In contrast, each SarsaRBF policy contains $16,320$ weight / RBF pairs, all of which require execution at *every* time step.

MM-NEAT solutions under Ms. Pac-Man assume a neural network representation with an average of ≈ 47 nodes and ≈ 90 links in the champion networks. Each node computes the sum of products over some portion of the links, followed by the tanh activation function[8]. If we assume 5 operations for the activation function at

[7] Ms. Pac-Man is required to move in every time step, thus oscillating back and forth is the only way to remain in the same maze location over multiple time steps.

[8] $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

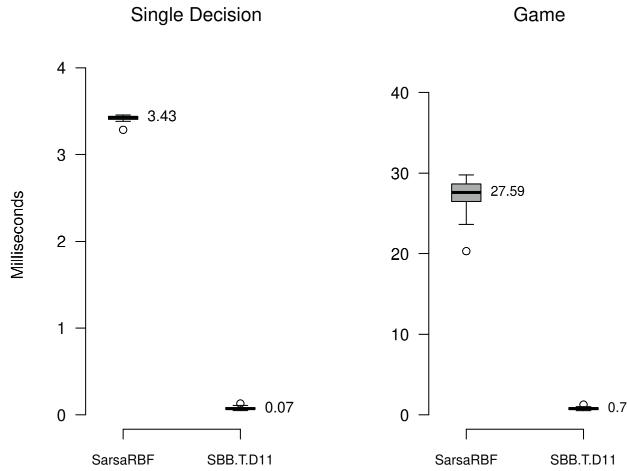


Figure 6.6: CPU time for the champion/final policy to select an action in a single time-step (Single Decision) and over an entire game of HFO (Game), averaged from 1000 test games. Box plots give the distribution over 20 independent runs for each algorithm. The nature of the RoboCup socket interface, as well as the fact that both SBB and SarsaRBF are programmed in c++, makes this direct comparison possible. A similar empirical comparison with MM-NEAT is not possible as MM-NEAT assumes a Java code base. Used by permission, ©2017 IEEE.

each node, and a total of 2 operations per link for the sum of products, then the number of operations at each time step is $links \times 2 + nodes \times 5 = 415$. In short, SBB policies are more efficient to implement post training, with the modular nature of the policy tree providing the basis for this simplicity. Figure 6.6 places this into context by reporting the average CPU time required for SarsaRBF and SBB.TD to make a decision at each time-step over 1000 test games in the HFO environment. Indeed, decision-making under SBB is several orders of magnitude faster.

6.9 Summary

The SBB framework for evolving policy trees has been extended to facilitate the utilization of task transfer, which represents a methodology for taking policies identified under source tasks and then learning how to reuse them to provide policies under a more difficult target task. Two task domains were considered. In the first case, the Half Field Offense subtask of RoboCup soccer was used as the target task, where HFO represents a benchmark known to be more difficult than the Keepaway task previously employed under task transfer (e.g. [146, 154]). SBB as originally formulated was used as a control, and was not able to solve this task directly. The methodology for task transfer under SBB proposed the following: 1) use multiple source tasks; and 2) evolve source task policies with diversity maintenance. Providing multiple source tasks enables the user to cover a ‘range’ of potentially useful starting points to construct solutions to the target task policy. The policy trees that were identified using task transfer achieved HFO performance competitive with the current state-of-the-art SarsaRBF, and were significantly simpler, resulting in execution times several orders of magnitude faster. This can be particularly important in real-time environments as more computational resources can be dedicated to other issues, such as strategic decision making, which are typically given a lower computational priority.

The second empirical study, in which we consider the game of Ms. Pac-Man, reinforces the observations made under RoboCup. Previous learning agents in Ms. Pac-Man have relied on prior knowledge to achieve state-of-the-art performance. However, the process for incrementally constructing policy trees adopted in this work is able to discover reusable code both with and without support for source tasks identified with prior information. Indeed, results are competitive with schemes that assume much more a priori information, such as Monte Carlo Tree Search. Finally, defining solutions in the form of policy trees provides a very efficient scheme for post training deployment.

From the perspective of constructing behavioural agents in general, the SBB Policy Tree framework enables a designer to incorporate prior biases through task transfer, while simultaneously supporting diversity maintenance. Task transfer may make the difference between discovering effective agent behaviours or not (with or without diversity). Diversity maintenance is never detrimental in the experiments herein, and is a must when no prior information is available for task transfer. Moreover, the diversity mechanisms adopted here are task-agnostic, thus independent of the particular application.

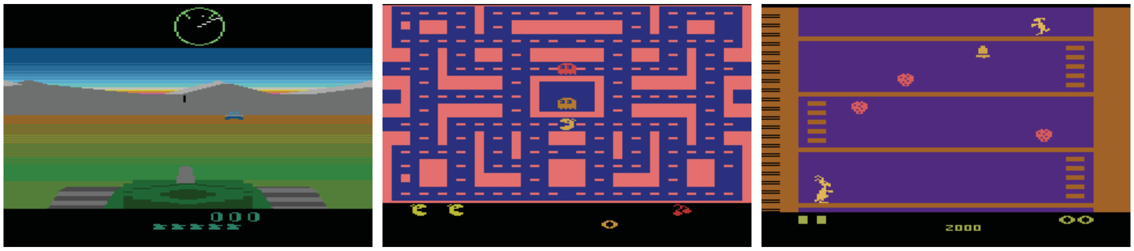
Chapter 7

Domain Description: Arcade Learning Environment

7.1 Overview

Released in 1977, the Atari 2600 has been a popular home video game console that was capable of running a large variety of games, each stored on interchangeable ROM cartridges. Hundreds of games were compatible with the console, bringing the diversity of an Arcade experience into the home through a single device. As each game is designed to be unique and challenging for human players, the Atari 2600 provides an interesting test domain for general artificial decision-making agents, Figure 7.1.

The Arcade Learning Environment or ALE [14] is an Atari 2600 video game emulator designed specifically to benchmark RL algorithms. The ALE allows RL agents to interact with hundreds of classic video games using the same interface as experienced by human players. That is, an RL agent is limited to interacting with the game using state, $\vec{s}(t)$, as defined by the game screen, and 18 discrete (atomic) actions, i.e. the set of Atari console joystick directions including 'no action', in combination with / without the fire button. Each game screen is defined by a 210×160 pixel matrix with 128 potential colours per pixel, refreshed at a frame rate of 60 Hz. Thus, in the most general case, ALE represents a high-dimensional *visual* reinforcement learning



(a) (b) (c)

Figure 7.1: Example Atari game environments. A diverse set of tasks are available, including first-person shooter games (a), maze tasks (b), adventure games (c), and many more.

challenge. While various methods have been proposed for hand-crafting sensory representations from the raw Atari screen frames, including game-specific background and object detection [14, 49, 86], the focus of this work is learning from high-dimensional, task-independent sensory representation. However, in practice the raw screen frames are preprocessed prior to being presented to an RL agent (See Section 7.2.1 for a summary of approaches assumed to date, and Section 9.3 for the specific approach assumed in this work).

Agent-environment interactions within the ALE are episodic. Each episode begins with the agent in a start state defined by the particular game title and continues over a sequence of discrete time steps until a 'game over' signal is received. Time steps are analogous to the 60Hz frame rate. In each step (frame) the agent observes the state of the game, $\vec{s}(t)$, and selects an action $a \in \mathcal{A}$. The 'game over' signal is accompanied by a final reward, or game score, which reflects the quality of the agent's decision-making during that episode.

Interestingly, important game entities often appear intermittently over sequential frames, creating visible screen flicker. This was a common technique game designers used to work around memory limitations in the original Atari hardware. However, it presents a challenge for RL because it implies that Atari game environments are partially observable. That is to say, it is often impossible to capture the complete game state from a single frame. Partial observability can be mitigated by averaging pixel colours across each pair of sequential frames [94], or even “frame stacking” [100], which concatenates previous frames with the current frame in order to reduce partial observability *and* make it possible for the agent to detect the direction in which objects are moving. However, these preprocessing steps are *not* used in this work.

The ALE is deterministic. That is, each game episode begins in the same start state and episode outcomes are fully determined by the state and action taken in each time step. Specifically, given a game stats s and action a, there is a deterministic next state s', or $p(s'|s,a) = 1$ [94]. As such, in some game titles it is possible to achieve a high score simply my memorizing an effective action sequence. There are several approaches to introducing variation and/or stochasticity into gameplay, where the two most commonly used in the literature are *no-op* and *sticky actions*.

- *no-op* [100] implies that diverse initial conditions are achieved by forcing the agent to select 'no action' for the first *no-op* frames of each game, where $no - op \in [0, 30]$, selected with uniform probability at the start of each game. However, some game titles will be more affected than others. For example, titles such as Ms. Pac-Man play a song for the first ≈ 70 game frames while the agent's actions are ignored (thus *no-op* has no effect), while the agent takes control immediately in other game titles. Furthermore, after the initial *no-op* period the environments becomes fully deterministic.[1]

- *Sticky Actions* [94] implies that agents stochastically skip screen frames with probability $p = 0.25$, with the previous action being repeated on skipped frames. This achieves two goals: 1) Artificial agents are limited to roughly the same reaction time as a human player; and 2) Stochasticity is introduced throughout the entire episode of gameplay.

Sticky actions are currently the recommended method of implementing stochasticity in the ALE [94], and are therefore assumed throughout this work. *no-op* is adopted in certain cases in order to make comparison with earlier work, in particular the well-known Deep Q-Networks (DQN) [100], as fair as possible.

7.2 Related Work in the Arcade Learning Environment

7.2.1 RL in the ALE

Historically, approaches to RL have relied on a priori designed task-specific state representations (inputs). This changed with the introduction of the Deep Q-Network or DQN ([100]). DQN employs a deep convolutional neural network architecture to encode a representation directly from screen capture (thus a task specific representation). A multi-layer perceptron is simultaneously trained from this representation to estimate a value function (the action selector) through Q-learning. Image preprocessing was still necessary and took the form of down sampling the original 210×160 RGB frame data to 84×84 and extracting the luminance channel. Moreover, a temporal sliding window was assumed in which the input to the first convolution layer

[1] An alternative test scenario has also appeared in which the RL agent takes over from game state identified by a human player in an attempt to introduce further diversity into RL agent start state selection [105, 99].

was actually a sequence of the four most recently appearing frames. This reduced the partial observability of the task, as all the game state should now be visible.

In assuming Q-learning, DQN is an off-policy method, for which one of the most critical elements is support for replay memory. As such, performance might be sensitive to the specific content of this memory (the 'memories' replayed are randomly sampled). The General Reinforcement Learning Architecture (or Gorila) extended the approach of DQN with a massively parallel distributed infrastructure (100's of GPUs) to support the simultaneous development of multiple DQN learners [105]. The contributions from the distributed learners periodically update a central 'parameter server' that ultimately represents the solution. Gorila performed better than DQN on most game titles, but not in all cases, indicating that there are possibly still sensitivities to replay memory content.

Q-learning is also known to potentially result in action values that are excessively high. Such 'overestimations' were recently shown to be associated with inaccuracies in the action values, where this is likely to be the norm during the initial stages of training [151]. A solution proposed for addressing this issue was to introduce two sets of weights, one for action selection and one for policy evaluation [151]. This was engineered into the DQN architecture by associating the two roles with DQN's online network and target network respectively.[2] The resulting Double DQN framework improved on the original DQN results for more than half of the 49 game titles from the ALE task.

Hierarchical Reinforcement Learning has also recently demonstrated improvements over DQN. Vezhnevets *et al.* proposed using parallel interacting learners operating at different timescales in order to separate the learning of subgoals (long time scale) from learning the decision-making policy (short time scale, defines the frame-to-action mapping) [155]. In particular, their approach made progress in the game Montezuma's Revenge, an environment known to be especially challenging due to its sparse reward signal. The subgoal learner, or "Manager", was able to establish intermediate intrinsic rewards that helped guide the policy learner, or "Worker", to the first extrinsic reward, or picking up the key in the first room (a task that requires a long sequence of actions before any extrinsic reward signal from the environment).

[2] The 'online' network in DQN maintains the master copy of the MLP, whereas the target network is updated during 'experience replay' [100].

Most recently, on-policy methods (e.g., Sarsa) have appeared in which multiple independent policies are trained in parallel [99]. Each agents' experience of the environment is entirely independent (no attempt is made to enforce the centralization of memory/experience). This means that the set of RL agents collectively experience a wider range of states. The resulting evaluation under the Atari task demonstrated significant reductions to computational requirements[3] and better agent strategies. That said, in all cases, the deep learning architecture is specified a priori and subject to prior parameter tuning on a subset of game titles.

Neuro-evolution represents one of the most widely investigated techniques within the context of agent discovery for games. Hausknecht *et al.* [49] performed a comparison of different neuro-evolutionary frameworks under two state representations: game-specific objects versus screen capture. Pre-processing for screen capture took the form of down sampling the original 210 × 160 RGB frame data to produce eight 'substrates' of dimension $16 \times 21 = 336$; each substrate corresponding to one of the eight colours present in a SECAM representation (provided by the ALE). If the colour is present in the original frame data, it appears at a corresponding substrate node. [49] compared Hyper-NEAT, NEAT and two simpler schemes for evolving neural networks under the suite of Atari game titles. Hyper-NEAT provides a developmental approach for describing large neural network architectures efficiently,[4] while NEAT provides a scheme for discovering arbitrary neural topologies as well as weight values, beginning with a single fully connected neuron. NEAT was more effective under the low dimensional object representation, whereas Hyper-NEAT was preferable for the substrate representation.

Finally, [87] revisit the design of task-specific state information using a hypothesis regarding the action of the convolutional neuron in deep learning. This resulted in a state space in the order of 110 million state variables when applied to Atari screen capture, but simplified decision-making to a linear model. Thus, an RL agent could be identified using the on-policy Temporal Difference method of Sarsa. In comparison to deep learning, the computational requirements for training and deployment are considerably lower, but the models produced are only as good as the ability to engineer appropriate inputs.

[3] A 16 core CPU as opposed to a GPU.

[4] In this case just over 900,000 weights in the case of screen capture input.

7.2.2 Multi-task RL in the ALE

The approaches reviewed in Section 7.2.1 assumed that a single RL agent was trained on each game title. Conversely, multi-task RL (MTRL) attempts to take this further and develop a single RL agent that is able to play multiple game titles. As such, MTRL is a step towards ’artificial general intelligence’, and represents a much more difficult task for at least two reasons: 1) MTRL agents must not ‘forget’ any of their policy for playing a previous game while learning a policy for playing a new game, and; 2) during test, an MTRL agent must be able to distinguish between game titles without recourse to additional state information.

To date, two deep learning approaches have been proposed for this task. Parisotto *et al.* first learn each game title independently and then use this to train a single architecture for playing multiple titles [109]. More recently [77] proposed a modification to Double DQN in which subsets of weights (particularly in the MLP) are associated with different tasks and subject to lower learning rates than weights not already associated with previously learned tasks. They were able to learn to play up to 6 game titles at a level comparable with the original DQN (trained on each title independently), albeit when the game titles are selected from the set of games for which DQN was known to perform well on.

7.3 Summary

Scaling RL to real-world tasks requires a representation that is: 1) Able to cope with high-dimensional sensor data; and 2) General enough to be applied to a wide variety of tasks without extensive parameter tuning. Video games in the ALE provide an interesting test domain for scalable RL. In particular, they cover a diverse range of dynamic task environments, all through a common high-dimensional visual interface, or the game screen. As such, a growing body of research has been employing the suite of Atari video game titles to study scalability and generalization from two perspectives: domain-independent reinforcement learning, and, more recently, multi-task RL. The empirical study documented in Chapter 9 considers both approaches to generalization in evaluating the Tangled Program Graph algorithm (Chapter 8).

Chapter 8

Algorithm Description: Tangled Program Graphs

8.1 Overview

The algorithm described in this section, Tangled Program Graphs (TPG), addresses the question of how the hierarchical interdependency between teams of programs can be established entirely through interaction with the task environment. To do so, TPG provides a complete framework for organizing multiple teams into variably deep/wide directed graph structures, or *policy graphs*. Policies are initialized in their simplest form and are then capable of self-organizing into complex structures through continuous, open-ended evolution. Thus, more complex topologies can naturally emerge as soon as they perform better than simpler solutions.

As in the Policy Tree approach (Chapter 5), a single team of programs represents the smallest stand-alone decision-making entity, or behavioural *module*. Constructing policy graphs through emergent (behavioural) modularity [107] implies that multiple independent teams are recombined with no prior knowledge regarding how many to include, which teams might work well together, or how to combine them. Unlike Policy Trees, organisms in TPG are developed over a single phase of evolution. Emergent events (i.e. major transitions) may occur at any time and the topology of the graph is unconstrained (See Figure 8.1(b) or 2.2). For example, policy graphs can interact with the task environment at various levels of abstraction simultaneously, referencing atomic actions directly or operating through an arbitrarily deep hierarchy of policies. This means that as TPG solutions become more capable and their interactions with the environment potentially uncover aspects of the task that could not be reached before, the policy graph is not restricted to operating *through* lower-level structures developed prior to the environmental change. This is distinct from the Policy Tree approach in which it was assumed that sufficient prior knowledge existed to design source tasks with sufficient diversity for solving the target task. Furthermore, as programs composing a team typically index different subsets of the state space (i.e., the screen in the case of visual reinforcement learning), the resulting policy graph will incrementally adapt, indexing more or less of the state space *and* defining the types of decisions made in different regions.

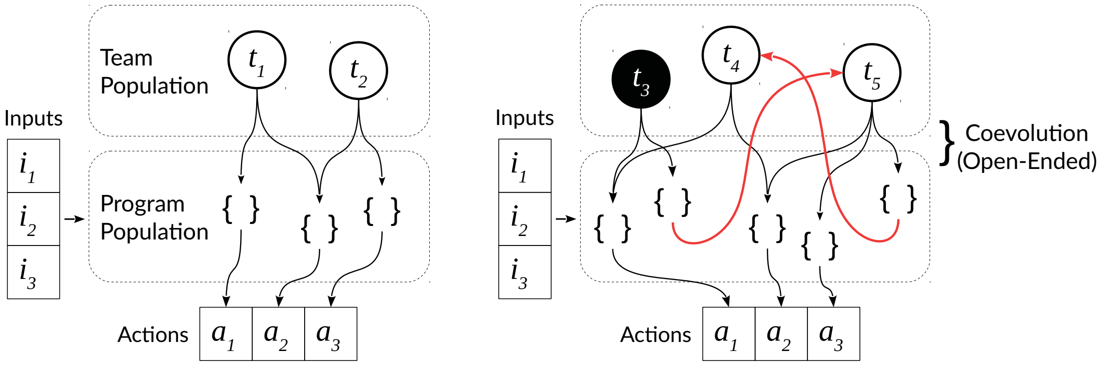
The following sections describe the complete TPG algorithm by first establishing the procedures for team initialization, variation, and evaluation, followed by a description of the overall GA used in policy development.

8.2 Initialization and Variation

Evolution begins with a program population in which program actions are limited to the task specific (atomic) actions, Figure 8.1(a). The team initialization procedure is identical to that previously established in Section 4.2.1. In order to provide for the evolution of hierarchically organized code under a completely open-ended process of evolution (i.e. emergent modularity), program variation operators (Algorithm 3) are allowed to introduce actions that index other teams within the team population. To do so, when a program's action is modified, it has a probability (p_{atomic}) of referencing either a different atomic action or any other team created in a *previous generation*. Specifically, the action set from which new program actions are sampled ($\mathcal{A}'$ in Line 10 of Algorithm 3) will correspond to the set of atomic actions, $\mathcal{A}$, with probability p_{atomic}, and will otherwise correspond to the set of teams present from any previous generation, $\mathcal{H}$. Notice that the *previous generation* constraint avoids new root nodes from being immediately subsumed within a policy graph before ever being evaluated in the task environment (and surviving a round of selection). In short, variation operators have the ability to *incrementally* construct multi-team graphs, Figure 8.1(b). Each vertex in the graph is a team, while each team member, or program, represents one outgoing edge leading either to another team or an atomic action.

8.3 Decision-Making in Policy Graphs (Evaluation)

Algorithm 9 details the process for decision-making with the TPG policy graph, which is repeated at every time step until an end-of-game state is encountered and fitness for the policy graph can be expressed. Naturally, decision-making in a policy graph begins at the root team (e.g. t_3 in Figure 8.1(b)), where each program in the team



(a) Initial policies (b) Emergence of policy graphs

Figure 8.1: TPG Policies. Decision-making in each time step begins at the root team and follows the edge with the winning program bid (output) until an atomic action is reached. The initial population contains only single-team policies, i.e. program actions refer exclusively to actions defined by the task environment (a). During evolution, program variation operators may introduce actions that reference other teams, indicated by the red lines in (b). Mutations of this nature correspond to major hierarchical transitions, or emergent events, which may result in an increase or decrease in the complexity of the organism.

will produce one bid relative to the current state observation, $\vec{s}(t)$. Graph traversal then follows the program / edge with the largest bid, repeating the bidding process for the *same* state, $\vec{s}(t)$, at every team / vertex along the path until an atomic action is reached. Thus, in sequential decision-making tasks, the policy graph computes *one* path from root to action at every time step, where only a subset of programs in the graph (i.e those in teams/vertexes along the path) require execution. Cycles may exist in the graph, but they are never followed during execution. That is, a team is never visited twice per decision. The single visit constraint is enforced through a test for vertex uniqueness. If the action of the program with highest output corresponds to a previously visited vertex, then the next highest bid is selected and the vertex uniqueness test is repeated for the next candidate vertex. Variation operators are constrained such that each team maintains at least one program that has an atomic action, hence guaranteeing cycles never appear during evaluation. This constraint is enforced within team variation operators through two simple rules:

1. Never remove from a team the only team member (program) with an atomic action. Thus, the sampling procedure in Line 9 of Algorithm 2 will be repeated until a legal program is identified.

2. When mutating the action of a program, if it is the only team member with an atomic action, then the new action is also sampled from the set of atomic actions (Line 10 of Algorithm 3).

Algorithm 9 Selecting an action through traversal of a policy graph. P is the current program population. $\mathcal{A}$ is the set of atomic actions. tm_i is the current team (initially a root node). $\vec{s}(t)$ is the state observation at time t. V is the set of teams visited throughout *this* traversal (initially empty). First, all programs in tm_i are executed relative to the current state $\vec{s}(t)$ (Lines 3,4). The algorithm then considers each program in order of bid (highest to lowest, Line 7). If the program has an atomic action, the action is returned (Line 8). Otherwise, if the program's action points to a team that has not yet been visited, the procedure is called recursively on that team. Thus, while a policy graph may contain cycles, they are not followed during traversal. In order to ensure an atomic is always found, team variation operators are constrained such that each team maintains at least one program that has an atomic action.

```
 1: procedure SelectAction($tm_i$, $\vec{s}(t)$, $V$)
 2:     $V = V \cup tm_i$                                ▷ add $tm_i$ to visited teams
 3:     for all $p_i \in tm_i$ do
 4:         $bid(p_i) = exec(p_i, \vec{s}(t))$           ▷ run program on $\vec{s}(t)$ and save result
 5:     end for
 6:     $tm'_i = sort(tm_i)$                             ▷ sort programs by bid, highest to lowest
 7:     for all $p_i \in tm'_i$ do
 8:         if $action(p_i) \in \mathcal{A}$ then return $action(p_i)$      ▷ atomic action reached
 9:         else if $action(p_i) \notin V$ then
10:             return SelectAction($action(p_i), \vec{s}(t), V$)          ▷ follow graph edge
11:         end if
12:     end for
13: end procedure
```

8.4 Overall TPG Training Algorithm

As multi-team policy graphs emerge, an increasingly tangled web of connectivity develops between the team and program populations. The number of unique solutions, or policy graphs, at any given generation is equal to the number of root nodes (i.e. teams that are not referenced as any program's action) in the team population. Only these root teams are candidates to have their fitness evaluated, and are subject to modification by the variation operators. As such, rather than pre-specify the desired team population size, only the number of *root* teams to maintain in the population, or R_{size}, requires prior specification. Evolution is driven by a generational GA (Algorithm 10) such that in each generation R_{gap} of the root teams, rt_i, are deleted and replaced by offspring of the surviving roots. The process for generating team offspring (Lines 5 - 9) uniformly samples and clones a root team, then applies mutation-based variation operators to the cloned team which remove, add, and mutate some of its programs (Section 4.2.2 with modifications detailed in Section 8.2). The team generation process introduces new root nodes until the number of roots in the population reaches R_{size}. The total number of sampling steps for generating offspring fluctuates, as root teams (along with the lower policy graph) are sometimes 'subsumed' by a new team. Conversely, graphs can be separated, for example through program action mutation, resulting in new root nodes / policies. This implies that after initialization, both team and program population size varies. Furthermore, while the number of root teams remains fixed, the number of teams that become 'archived' as internal nodes (i.e. a library of reusable code) fluctuates.

Limiting evaluation, selection, and variation to root teams only has 2 critical benefits:

1. The cost of evaluation and the size of the search space remains low because only a fraction of the team population (root teams) represent unique policies to be evaluated and modified in each generation.

2. Since only root teams are deleted, introduced, or modified, policy graphs are *incrementally* developed from the bottom up. As such, lower-level complex structures within a policy graph are protected as long as they contribute to an overall strong policy.

Table 8.1: Parameters associated with the TPG training procedure, in addition to those outlined in Chapter 4

Parameter	Description
R_{size}	Number of root teams to maintain in the team population.
R_{gap}	Number of root teams deleted and introduced in each generation
p_{atomic}	Probability of modified program action referencing an atomic action
t_{eval}	Number of evaluations per team in each generation
l_{eval}	Maximum number of evaluations per team

Algorithm 10 The overall TPG training algorithm. Parameters are listed in Table 8.1. After initialization (Line 3), the GA continues in a loop until being terminated due to an external resource budget, for example, a wall-clock time constraint. T^t and P^t refer to the team and program populations at time t. VARIATION is a reference to the team variation procedure detailed in Section 4.2.2 with additional support for TPG outlined in Section 8.2. Policies are evaluated in t_{eval} episodes per generation, up to a max of l_{eval} episodes per lifetime. $numEval(rt_i)$ returns the number of evaluations for root team (policy) rt_i so far. When the cost of evaluations is high, these parameters can be set such that weak policies are identified and replaced early, while promising policies are verified with additional evaluations. Note that teams are evaluated as a whole, i.e. programs have no concept of individual fitness. As such, the program population is updated and modified implicitly through team variation operators, (Line 8) and selection (Line 15). When teams are deleted in Line 15, programs with no remaining team membership are also deleted. *task* refers to the task environment. Episode outcome refers to the final reward for an episode (e.g. the game score in the ALE).

```
 1: procedure TRAIN
 2:     $t = 0$
 3:     initialize $R_{size} - R_{gap}$ teams, add to $T^t$ (add new programs to $P^t$)
 4:     loop
 5:         while $T^t$ contains less than $R_{size}$ root teams do          ▷ Variation
 6:             uniformly sample parent root team $rt_i \in T^t$
 7:             copy $rt_i$ into $rt_j$
 8:             $rt'_j \leftarrow$ VARIATION($rt_j$)
 9:             add $rt'_j$ to $T^t$
10:         end while
11:         for all root teams $rt_i \in T^t : numEval(rt_i) < l_{eval}$ do     ▷ Evaluation
12:             deploy $rt_i$ for $t_{eval}$ episodes in task (Algorithm 9)
13:             $Rank(rt_i) \leftarrow$ mean episode outcome for $rt_i$
14:         end for
15:         delete $R_{gap}$ lowest ranked root teams from $T^t$              ▷ Selection
16:         delete from $P^t$ programs that are not part of any team
17:         $t = t + 1$
18:     end loop
19: end procedure
```

8.5 On Diversity Maintenance

No explicit diversity maintenance is employed in the development of policy graphs. Initial experiments with the framework had adopted an approach to fitness regularization similar to that described in Section 5.4.1, but no significant benefit was observed. This is most likely due to the high-dimensionality of the visual reinforcement learning task under which TPG is benchmarked. In particular, a large input space implies that initial programs are more likely to specialize in particular regions without explicitly enforcing selective pressure to do so. Furthermore, the neutrality test (Section 4.3.3) is still in effect, and while this does not necessarily imply diversity of team behaviours, it at least ensures that program bidding behaviours are unique. Finally, the diversity-generating capacity of TPG is significantly more open-ended than under the Policy Tree representation, which potentially mitigates the requirement for explicit diversity maintenance.

Chapter 9

Empirical Evaluation: Tangled Program Graphs

9.1 Overview

This chapter documents an empirical evaluation of the Tangled Program Graph (TPG) algorithm (Chapter 8) in the Arcade Learning Environment (ALE), a video game emulator specifically designed for addressing dynamic, high-dimensional, and partially observable tasks in RL (Chapter 7). The ALE supports hundreds of game titles and has recently received significant attention on account of human-competitive results from deep learning, e.g. deep Q-networks (DQN), [100]. While a range of comparator algorithms will be considered, the focus will be on a comparison with DQN, which represents a high-quality baseline RL algorithm with extensive results under the ALE. DQN also represents the starting point for a variety of deep learning approaches to video game playing, as reviewed in Chapter 7.

In all the TPG experiments in this chapter, state observations are defined in terms of direct screen capture, while actions are limited to those of the original Atari console (Chapter 7). Thus, learning agents interact with games via the same interface experienced by human players. Two broad approaches to policy development are investigated:

1. Single-task learning, in which a distinct RL agent is developed for each game title.

2. Multi-task learning, in which multiple game titles are learned simultaneously, producing a single RL agent capable of playing multiple game titles from direct screen capture alone (i.e. agents are not provided any additional information regarding what game is currently being played).

TPG discovers policies that are competitive with DQN in both cases. However, unlike deep learning, the proposed TPG framework takes an explicitly emergent, developmental approach to policy identification. The goal of this study to establish to what degree TPG's capacity for constructing policy graph topologies 'bottom-up' is able to match the quality of deep learning solutions without incurring the corresponding computational complexity. Specifically, deep learning assumes that the neural architecture is designed a priori, with the same architecture employed for each game title. Thus, deep learning always performs millions of calculations *per decision*. TPG, on the other hand, has the potential to tune policy complexity to each task environment, or game title, requiring only ≈ 1000 calculations per decision in the most complex case, and ≈ 100 calculations in the simpler cases.

In short, the aim of this work is to demonstrate that much simpler solutions can be *discovered* to dynamic, high-dimensional, and partially observable environments in RL without making any prior decisions regarding model complexity. As a consequence, the computational costs typically associated with deep learning are avoided without impacting on the quality of the resulting policies, i.e. the cost of training and deploying a solution is now much lower. Solutions operate in real-time without any recourse to multi-core or GPU hardware platforms, thus potentially simplifying the developmental/deployment overhead in posing solutions to challenging RL tasks.

9.2 Experimental Methodology

For comparative purposes, evaluation of TPG will assume the same general approach as established in the original DQN evaluation [100]. Thus, we assume the same subset of 49 Atari game titles[1]. Each game was designed to be interesting and challenging for human players, and thus task environments with a wide range of properties are identified. The stochastic version of the ALE is employed throughout this chapter (i.e., Sticky Actions are present), with the default frame skip probability of 0.25 assumed (See Chapter 7). Each episode (training and test) continues until the simulator returns a 'game over' signal or a maximum of 18,000 frames is reached. Post training, the champion TPG agents are tested over 30 test episodes initialized with a stochastically selected number of initial *no-op* actions. In general, *no-op* results in random initial start states but does *not* imply a stochastic environment. The limitations of the *no-op* approach (See Chapter 7) imply that training and test results reported for

[1] A preliminary comparison for TPG on the 20 games for which DQN is known to perform worse than 75% of a human game tester appears in [72]. The comparator set of algorithms was also limited to DQN and Hyper-NEAT.

DQN in [100] are relative to a deterministic version of ALE. Thus, while adopting the stochastic version of ALE in this study implies a more challenging environment than used by the principal DQN study, including stochasticity will make the results reported herein comparable with the widest range of previous and future studies, as the settings employed here are now explicitly recommended by the authors of the ALE [94]. Finally, the available actions per game is also assumed to be known, where this is usually a subset of the 18 possible atomic actions (joystick positions) in the ALE.[2]

Five independent TPG runs are performed per game title, where this appears to reflect most recent practice for deep learning results.[3] The same parameterization for TPG was used for all games (Section 9.4). The only information provided to the agents was the number of atomic actions available for each game, the preprocessed screen frame during play (Section 9.3), and the final game score. Each policy graph was evaluated in 5 game episodes per generation, up to a maximum of 10 episodes per lifetime. Fitness for each policy graph is simply the average game score over all episodes. A single champion policy for each game was identified as that with the highest training reward at the end of evolution.

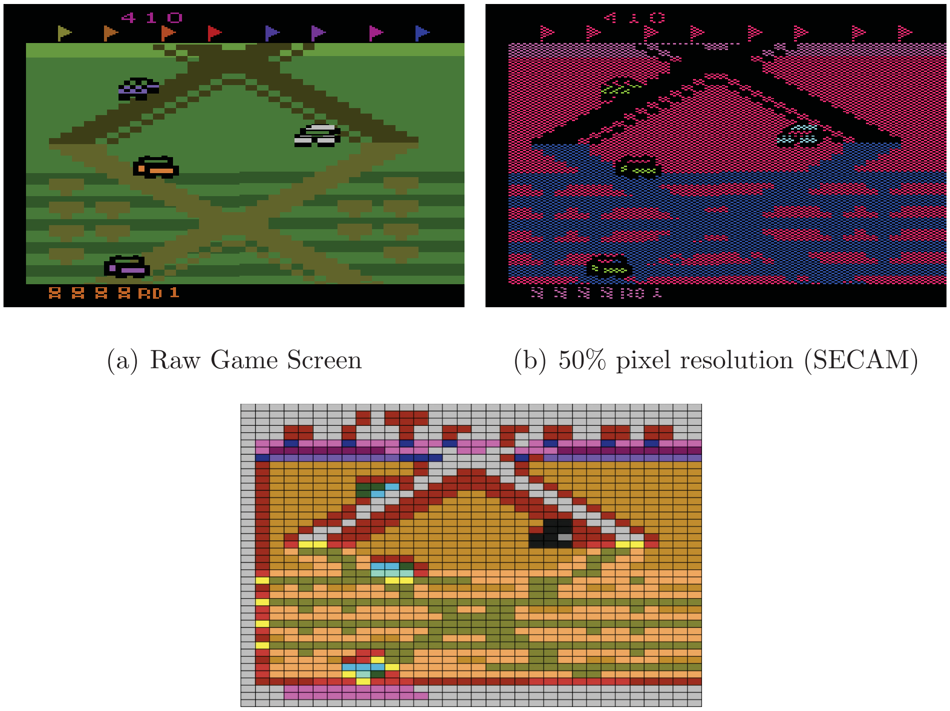
9.3 Screen Capture State Space

Based on the observation that the visual input has a lot of redundant information (i.e. visual game content is designed for maximizing entertainment value, as opposed to a need to convey content with a minimal amount of information), we adopt a quantization approach to preprocessing. The following 2-step procedure is applied to every game frame:

1. A checkered pattern mask is used to sample 50% of the pixels from the raw game screen (Figure 9.1(b)). Each remaining pixel assumes the 8-colour SECAM encoding. The SECAM encoding is provided by ALE as an alternative to the default NSTC 128-colour format. Uniformly skipping 50% of the raw screen pixels improves the speed of state retrieval while having minimal effect on the

[2] The study of [87] question this assumption, but find that better performance resulted when RL agents were constructed with the full action space.

[3] The original DQN results only reflected a single run per title [100].



(a) Raw Game Screen

(b) 50% pixel resolution (SECAM)

(c) Decimal State Variables

Figure 9.1: Screen quantization steps, reducing the raw Atari pixel matrix (a) to 1344 decimal state variables (c) using a checkered subsampling scheme (b).

final representation, since important game entities are usually larger than a single pixel.

2. The frame is subdivided into a 42×32 grid.[4] Each grid tile is described by a single byte, in which each bit encodes the presence of one of eight SECAM colours within that tile. The final quantized screen representation includes each tile byte as a decimal value, so defining a sensory state space $\vec{s}(t)$ of $42 \times 32 = 1,344$ decimal state variables in the range of $0 - 255$, visualized in Figure 9.1(c) for the game Up 'N Down at time step (frame) t.

This state representation is inspired by the Basic method defined in [14]. Note, however, that this method does not use a priori background detection or pairwise combinations of state variables.

In comparison to the DQN approach (e.g. [100, 105]), no attempt is made to design out the partially observable properties of game content (see discussion of Section

[4] Implies that the original 210×160 screen is divided by 5.

7.2.1). Moreover, the deep learning architecture's three layers of convolution filters reduce the down sampled $84 \times 84 = 7,056$ pixel space to a dimension of $3,136$ before applying a fully connected multi-layer perceptron (MLP).[5] It is the combination of convolution layer and MLP that represents the computational cost of deep learning. Naturally, this imparts a fixed computational cost of learning as the entire DQN architecture is specified a priori (Section 9.5.5).

In contrast, TPG evolves a decision-making agent from a $1,344$ dimensional space. In common with the DQN approach, no feature extraction is performed as part of the preprocessing step, just a quantization of the original frame data. Implicit in this is an assumption that the state space is highly redundant. TPG therefore perceives the state space, $\vec{s}(t)$ (Figure 9.1(c)), as read-only memory. Each TPG program then defines a potentially unique subset of inputs from $\vec{s}(t)$ for incorporation into their decision-making process. The emergent properties of TPG are then required to develop the complexity of a solution, or policy graph, with programs organized into teams and teams into graphs. Thus, rather than assuming that all screen content contributes to decision-making, the approach adopted by TPG is to adaptively subsample from the quantized image space. The specific subset of state variables sampled within each agent policy is an emergent property, discovered through interaction with the task environment alone. The implications of assuming such an explicitly emergent approach on computational cost will be revisited in Section 9.5.5.

9.4 Parameterization

Deploying population based algorithms can be expensive on account of the number of parameters and inter-relation between different parameters. In this work, no attempt has been made to optimize the parameterization, Table 9.1, instead we carry over a basic parameterization from experience with evolving single teams under a supervised learning task [91]. This approach is also distinct from most of the genetic programming literature as applied to games in which a set of task specific operators are first identified (e.g. [121]). Such an approach is justified on the basis that previous research also assumes inputs designed for a specific game title. Given that the entire purview of TPG is to identify solutions under a 'visual' RL context, the instruction

[5] See Appendix E for a detailed explanation of how DQN's complexity is calculated.

set supports arithmetic, trigonometric, logarithmic and a conditional operator. This also means a common instruction set could be assumed for all experiments in this thesis. Naturally, no claims are made regarding the optimality of the composition of instruction set itself.

Table 9.1: Parameterization of TPG.

Neutrality test (Section 4.3.3)	
Number of historical samples in diversity test	50
Threshold for bid uniqueness (τ)	10^{-4}

Team population	
Number of (root) teams in initial population (R_{size})	360
Number of root nodes that can be replaced per generation (R_{gap})	50%
Probability of deleting or adding a program (p_{md}, p_{ma})	0.7
Max. initial team size (ω)	5
Max. team size (Ω)	∞
Prob. of creating a new program (p_{mm})	0.2
Prob. of changing a program action (p_{mn})	0.1
Prob. of defining an action as a team or atomic action (p_{atomic})	0.5
Number of episodes per generation for each root team (t_{eval})	5
Max. number of episodes per lifetime for each root team (l_{eval})	10

Program population	
Total number of registers per program (*numRegisters*)	8
Max. number of instructions a program may take (*maxProgSize*)	96
Prob. of deleting or adding an instruction within a program (p_{delete}, p_{add})	0.5
Prob. of mutating a instruction within a program (p_{mutate})	1.0
Prob. of swapping a pair of instructions within a program (p_{swap})	1.0

Three basic categories of parameter are listed: Neutrality test (Section 4.3.3), Team population, and Program population (Figure 8.2). In the case of the Team population, the single biggest parameter decisions are the population size, R_{size} (how many *root* teams to simultaneously support), and how many root teams to replace at each generation (R_{gap}). The parameters controlling the application of the variation operators common to earlier instances of SBB ($p_{md}, p_{ma}, p_{mm}, p_{mn}$) also assume the values used under supervised learning tasks [91]. Conversely, p_{atomic} represents a parameter specific to TPG, where this defines the relative chance of mutating an action to an atomic action versus a pointer to a team (Section 8.2).

Likewise, the parameters controlling properties of the Program population assume values used for SBB as applied to supervised learning tasks for all but $maxProgSize$.

In essence this has been increased to the point where it is unlikely to be encountered during evolution.

The computational limit for TPG is defined in terms of a computational resource time constraint. Thus, experiments ran on a shared cluster with a maximum runtime of 2 weeks per game title. The nature of some games allowed for > 800 generations, while others limited evolution to a few hundred. No attempt was made to parallelize execution within each run (i.e. the TPG code base executes as a single thread), the cluster merely enabled each run to be made simultaneously. Incidentally, the DQN results required 12–14 days per game title on a GPU computing platform [105].

9.5 Single-Task Learning

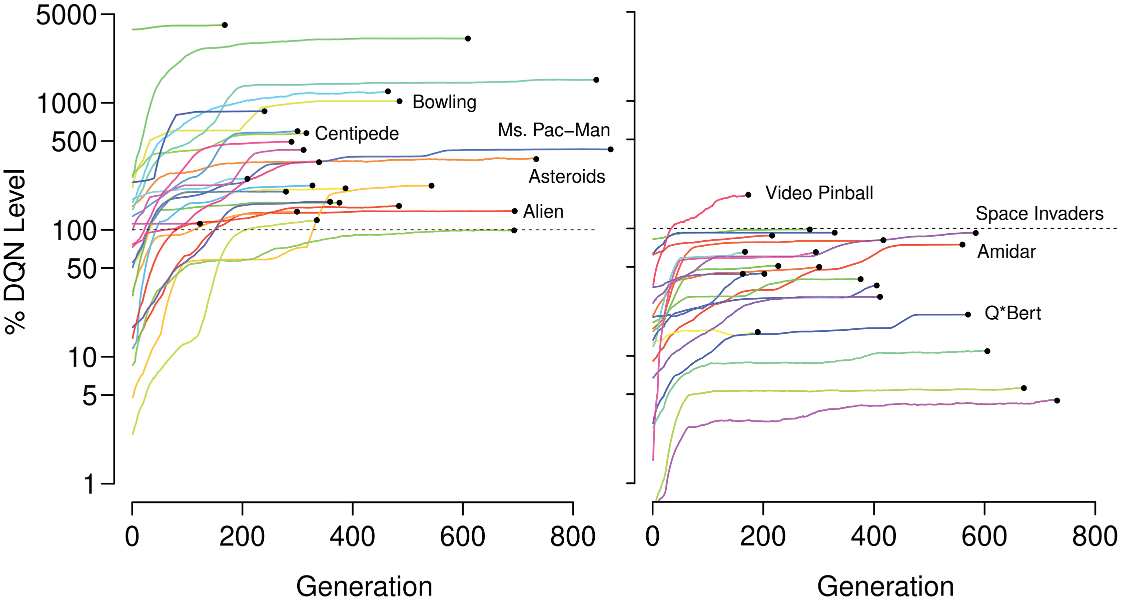
9.5.1 Overview

This section documents TPG's ability to build decision-making policies in the ALE from the perspective of domain-independent AI, that is, discovering policies for a variety of ALE game environments with no task-specific parameter tuning. Before presenting detailed results, we provide an overview of *training performance* for TPG on the suite of 49 ALE titles common to most benchmarking studies (Section 7.2.1). Figure 9.2 illustrates average TPG training performance (across the 5 runs per game title) as normalized relative to DQN's test score from the same game titles (100%) and random play (0%), [100]. The random agent simply selects actions with uniform probability at each game frame. Normalized score is calculated as:

$$100 \times \frac{TPG_{score} - RandomPlay_{score}}{DQN_{score} - RandomPlay_{score}} \tag{9.1}$$

Normalizing scores makes it possible to plot TPG's progress relative to multiple games together regardless of the scoring scheme in different games, and facilitates making a direct comparison with DQN.

Under test conditions, TPG exceeds DQN's score in 27 games (Figure 9.2(a)), while DQN maintains the highest score in 21 games (Figure 9.2(b)). Thus, TPG and DQN are broadly comparable from a performance perspective, each matching/beating the other in a subset of game environments. Indeed, there is no statistical difference between TPG and DQN test scores over all 49 games, Section 9.5.2. However, TPG



(a) (b)

Figure 9.2: TPG training curves, each normalized relative to DQN's score in the same game (100%) and random play (0%). (a) shows curves for the 27 games in which TPG ultimately exceeded the level of DQN under *test* conditions. (b) shows curves for the 21 games in which TPG did not reach DQN level during test. Note that in several games, TPG began with random policies (generation 1) that exceeded the level of DQN. Note that these are training scores averaged over 5 episodes in the given game title, and are thus not as robust as DQN's test score used for normalization. Also, these policies were often degenerate. For example, in Centipede, it is possible to get a score of 12,890 by selecting the 'up-right-fire' action in every frame. While completely un-interesting, this strategy exceeds the test score reported for DQN (8,390) *and* the reported test score for a human professional video game tester (11,963) [100]. Regardless of their starting point, TPG policies improve throughout evolution to become more responsive and interesting. Note also that in Video Pinball, TPG exceeded DQN's score during training but not under test. Curve for Montezuma's Revenge not pictured, a game in which neither algorithm scores any points.

produces much simpler solutions in *all* cases, largely due to its emergent modular representation, which automatically scales through interaction with the task environment. That is to say, concurrent to learning a strategy for gameplay, TPG explicitly answers the question of: 1) *what to index* from the state representation for each game; and, 2) what components *from other candidate policies* to potentially incorporate within a larger policy. Conversely, DQN assumes a particular architecture, based on a specific deep learning–MLP combination, in which all state information *always* contributes.

9.5.2 Detailed Results

In this section, performance of TPG policies are compared under post-training test conditions against a representative set of comparator algorithms. Two sets of comparator algorithms are considered:

- **Screen capture state:** construct models from game state, $\vec{s}(t)$, defined in terms of some form of screen capture input.[6] These include the original DQN deep learning results [100], DQN as deployed through a massive distributed search [105], double DQN [151], and hyper-NEAT [49]. While the original DQN report emphasized comparison with a human professional game tester [100], we avoid such a comparison here primarily because the human results are not reproducible.

- **Engineered features:** define game state, $\vec{s}(t)$, in terms of features designed a priori; thus, significantly simplifying the task of finding effective policies for game play, but potentially introducing unwanted biases. Specifically, the Hyper-NEAT and NEAT results use hand crafted 'Object' features specific to each game title in which different 'substrates' denote the presence and location of different classes of object (see [49] and the discussion of Section 7.2.1). The Blob-PROST results assume features designed from an attempt to reverse engineer the process performed by DQN [87]. The resulting state space is a vector of $\approx 110 \times 10^6$ variables from which a linear RL agent is constructed (Sarsa). Finally, the best performing Sarsa RL agent (Conti-Sarsa) is included from the

[6] Reviewed in Section 7.2.1 for comparator algorithms and detailed in Section 9.3 for TPG.

DQN study [100] where this assumes the availability of 'contingency awareness' features [15].

In each case TPG based on screen capture will be compared to the set of comparator models across a common set of 49 Atari game titles. Statistical significance will be assessed using the Friedman test, where this is a non-parametric form of ANOVA [30, 58]. Specifically, parametric hypothesis tests assume commensurability of performance measures. This would imply that averaging results across multiple game titles makes sense. However, given that the score step size and types of property measured in each title are typically different, then averaging Null test performance across multiple titles is no longer commensurable. Conversely, the Friedman test establishes whether or not there is a pattern to the ranks. Rejecting the Null hypothesis implies that there is a pattern, and the Nemenyi post hoc test can be applied to assess the significance [30, 58].

In the case of RL agents derived from screen capture state information (Table D.1, Appendix D), the Friedman test returns a $\chi^2_F = 21.41$ which for the purposes of the Null hypothesis has an equivalent value from the F-distribution of $F_F = 5.89$ [30]. The corresponding critical value $F(\alpha = 0.01, 4, 192)$ is 3.48, hence the Null hypothesis is rejected. Applying the post hoc Nemenyi test ($\alpha = 0.05$) provides a critical difference of 0.871. Thus, relative to the best ranked algorithm (Gorila), only Hyper-NEAT is explicitly identified as outside the set of equivalently performing algorithms (or $2.63 + 0.871 < 3.87$). This conclusion is also borne out by the number of game titles that each RL agent provides best case performance; Hyper-NEAT provides 4 best case game titles, whereas TPG, Double DQN and Gorila return at least 11 best title scores each (Table D.1, Appendix D).

Repeating the process for the comparison of TPG[7] to RL agents trained under hand crafted features (Table D.2, Appendix D), the Friedman test returns a $\chi^2_F = 80.59$ and an equivalent value from the F-distribution of $F_F = 33.52$. The critical value is unchanged as the number of models compared and game titles is unchanged, hence the Null hypothesis is rejected. Likewise the critical difference from the post hoc Nemenyi test ($\alpha = 0.05$) is also unchanged 0.871. This time only the performance of the Conti-Sarsa algorithm is identified as significantly worse (or $2.16 + 0.871 < 4.76$).

[7] TPG still assumes screen capture state.

In summary, these results mean that TPG is able to provide an RL agent that performs as well as current state-of-the-art, despite having to develop all the architectural properties of a solution. Conversely, DQN assumes a common pre-specified deep learning topology consisting of millions of weights. Likewise, Hyper-NEAT assumes a pre-specified model complexity of $\approx 900,000$ weights, irrespective of game title. As will become apparent in the next section, TPG is capable of evolving policy complexities that reflect the difficulty of the task.

9.5.3 Simplicity Through Emergent Modularity

As discussed in Section 8.2, the simplest stand-alone decision-making entity in TPG is a single team of programs, where all policies are initialized as a single-team of between 2 and ω programs. Throughout evolution, search operators may incrementally combine teams to form policy graphs. By compartmentalizing decision-making over multiple independent modules (teams), and incrementally combining modules into policy graphs, two critical benefits are achieved:

Adaptive Complexity: The number and complement of programs per team and teams per graph is an emergent, open-ended property driven by interaction with the task environment. That is, policies are initialized in their simplest form and only complexify when/if simpler solutions are outperformed.

State Space Selectivity: Each program indexes a small proportion of the state space. As the the number of teams and programs in each policy increases, the policy will index more of the state space *and* optimize the decisions made in each region. However, recall form section 8.3 that a single decision, or mapping from state observation to atomic action, requires traversing a single path from root node to atomic action. As such, while the decision-making capacity of the policy graph expands through environment-driven complexification, the modular nature of a graph representation implies that the *cost* of making each decision, as measured by the number of teams/programs which require execution, remains relatively low

Figures 9.3 and 9.4 quantify these properties by examining, for the champion policy throughout evolution from game titles in which TPG matched or exceeded the score from DQN (Figure 9.2(a) and Table D.1, Appendix D), the number of teams per policy vs. teams visited per decision (Figure 9.3) and the proportion of input space covered by the policy as a whole vs. the proportion indexed per decision (Figure 9.4).

Development of modularity for TPG policies is non-monotonic, and the specificity of team compliment as a function of game environment is readily apparent in Figure 9.3. For example, a game such as Asteroids may see very little in the way of increases to team complement as generations increase. Conversely, Ms. Pac-Man, which is known to be a complex task [111, 117], saw the development of a policy graph incorporating ≈ 200 teams. Importantly, making a decision in any single time step requires following *one* path from the root team to atomic action. Thus, the cost in mapping a single game frame to an atomic action is not linearly correlated to the graph size. For example, while the number of teams in the Alien policy was ≈ 60, on average only 4 teams were visited per graph traversal during test (See × symbols in Figure 9.3). Indeed, while the total number of teams in champion TPG policy graphs ranges from 7 (Asteroids) to 300 (Bowling), the average number of teams visited per decision is typically less than 5, Figure 9.3. The trajectory denoted by 'Random' in Figure 9.3 refers to a run in which policies were assigned random fitness values. The lack of development confirms that complex policies emerge by selective pressure rather than drift or other potential biases.

9.5.4 Evolving Adapted Visual Fields

Each Atari game represents a unique graphical environment, with important events occurring in different areas of the screen, at different resolutions, and from different perspectives (e.g. global maze view versus first-person shooter). Part of the challenge with high-dimensional visual input data is determining what information is relevant to the task. Naturally, as TPG policy graphs develop, they will incrementally index more of the state space. This is likely one reason *why* they grow more in certain environments. Figure 9.4 plots the proportion of input space indexed by champion policy graphs throughout development, where this naturally correlates with the policy graph development shown in Figure 9.3. Thus, the emergent developmental approach to model building in TPG can also be examined from the perspective of the efficiency with which information from the state space, $\vec{s}(t)$ is employed. In essence, TPG policies have the capacity to develop their own Adapted Visual Fields (AVF).

For example, Figure 9.5 shows the Adapted Visual Field (AVF) of champion

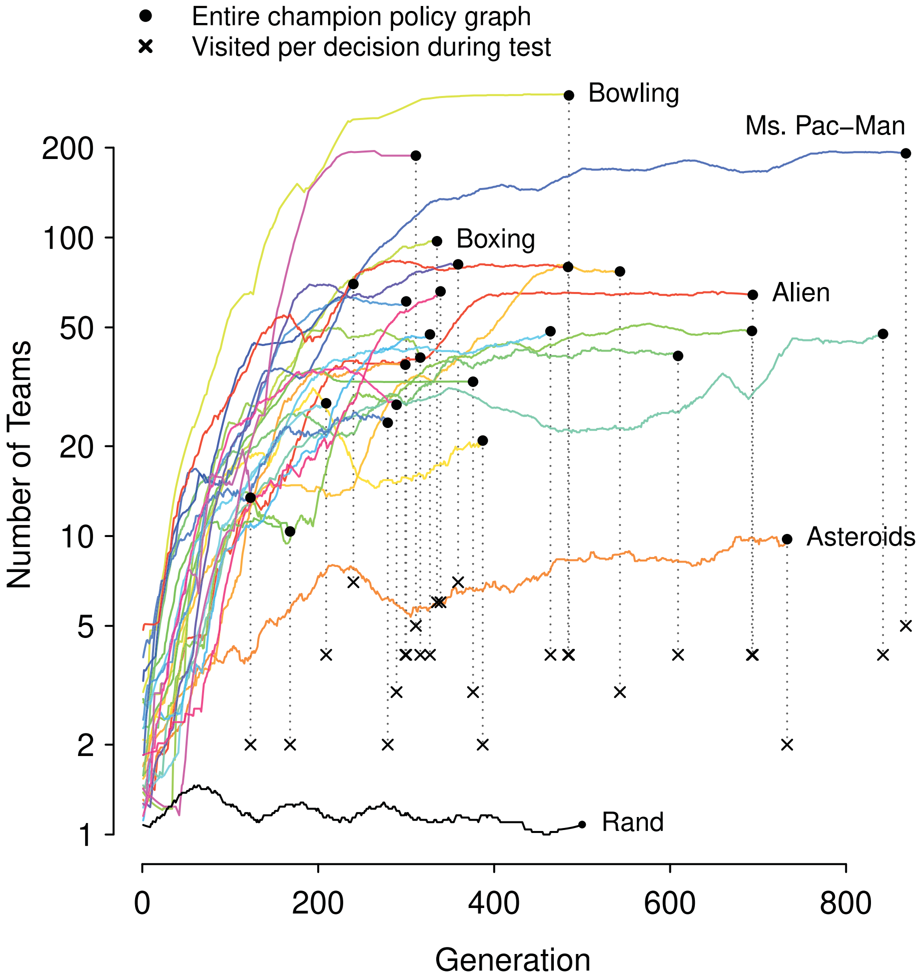


Figure 9.3: Development of the number of teams per champion TPG policy graph as a function of generation and game title. The run labeled 'Rand' reflects the number of teams per policy when selection pressure is removed, confirming that module emergence is driven by selective pressure rather than drift or other potential biases. Black circles indicate the total number of teams in each champion policy, while × indicates the average number of teams visited to make each single decision during test. For clarity only the 27 game titles with TPG agent performance ≥ DQN are depicted.

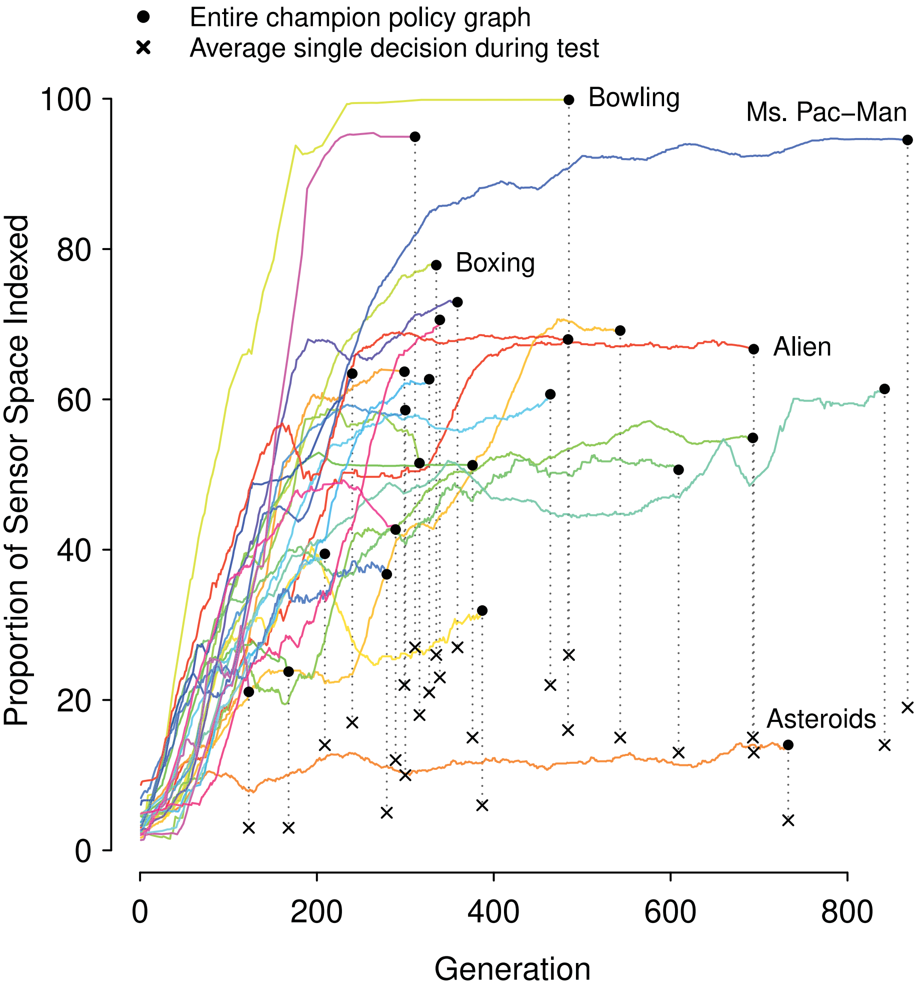
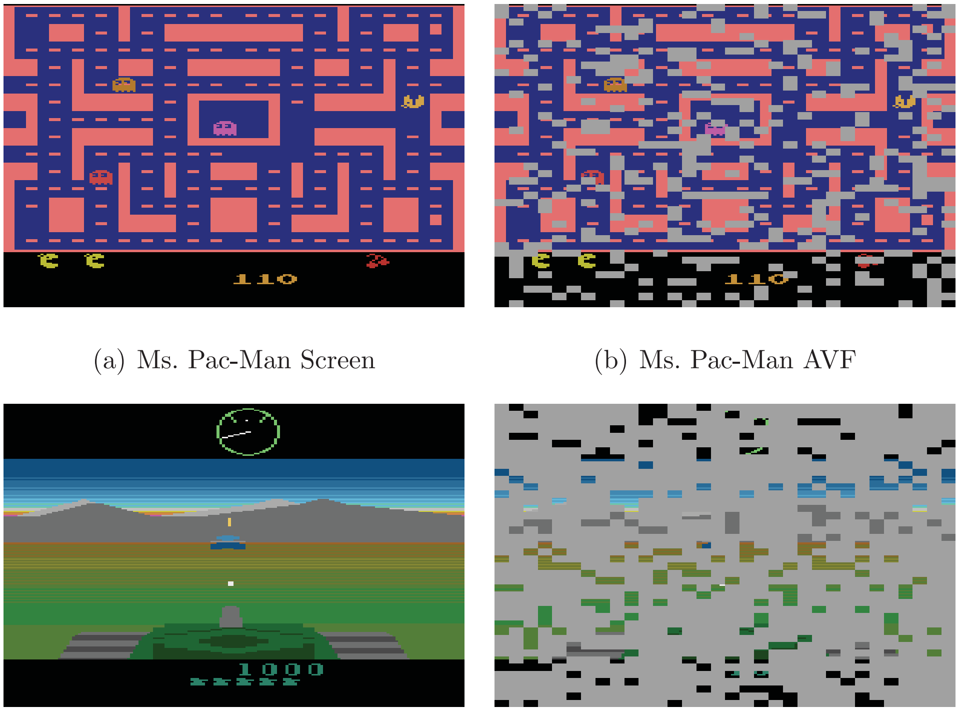


Figure 9.4: Development of the proportion of input space indexed by champion TPG policies. Black circles indicate the total proportion indexed by each champion policy, while × indicates the average proportion observed to make each single decision during test. For clarity only the 27 game titles with TPG agent performance ≥ DQN are depicted.



(a) Ms. Pac-Man Screen

(b) Ms. Pac-Man AVF

(c) Battle Zone Screen

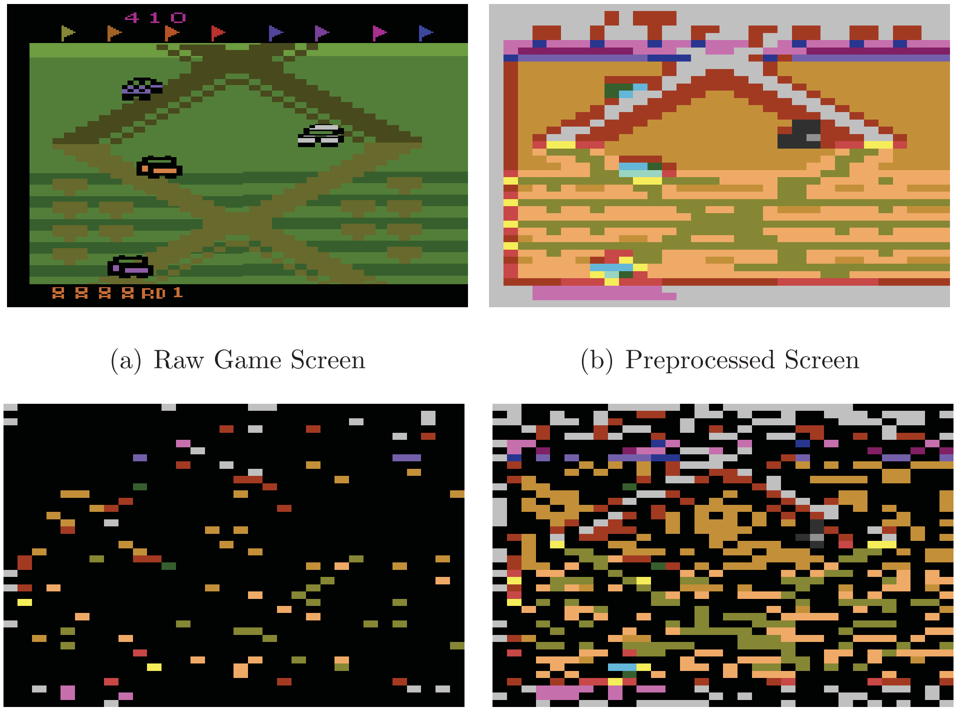
(d) Battle Zone AVF

Figure 9.5: Adapted Visual Field (AVF) of champion TPG policies in Ms.Pac-Man and Battle Zone. Grey regions indicate areas of the screen not indexed by the policy.

TPG agents in Ms. Pac-Man and Battle Zone. In the case of Ms. Pac-Man, the game defines a 2-dimensional maze environment that the player navigates in order to collect pills, where the pills are evenly distributed throughout the maze. Relatively high resolution is required in order to distinguish objects such as the agent's avatar and pills from the maze walls, and near-complete screen coverage is required to locate all the active pills and guide the avatar to/from any maze location. On the other hand, Battle Zone is a first person shooter game in which the agent can swivel left or right to position targets at centre-screen before shooting. While even screen coverage is helpful in locating targets and determining the direction to swivel, targets are large and thus low-resolution coverage is sufficient. Interestingly, Battle Zone includes a high-resolution global radar view in the top centre of the screen, which the champion policy's low-resolution AVF did not make efficient use of. Nonetheless, the bare-bones policy was able to out-perform DQN (and the human video game tester from [100]) without this advantage.

While the proportion of the visual field (input space) covered by a policy's AVF scales to the task environment, ranging from about 10% (Asteroids) to 100% (Bowling), the average proportion required to make each decision remains low, or less than 30% (See × symbols Figure 9.4). Figure 9.6 provides an illustration of the AVF as experienced by a single TPG team (c) versus the AVF for an entire champion TPG policy graph (d) in the game "Up 'N Down". This is a driving game in which the player steers a Dune Buggy along a track that zig-zags vertically up and down the screen. The goal is to collect flags along the route and avoid hitting other vehicles. The player can smash opponent cars by jumping on them using the joystick fire button, but looses a turn upon accidentally hitting a car or jumping off the track. TPG was able to exceed the level of DQN in Up and Down (test games consistently ended due to the 18,000 frame limit rather than agent error) with a policy graph that indexed only 42% of the screen in total, and an average 12% of the screen per decision (See column %SP in Table 9.3). The zig-zagging patterns that constitute important game areas are clearly visible in the policy's AVF. In this case, the policy *learned* a simplified sensor representation well tailored to the task environment. It is also apparent that in the case of the single TPG team, the AVF does not index state information from a specific local region, but instead samples from a diverse spatial range across the entire image (Figure 9.6(c)).

The proportion of visual input space used for decision-making under test is included within a more detailed discussion of complexity in Section 9.5.6. In order to provide more detail relative to Adapted Visual Fields, the reader is referred ahead to column %SP in Table 9.3, which gives the percent of state space (screen) indexed by the policy as a whole and per decision. Maze tasks, in which the goal involves directing an avatar through a series of 2-D mazes (eg. Bank Heist, Ms. Pac-Man, Venture) typically require near-complete screen coverage in order to navigate all regions of the maze, and relatively high-resolution is important to distinguish various game entities from maze walls. However, while the policy as a whole may index most of the screen, the modular nature of the representation implies that no more than 27% of the indexed space is considered before making each decision (Table 9.3, Column %SP), significantly improving the run-time complexity of the policy. Furthermore, adapting the visual field implies that extensive screen coverage is only used when



(a) Raw Game Screen

(b) Preprocessed Screen

(c) Single Team AVF

(d) Policy AVF

Figure 9.6: Adapted Visual Field (AVF) of champion TPG policy graph in Up 'N Down. Black regions indicate areas of the screen not indexed by the policy. (a) shows the raw game screen. (b) shows the preprocessed state space, where each decimal state variable (0-255) is mapped to a unique colour. (c) shows the AVF for a single team along the active path through the policy graph at this time step, while (d) shows the AVF for the policy graph as a whole. Both AVFs exhibit patterns of sensitivity consistent with important regularities of the environment, specifically the zig-zagging track.

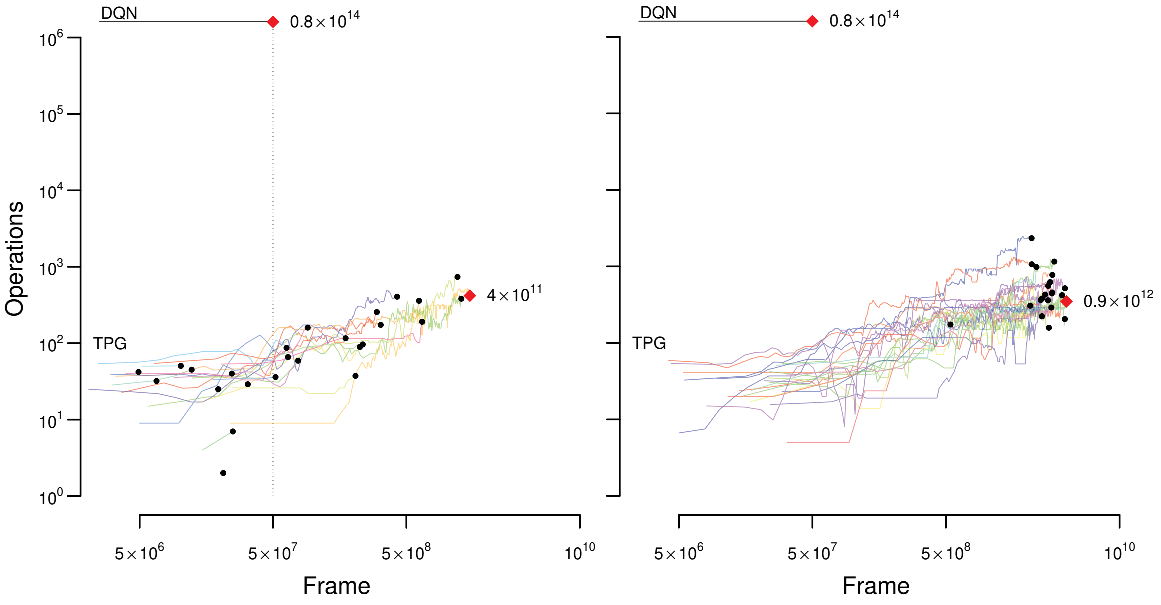
necessary. Indeed, in 10 of the 27 games for which TPG exceeded the score of DQN, it did so while indexing less that 50% of the screen, further minimizing the number of instructions required per decision.

9.5.5 Computational Cost

This section investigates the issue of computational cost to build solutions, while Section 9.5.6 will consider the cost of decision-making post training. The budget for model building in DQN was to assume a fixed number of decision-making frames per game title (50 million). The cost of making each decision in deep learning is also fixed a priori, a function of the preprocessed image (Section 9.5.5) and the complexity of a multi-layer preceptron (MLP). Simply put, the former provides an encoding of the original state space into a lower dimensional space, the latter represents the decision-making mechanism. Appendix E provides a detailed calculation of DQN's computational complexity.

As noted in Section 9.4, TPG runs are limited to a fixed computational time of 2 weeks per game title. However, under TPG the cost of decision-making is variable as solutions do not assume a fixed topology. We can now express computational cost in terms of the cost to reach the DQN performance threshold (27 game titles), and the typical cost over the two week period (remaining 21 game titles). Specifically, let T be the generation at which a TPG run exceed the performance of DQN. $P(t)$ denotes the number of policies in the population at generation t. Let $i(t)$ be the average number of instructions required by each policy to make a decision, and let $f(t)$ be the total number of frames observed over all policies at generation t, then the total number of operations required by TPG to discover a decision-making policy for each game is $\sum_{t=1}^{T} P(t) \times i(t) \times f(t)$. When viewed step-wise, this implies that computational cost can increase or decrease relative to the previous generation, depending on the complexity of evaluating TPG individuals (which are potentially all unique topologies).

Figure 9.7 plots the number of instructions required for each game over all decision-making frames observed by agents during training. Figure 9.7(a) characterizes computational cost in terms of solutions to the 27 game titles that reached the DQN performance threshold, i.e. the computational cost of reaching the DQN performance



(a)

(b)

Figure 9.7: Number of operations per frame (y-axis) over all game frames observed during training (x-axis). (a) shows the subset of games up to the point where TPG exceeded DQN test score. (b) shows games for which TPG did not reach DQN test score. Red diamonds denote the most complex cases, with text indicating the cumulative number of operations required to train each algorithm up to that point. DQN's architecture is fixed a priori, thus cumulative computational cost at each frame is simply a sum over the number of operations executed up to that frame. TPG's complexity is adaptive, thus produces a unique development curve and max operations for each game title. Frame limit for DQN was 50 million (5×10^7). Frame limit for TPG, imposed by a cluster resource time constraint of 2 weeks, is only reached in (b).

threshold. Conversely, Figure 9.7(b) illustrates the computational cost for games that never reached the DQN performance threshold, i.e. terminated at the 2-week limit. As such this is representative of the overall cost of model building in TPG for the ALE task given a two week computational budget. In general, cost increases with an increasing number of (decision-making) frames, but the cost benefit of the non-monotonic, adaptive nature of the policy development is also apparent.

It is also readily apparent that TPG typically employed more than the DQN budget for decision-making frames (5×10^7). However, the cost of model construction is also a function of the operations per decision. For example, the parameterization adopted by DQN results in an MLP hidden layer consisting of $1,605,632$ weights, or a total computational cost in the order of 0.8×10^{14} over all $50,000,000$ training frames. The total cost of TPG model building is 4×10^{11} in the worst case (Figure 9.7(a)). Thus, the cost of the MLP step, *without* incorporating the cost of performing the deep learning convolution encoding ($>$ 3 million calculations at layer 1 for the parameterization of [100]), exceeds TPG by several orders of magnitude. Moreover, this does not reflect the additional cost of performing a single weight update, i.e. the backward pass.

9.5.6 Cost of Real-Time Decision Making

Table 9.2 summarizes the cost / resource requirement when making decisions post training, i.e. the cost of deploying an agent. Liang *et al.* report figures for the memory and wall clock time of Blob-PROST on a 3.2GHz Intel Core i7-4790S CPU [87]. Computational cost for DQN is essentially static due to a fixed architecture being assumed for all games. Blob-PROST complexity is a function of the diversity of colour pallet in the game title. Apparently the 9GB figure was the worst case, with 3.7GB representing the next largest memory requirement. It is apparent that TPG solutions are typically 2 to 3 orders of magnitude faster than DQN and an order of magnitude faster than Blob-PROST.

TPG model complexity is an evolved trait (Section 9.5.3) and only a fraction of the resulting policy graph is ever visited per decision. Table 9.3 provides a characterization of this in terms of three properties of champion team for each game title:

- Teams (Tm) – both the total number of teams per champion and corresponding

Table 9.2: Wall clock time for making each decision and memory requirement. † Values for TPG reflect the memory utilized to support the *entire* population whereas only one champion agent is deployed post training, i.e. tens to hundreds of kilobytes. TPG wall-clock time is measured on a 2.2GHz Intel Xeon E5-2650 CPU.

Method	Decisions per sec	Frames per sec	Memory
DQN	5	20	9.8 GB
Blob-PROST	56 – 300	280 – 1500	50MB – 9GB
TPG	758 – 2853	3032 – 11412	118MB – 146MB†

average number of teams visited per decision.

- Instructions per decision (Ins) – the average number of instructions executed per agent decision. Note that as a linear genetic programming representation is assumed, most intron code can be readily identified and skipped for the purposes of program execution [20]. Thus, 'Ins' reflects the code actually executed.

- Proportion of visual field (%SP) – the proportion of the state space (Section 9.3) indexed by the entire policy graph versus that actually indexed per decision. This reflects the fact that GP individuals, unlike deep learning or Blob-PROST, are never forced to explicitly index all of the state space. Instead the parts of the state space utilized per program is an emergent property (discussed in detail in Section 9.5.4).

It is now apparent that on average only 4 teams require evaluation per decision (parenthesis value in Tm column, Table 9.3). This also means that decisions are typically made on the basis of $3 - 27\%$ of the available state space (parenthesis value in %SP column, Table 9.3). Likewise, the number of instructions executed is strongly dependent on the game title. The TPG agent for Time Pilot executed over a thousand instructions per action, whereas the TPG agent for Asteroids only executed 96. In short, rather than having to assume a fixed decision-making topology with hundreds of thousands of parameters, TPG is capable of discovering emergent representations appropriate for each task.

Table 9.3: Characterizing overall TPG complexity. Tm denotes the total number of teams in champions versus the average number of teams visited per decision (value in parenthesis). Ins denotes the average number of instructions executed to make each decision. %SP denotes the total proportion of the state space covered by the policy versus (value in parenthesis).

Title	Tm	Ins	%SP	Title	Tm	Ins	%SP
Alien	67(4)	498	68(13)	Amidar	132(6)	1066	87(23)
Assault	37(4)	420	50(14)	Asterix	77(5)	739	73(20)
Asteroids	7(2)	96	10(4)	Atlantis	39(4)	939	64(22)
Bank Heist	94(3)	532	75(15)	Battle Zone	15(2)	191	24(6)
Beam Rider	115(4)	443	83(13)	Bowling	300(4)	927	100(26)
Boxing	102(6)	1156	79(26)	Breakout	6(3)	158	8(6)
Centipede	36(4)	587	48(18)	C. Command	49(4)	464	54(15)
Crazy Climber	150(3)	1076	99(28)	Demon Attack	19(3)	311	26(8)
Double Dunk	10(2)	98	20(3)	Enduro	24(3)	381	37(10)
Fishing Derby	33(3)	472	50(15)	Freeway	18(4)	296	26(11)
Frostbite	45(4)	434	53(13)	Gopher	4(2)	156	8(5)
Gravitar	49(4)	499	62(14)	H.E.R.O	96(5)	979	75(24)
Ice Hockey	29(4)	442	39(14)	James Bond	41(4)	973	59(22)
Kangaroo	52(4)	877	64(21)	Krull	62(4)	376	58(10)
Kung-Fu Master	31(2)	137	44(5)	M's Revenge	403(2)	722	100(22)
Ms. Pac-Man	197(5)	603	95(19)	Name This Game	93(3)	361	77(13)
Pong	12(4)	283	20(10)	Private Eye	71(7)	761	64(17)
Q*Bert	255(8)	2346	99(46)	River Raid	7(3)	286	15(9)
Road Runner	86(7)	1169	74(27)	Robotank	42(2)	252	47(8)
Seaquest	58(4)	579	60(15)	Space Invader	68(4)	624	65(17)
Star Gunner	17(4)	516	35(15)	Tennis	3(2)	71	5(3)
Time Pilot	189(5)	1134	95(27)	Tutankham	36(2)	464	58(14)
Up and Down	28(3)	425	42(12)	Venture	77(6)	1262	74(23)
Video Pinball	38(3)	399	55(13)	Wizard of Wor	23(4)	433	36(12)
Zaxxon	81(4)	613	68(16)				

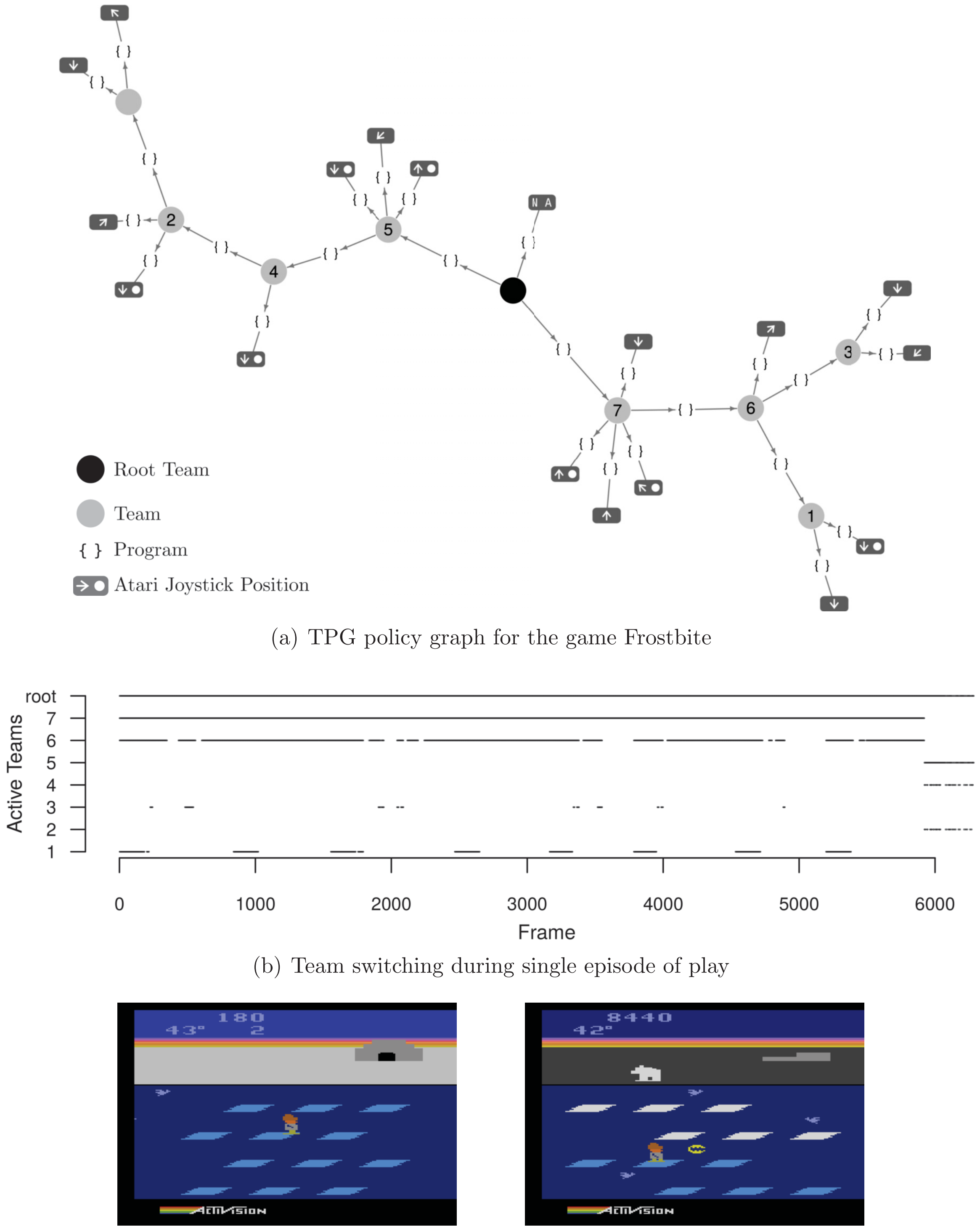
9.5.7 Characterization of Learned Strategies

Files `TPG-Up_N_Down.mp4` and `TPG-Frostbite.mp4` included with this thesis provide examples of how a TPG policy interacts with Atari games. The animations show the active path through the graph at each time step (Right), along with the preprocessed state variables indexed by all teams along the active path (Bottom Left), and the raw game screen (Upper Left). As the game state changes, a different path through the graph is used to select an atomic action, thus a different subset of the state variables are also used (Bottom Left).

As described in Section 9.5.4, the game Up 'N Down is a driving game in which the player steers a Dune Buggy along a track that zig-zags vertically up and down the screen. The goal is to collect flags along the route and avoid hitting other vehicles. The player can smash opponent cars by jumping on them using the joystick fire button, but looses a turn upon accidentally hitting a car or jumping off the track. The Up / Down actions increment or decrement the dune buggy's acceleration in the given direction, while the addition of the Fire button causes the dune buggy to jump. The champion policy graph for Up 'N Down uses only Fire, Up-Fire, and No-Action in defining a simple yet effective policy, which exceeds the test score of DQN, but not Hyper-NEAT (Table D.1, Appendix D). This policy graph contains 28 teams[8] and indexes 42% of the state space. However, on average only 3 teams, 425 instructions, and 12% of the available state variables are used to make each decision (See Table 9.3). Thus, a division of labor among the programs in the graph effectively decomposes the task. The resulting strategy throttles the upward acceleration of the dune buggy by toggling Up-Fire and No-Action, gaining points by jumping on other cars to destroy them but never loosing a life by hitting them head-on. The policy also manages to collect most of the flags and only rarely looses a life by leaving the track. Indeed, the episode ends due to the $18,000$ frame limit rather than agent error.

The objective in the game Frostbite is to build igloos by jumping on floating blocks of ice, while trying to avoid hazards such as clams and polar bears. The champion policy graph in Frostbite is slightly more complex than in Up 'N Down, but the cost per decision is similar. The policy graph contains 45 teams and indexes 53% of the state space. However, on average only 4 teams, 434 instructions, and

[8] Note that only the graph nodes that were active during *this game episode* are pictured.



(a) TPG policy graph for the game Frostbite

(b) Team switching during single episode of play

(c) Screen colours frames 0 - 5922

(d) Screen colours frames > 5922

Figure 9.8: Temporal switching of TPG policy substructures (teams) during gameplay. The champion policy graph for the game Frostbite is illustrated in (a), where only teams that contributed to decision-making during test are shown. (b) plots which teams contributed to each decision (frame) over a single episode of play. The environment's colour scheme before and after frame 5922 are shown in (c) and (d), an environmental change that activates the left side of the policy graph (teams 5, 4, and 2).

13% of the available state variables are used to make each decision (See Table 9.3). The policy defines an effective long-term strategy, using a variety of atomic actions to move around the environment while jumping on ice blocks. As soon as the igloo is complete, the agent navigates directly inside to accumulate a wealth of points. As a complement to the video `TPG-frostbite.mp4`, Figure 9.8 provides an illustration of the temporal switching of policy substructures during a single episode of play in the game Frostbite. Interestingly, for the first ≈ 3.5 minutes of play (5922 frames) only one region of the policy graph is active (right-side of the root team, or teams 7, 6, 3, and 1). During this time, team 7 switches between a variety of atomic actions and often defers decision-making to team 6, which in turn switches between atomics and teams 3 and 1. At about 3:37 in the video, the colour scheme of the environment changes and the policy begins using the rest of its graph (left of the root node, or teams 5,4, and 2). This is an example of how the modular structure of TPG's representation allows a policy to combine and organize a variety of decision-making structures, recalling only those relevant to the current environmental conditions.

As in the frostbite example, it is common for a specific substructure within a policy graph to be active across contiguous states in a decision-making sequence. This implies that a higher-level team has learned which states to map to this particular substructure, or using the terminology of the options framework for temporal abstractions [137], the higher-level team learns the *initiation set* and *termination condition* for the substructure. Furthermore, complete structures of this nature, which now consist of the three components necessary for temporally abstract knowledge (i.e. a policy, an initiation set, and a termination condition) may be exchanged between organisms through team variation operators during evolution. This allows credit assignment to identify policy substructures responsible for temporally-extended decision-making. (Indeed, the substructure consisting of teams 5, 4, and 2 in Figure 9.8(a) appeared in 19 other teams from the same population and was used for the same temporally contiguous region of the state space). These preliminary observations suggest that the representation supports the evolution of temporal abstractions (without a prior definition of subgoals). More research is required to determine how frequently this occurs and how beneficial it actually is to the learning process, but it is worth noting that TPG test scores in Frostbite, Ms. Pac-Man, and Alien were competitive with the maximum (training) score reported by another HRL model that explicitly addressed the temporal credit assignment problem [155].

9.6 Multi-Task Learning

9.6.1 Overview

Up to this point this chapter has demonstrated the ability of TPG to build strong single-task decision-making policies, where a unique policy graph was trained from scratch for each Atari game title. This section reports on TPG's ability to learn multiple tasks simultaneously, or Multi-task Reinforcement Learning (MTRL). MTRL operates with the same state representation as single-task learning. That is, state variables consist of raw screen capture with no additional information regarding which game is currently being played. Furthermore, the full Atari action set is available to agents at all times. Previous work has established the capability of TPG to learn 3 Atari games simultaneously using MTRL [73]. This section extends that work by simplifying the MTRL methodology and increasing the task set to 5 Atari titles. When the TPG population is trained on multiple Atari games simultaneously, a single run can produce multiple champion policies, one for each game title, that match or exceed the level of play reported for DQN. In some cases, a multi-task policy (i.e. a single policy graph capable of playing multiple titles) also emerges that plays *all* games at the level of DQN. Furthermore, the training cost for TPG under MTRL is no greater than task-specific learning, and the complexity of champion multi-task TPG policies is still significantly less than task-specific solutions from deep learning.

9.6.2 Task Groups

In order to investigate TPG's ability to learn multiple Atari game titles simultaneously, a variety of task groupings, i.e. specific game titles to be learned simultaneously, are created from the set of games for which single-task runs of TPG performed well. Relative to the four comparison algorithms which use a screen capture state representation, TPG achieved the best reported test score in 15 of the 49 Atari game titles considered (Table D.1, Appendix D). Thus, task groupings for MTRL can be created in an unbiased way by partitioning the list of 15 titles in alphabetical order.

Table 9.4: Task groups used in multi-task reinforcement learning experiments. Each group represents a set of games to be learned simultaneously. See text for an explanation of the groupings.

3-Title Groups	Game	5-Title Groups
3.1	Alien	5.1
	Asteroids	
	Bank Heist	
3.2	Battle Zone	
	Bowling	
	Centipede	5.2
3.3	Chopper Command	
	Fishing Derby	
	Frostbite	
3.4	Kangaroo	
	Krull	5.3
	Kung-Fu Master	
3.5	Ms. Pac-Man	
	Private Eye	
	Time Pilot	

Specifically, Table 9.4 identifies 5 groups of 3 games each and 3 groups of 5 games each.

Importantly, while it is possible to categorize Atari games by hand in order to support incremental learning [22], no attempt was made here to organize game groups based on perceived similarity or multi-task compatibility. Such a process would be labour intensive and potentially misleading, as each Atari game title defines its own graphical environment, colour scheme, physics, objective(s), and scoring scheme. Furthermore, joystick actions are not necessarily correlated between game titles. For example, the 'down' joystick position generally causes the avatar to move vertically down the screen in maze games (eg. Ms. Pac-Man, Alien), but might be interpreted as 'pull-up' in flying games (Zaxxon), or even cause a spaceship avatar to enter hyperspace, disappearing and reappearing at a random screen location (Asteroids). However, even with such diversity in the nature of Atari games, it is useful to test for inter-game regularities in each game group. To do so, champion TPG policies as developed and reported in Section 9.5.2 are re-evaluated over all games within each group, with the resulting test scores shown in Figure 9.9 (3-title groups) and Figure 9.10 (5-title groups). Each policy in Figures 9.9 and 9.10 was exposed to a single game title during evolution and is tested on the other titles within the group in which it appears. With one exception, no policy is able to achieve significant scores in any game other than the title it experienced during training. The exception to this is the policy trained for Time Pilot, which is able to exceed the level of DQN in the game Private Eye with no prior training experience in that game title (See Figures 9.9(e) and 9.10(c)). However, the policy trained for Private Eye does *not* exhibit any significant ability in Time pilot. Thus, with the exception of $TimePilot \rightarrow PrivateEye$, proficiency in any single game title does not directly transfer to any other game within the same group.

9.6.3 Task Switching

As in single-task learning, each policy is evaluated in 5 episodes per generation. However, under MTRL, new policies are first evaluated in one episode under each game title in the current task group. Thereafter, the game title for each training episode is selected with uniform probability from the set of titles in the task group. The maximum training episodes for each policy is 5 episodes under each game title. For each consecutive block of 10 generations, one title is selected with uniform probability to be the *active* title for which selective pressure is applied. Thus, while a policy may store the final score from up to 5 training episodes for each title, fitness at any given generation is the average score over up to 5 episodes in the *active title only*. Thus, selective pressure is only explicitly applied relative to a single game title. However, stochastically switching the active title at regular intervals throughout evolution implies that a policy's long-term survival is dependent on a level of competence in *all* games.

9.6.4 Elitism

There is no multi-objective fitness component in the formulation of MTRL proposed in this work. However, a simple form of elitism is used to ensure the population as a whole never entirely forgets any individual game title. As such, the single policy with the best average score in each title is protected from deletion, regardless of which title is currently active for selection. Note that this simple form of elitism does not protect multi-task policies, which may not have the highest score for any single task, but *are*

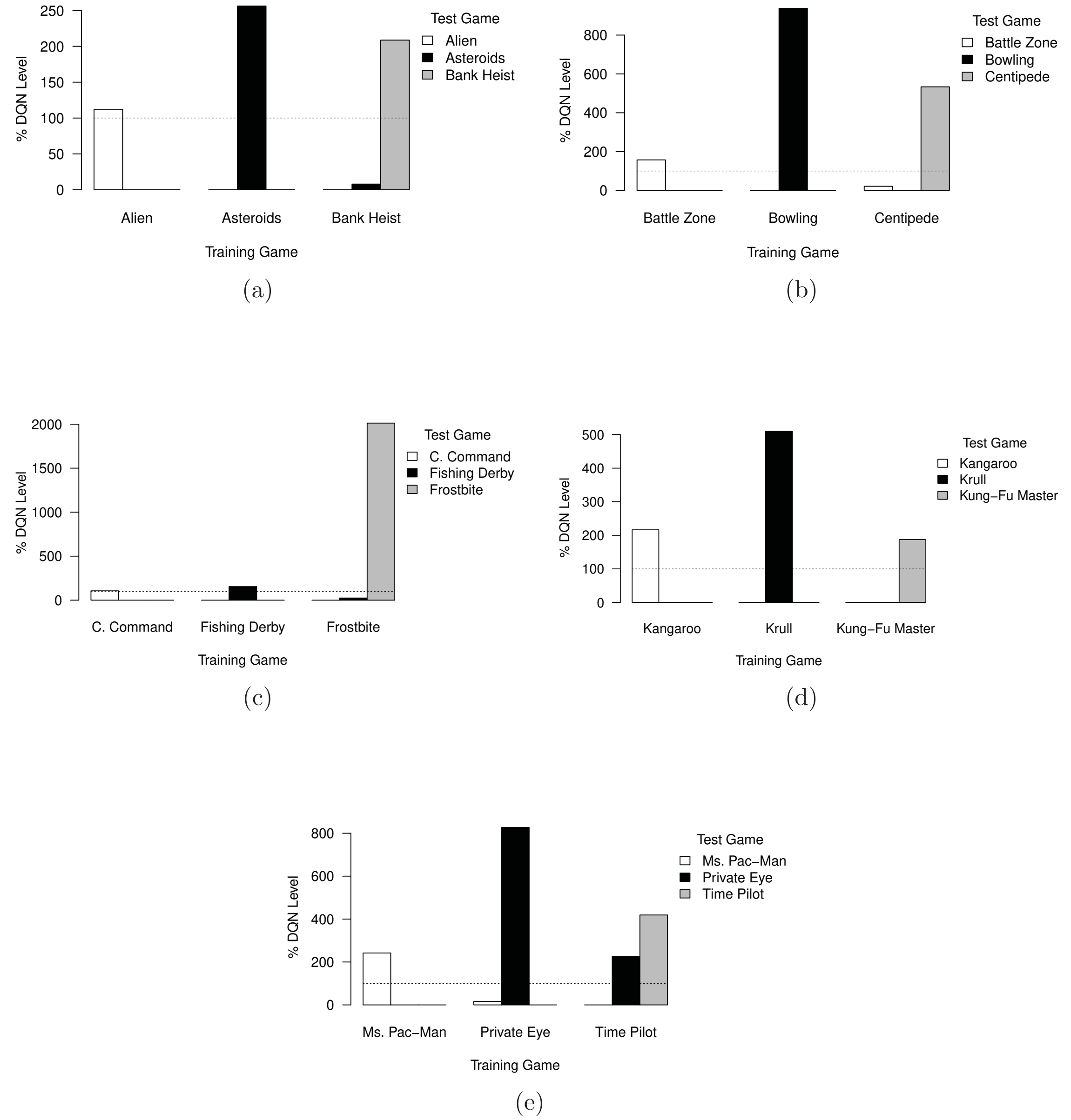
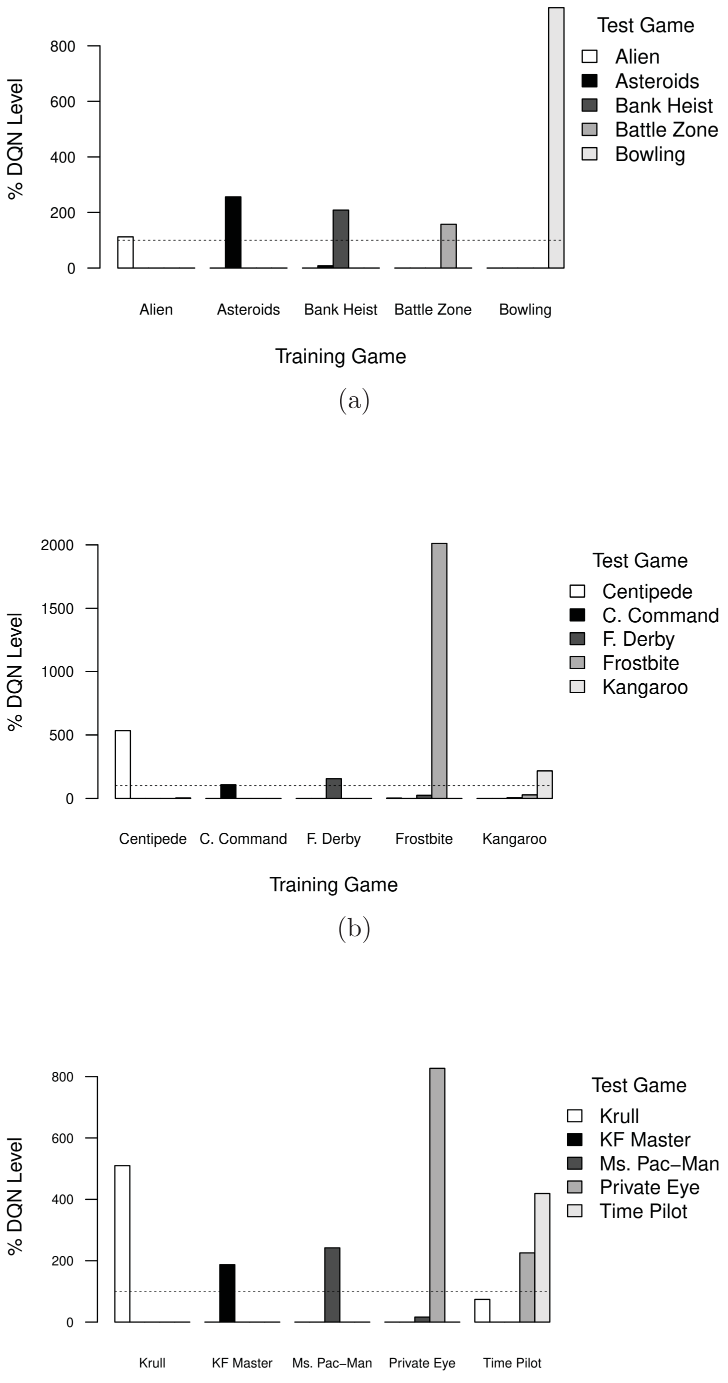


Figure 9.9: Test results without MTRL for 3-title game groups. The champion (game-specific) TPG policy for each game from Section 9.5.2 is re-tested in 30 episodes under each game title and mean scores are reported. x-axis indicates which game the policy was *trained* for, while shading of bars indicates the active game under test. (i.e. Each policy was trained for *one* game and has never seen the other two before). All scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). DQN scores are from [100].



(a)

(b)

(c)

Figure 9.10: Test results without MTRL for 5-title game groups. The champion (game-specific) TPG policy for each game from Section 9.5.2 is re-tested in 30 episodes under each game title and mean scores are reported. x-axis indicates which game the policy was *trained* for, while shading of bars indicates the active game under test. (i.e. Each policy was trained for *one* game and has never seen the other four before). All scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). DQN scores are from [100].

able to perform relatively well on multiple tasks. Failing to protect multi-task policies became problematic under the methodology of our first MTRL study [73]. Thus, a simple form of multi-task elitism is employed in this work. The elite multi-task team is identified in each generation using the following 2-step procedure:

1. Normalize each policy's mean score on each task relative to the rest of the current population. Normalized score for team tm_i on task t_j, or $sc^n(tm_i, t_j)$, is calculated as $(sc(tm_i, t_j) - sc_{min}(t_j))/(sc_{max}(t_j) - sc_{min}(t_j))$, where $sc(tm_i, t_j)$ is the mean score for team tm_i on task t_j and $sc_{min,max}(t_j)$ are the population-wide min and max mean scores for task t_j.

2. Identify the multi-task elite policy as that with the highest minimum normalized score over all tasks. Relative to all root teams in the current population, R, the elite multi-task team is identified as $tm_i \in R \mid \forall tm_k \in R : min(sc^n(tm_i, t_{\{1..n\}}) > min(sc^n(tm_k, t_{\{1..n\}})$, where $min(sc^n(tm_i, t_{\{1..n\}})$ is the minimum normalized score for team tm_i over all tasks in the game group and n denotes the number of titles in the group.

Thus, in each generation, elitism identifies 1 champion team for each game title and 1 multi-task champion, where elite teams are protected from deletion in that generation.

9.6.5 Detailed Results

The parameterization used for TPG under multi-task reinforcement learning is identical to that described in Table 9.1 with the exception of R_{size} parameter, or the number of root teams to maintain in the population. Under MTRL, the population size was reduced to 90 (1/4 of the size used under single-task learning) in order to speed up evolution and allow more task switching cycles to occur throughout the given training period[9]. A total of 5 independent runs were conducted for each task group in Table 9.4. Multi-task elite teams represent the champions from each run at any point during development. Post training, the final champions from each run are subject to the same test procedure as identified in Section 9.2 for each game title.

[9] As under single-task experiments, the computational limit for MTRL is defined in terms of a computational resource time constraint. Experiments ran on a shared cluster with a maximum runtime of 1 week per run.

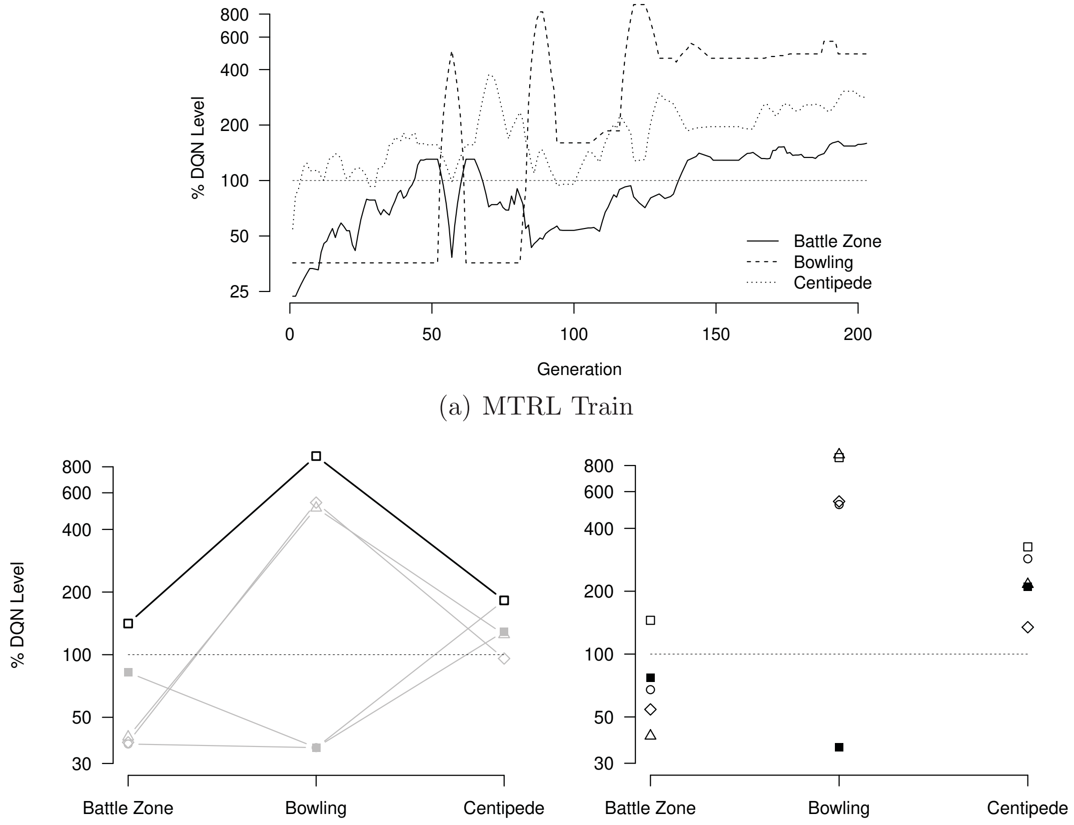
Figure 9.11 reports the training and test performance for task group 3.2, where all TPG scores are normalized relative to scores reported for DQN in [100]. By generation ≈ 140, the best multi-task policy is able to play all 3 game titles at the level reported for DQN[10]. The runs were terminated at generation ≈ 200, although multi-task policies were still improving for most tasks. Under test, the multi-task champion (ie. a single policy that plays all game titles at a high level) exceeds DQN in all 3 games titles, Figure 9.11(b). Note that in the case of task group 3.2, only one run managed to produce a multi-task champion of this quality (highlighted in black in Figure 9.11(b)), while most other runs produced multi-task champions that play 2/3 games at the level of DQN (grey data points in Figure 9.11(b)).

While the primary focus of MTRL is to produce multi-task policies, a byproduct of the methodology employed here (i.e. task-switching and elitism rather than multi-objective methods) is that each run also produces high-quality single-task policies (i.e. policies that excel at one game title). Test results for these game-specific specialists, which are simply the 3 elite single-task policies at the end of evolution, is reported in Figure 9.11(c). While not as proficient as policies trained on a single task (Section 9.5.2), at least one single-task champion emerges from MTRL in task group 3.2 that matches or exceeds the score from DQN in each game title.

Figure 9.12 reports the MTRL training and test performance for TPG relative to game group 5.3. In this case, the training performance of the single best multi-task policy matches or exceeds the level of DQN in all 5 game titles by generation ≈ 750, Figure 9.12(a). Under test, the multi-task champion exceeds DQN in 4 of the 5 games, while reaching over 90% of DQN's score in the remaining title (Krull), Figure 9.12(b). Again, only one run produced a multi-task policy capable of matching DQN in all 5 tasks. However, all runs produce a single-task champion for all game titles that matches or exceeds the level of DQN, Figure 9.12(c)

Detailed multi-task training and test results for TPG under the remaining game groups (Table 9.4) are included in Appendix F. Table 9.5 provides a summary overview of test scores for the champion multi-task and single-task policy relative to each game group. Test scores that match or exceed that of DQN are highlighted in grey. For the 3-title groups, TPG produced multi-task champions capable of playing

[10] Note that training scores reported for TPG in Figure 9.11(a) are averaged from a max of 5 episodes in each game title, and are thus not as robust as the test scores reported in Figure 9.11(b)

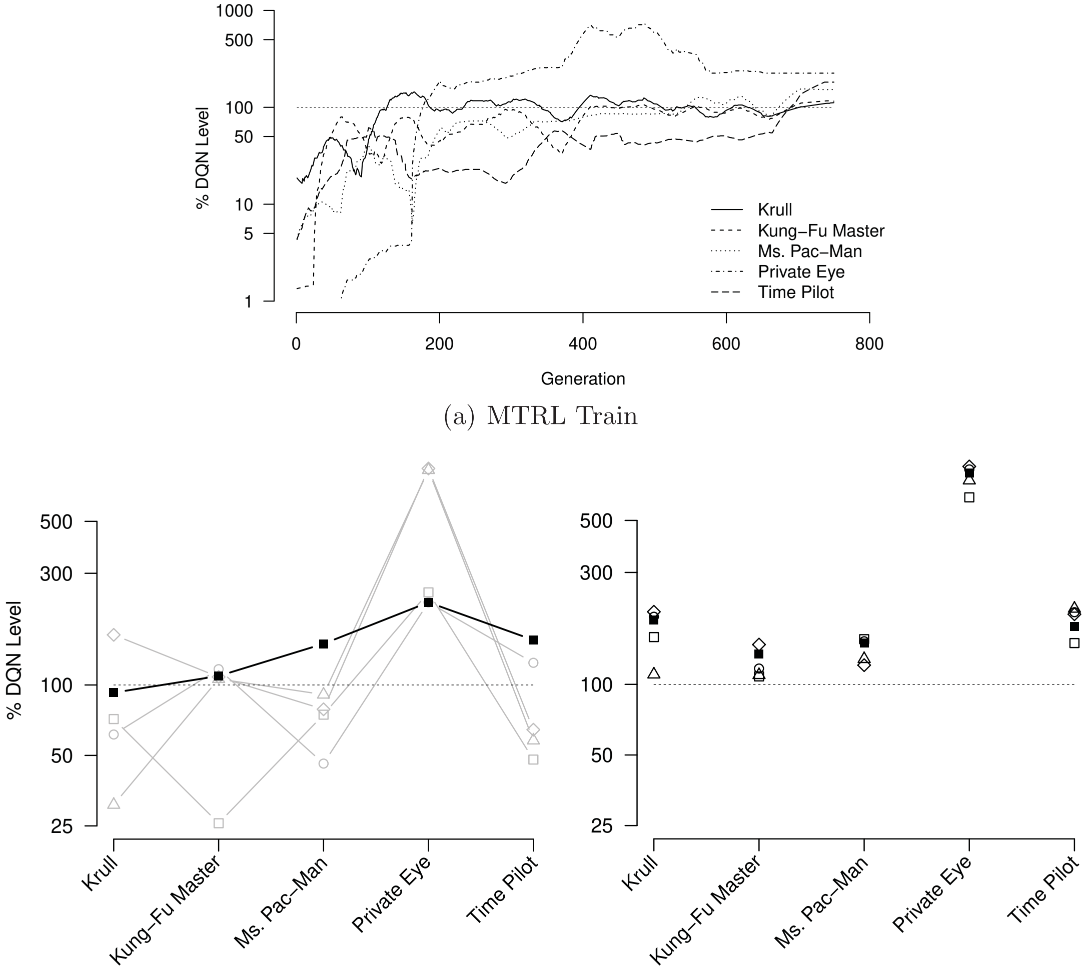


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure 9.11: TPG multi-task reinforcement learning results for task group 3.2. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].



(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure 9.12: TPG multi-task reinforcement learning results for game group 5.3. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

Table 9.5: Summary of multi-task learning results over all task groups. MT and ST report test scores for the single best multi-task (MT) and single-task (ST) policy for each game group over all 5 independent runs. Scores that match or exceed the test score reported for DQN in [100] are highlighted in grey (The MT score for Krull in group 5.3 is 90% of DQN's score, and is considered a match).

MT	ST	Group	Game	Group	MT	ST
864	1494.3	3.1	Alien	5.1	346.7	759
2176	2151		Asteroids		1707	2181.3
1085	1085		Bank Heist		724	724
36166	37100	3.2	Battle Zone		11800	30466.7
197	197		Bowling		107	212
13431.2	22374.6		Centipede	5.2	9256.9	20480.2
3173.3	3266.7	3.3	Chopper Command		1450	2716.7
-66.6	-38.4		Fishing Derby		24.967	27.7
2900.7	4216		Frostbite		2087.3	4283.3
11200	10940	3.4	Kangaroo		11200	11893.3
4921.3	17846.7		Krull	5.3	3644.3	6099.7
25600	42480		Kung-Fu Master		25393.3	34273.3
3067.7	3164.7	3.5	Ms. Pac-Man		3312.3	3430
14665	14734.7		Private Eye		4000	15000
7846.7	8193.3		Time Pilot		7270	8570

all 3 games titles in groups 3.2, 3.4, and 3.5, while the multi-task champions learned 2/3 titles in group 3.1 and only 1/3 titles in group 3.3. For the 5-title groups, TPG produced multi-task champions capable of playing all 5 titles in group 5.3, 4/5 titles in group 5.2, and 3/5 titles in group 5.1. It seems that Alien and Chopper Command are two game titles that TPG had difficulty learning under the MTRL methodology adopted here (neither multi-task or single-task policies emerged for either game title). Interestingly, while Fishing Derby was difficult to learn when grouped with Frostbite and Chopper Command (group 3.3), adding 2 additional game titles to the task switching procedure (i.e. group 5.2) seems to have been helpful to learning Fishing Derby. Note that test scores from policies developed under the MTRL methodology are generally not as high as scores achieved through single-task learning for the same game titles (Section 9.5.2). This is primarily due to the extra challenge of learning multiple task simultaneously. However, it is important to note that the population size for MTRL experiments was 1/4 of that used for single-task experiments and the computational budget for MTRL was half that of single-task experiments. Indeed, the MTRL results here represent a proof of concept for TPG's multi-task ability rather than an exhaustive study of its full potential.

9.6.6 Modular Task Decomposition

Problem decomposition takes place at two levels in TPG: 1) Program-level, in which individual programs within a team each define a unique context for deploying a single action; and 2) Team-level, in which individual teams within a policy graph each define a unique program complement, and therefore represent a unique mapping from state observation to action. Moreover, since each program typically indexes only a small portion of the state space, the resulting mapping will be sensitive to a specific region of the state space. This section examines how modularity at the *team*-level supports the development of multi-task policies.

As TPG policy graphs develop, they will subsume an increasing number of stand-alone decision-making modules (teams) into a hierarchical decision-making structure. Recall from Section 8.4 that only root teams are subject to modification by variation operators. Thus, teams that are subsumed as interior nodes of a policy graph undergo no modification. This property allows a policy graph to avoid (quickly) unlearning tasks that were experienced in the past under task switching but are not currently the *active* task. This represents an alternative approach to avoiding ”catastrophic forgetting“ [77] during the continual, sequential learning of multiple tasks. The degree to which individual teams specialize relative to each objective experienced during evolution, i.e. the game titles in a particular game group, can be characterized by looking at which teams contribute to decision-making at least once during test, relative to each game title.

Figure 9.13 shows the champion multi-task TPG policy graph from the group 3.2 experiment (Figure 9.11). The Venn diagram indicates which teams are visited at least once while playing each game, over all test episodes. Naturally, the root team contributes to every decision (black circle in the graph, center of Venn diagram). 5 teams contribute to playing both Bowling and Centipede, while the rest of the teams specialize for a specific game title. In short, both generalist and specialist teams appear within the same policy and *collectively* define a policy capable of playing multiple game titles. Appendix G repeats the analysis for the case of Groups 3.4 and 3.5, where these correspond to the other two most effective multi-task solutions under 3 game titles (Table 9.5).

Figure 9.14 depicts the champion multi-task policy graph from the group 5.3 experiment (i.e. a single policy graph capable of playing 5 games titles). Not surprisingly, this policy is significantly more complex than the example 3-task policy graph (Figure 9.13). Of the 32 active teams in this graph, only two contribute to every task, while the rest specialize on between 1 and 4 tasks (8 different combinations in total). Krull (Game A in the graph) is the only title for which any team specializes exclusively. Naturally, teams that are active for Time Pilot (Game D in the graph) are almost always shared with Private Eye (game E in the graph), reflecting the known regularity between these titles (see Figure 9.10(c)).

Finally, the importance of team-level modularity to multi-task TPG development can be confirmed by testing the effect of the p_{atomic} parameter, which influences the rate at which policy graphs develop. The default setting of $p_{atomic} = 0.5$ has been used in all experiments so far, meaning that when a program's action pointer is modified, it has an equal probability of referencing either an atomic action or another team (created in any previous generation). A setting of $p_{atomic} = 1.0$ implies that action

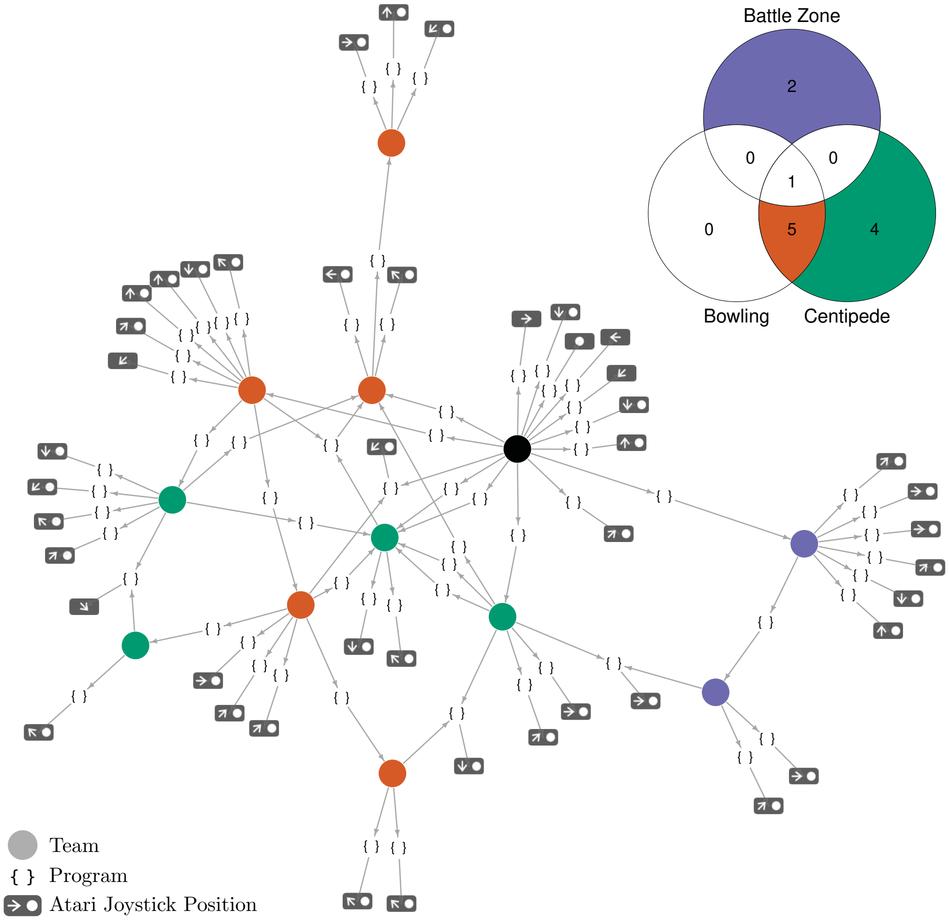


Figure 9.13: Champion multi-task TPG policy graph from the group 3.2 experiment. Decision-making in a policy graph begins at the root node (black circle) and follows *one* path through the graph until an atomic action (joystick position) is reached (See Algorithm 9). Venn diagram indicates which teams are visited while playing each game, over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision-making during test are shown. Figure requires viewing in colour.

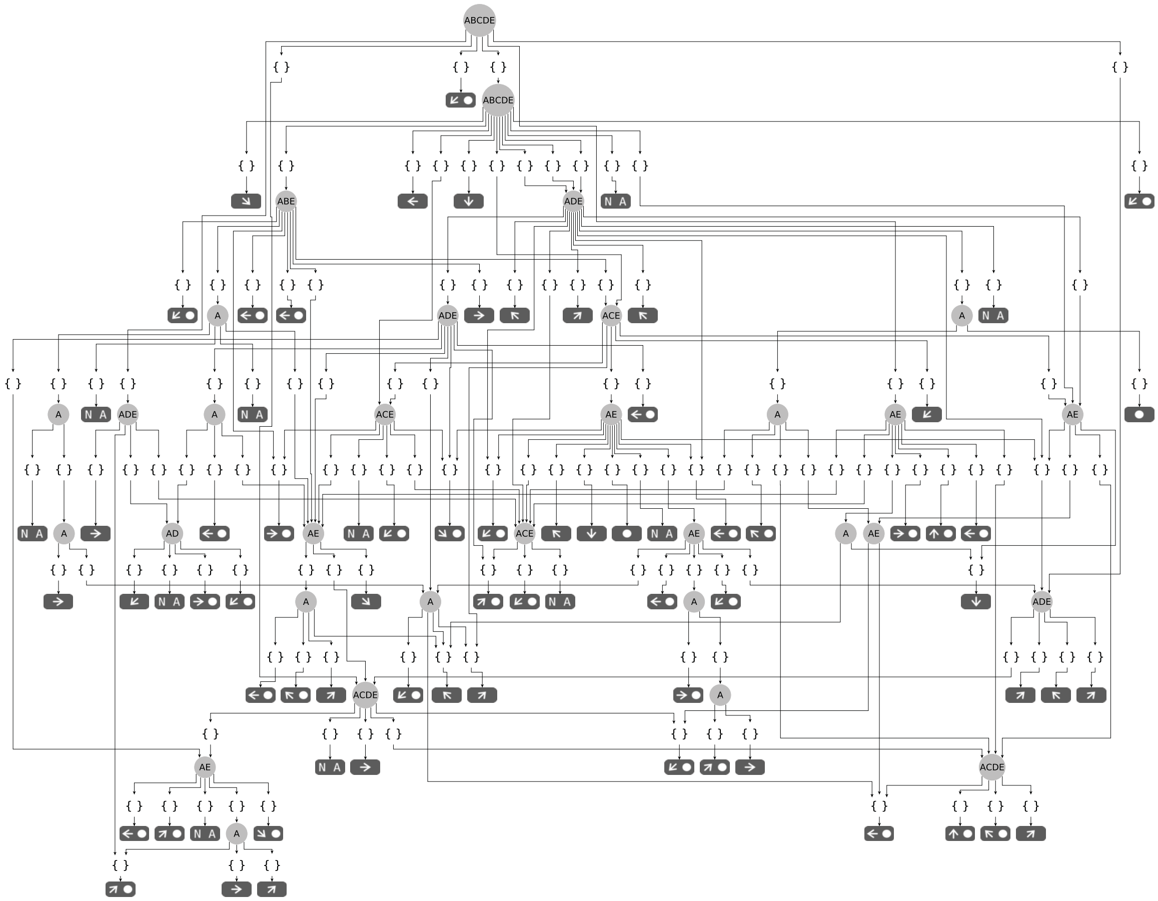


Figure 9.14: Champion multi-task TPG policy graph from the group 5.3 experiment. Decision-making in a policy graph begins at the root node (top of graph) and follows *one* path through the graph until an atomic action (joystick position) is reached (See Algorithm 9). Letters inside team nodes indicates which teams are visited while playing each game (alphabetical order), over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision-making during test are shown.

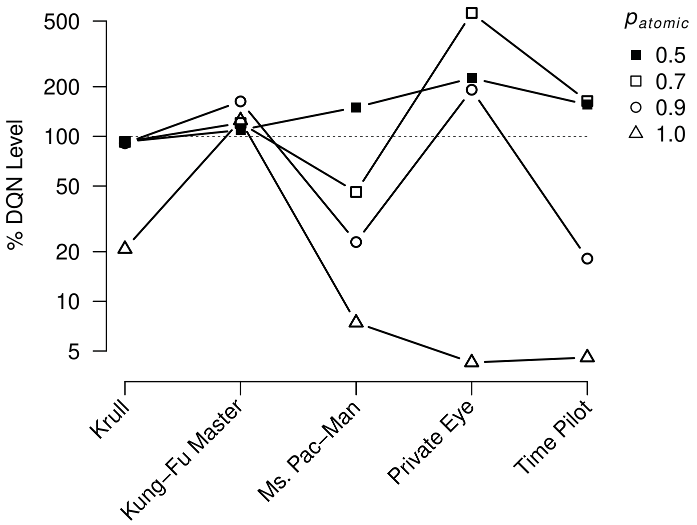


Figure 9.15: Test results for champion policy graphs from 4 independent MTRL experiments in the task group 5.3, each with a unique setting for the p_{atomic} parameter. Each experiment included 5 independent runs with a budget of 750 generations each. Post training, the elite multi-task policies from each run was tested in 30 episodes under each task as per established test procedure (Section 9.2). Test performance for the single best policy from each experiment is shown in the plot. The line connecting points emphasizes that scores are from the same (multi-task) policy.

pointers will always refer to atomic actions, and thus multi-team graphs will never emerge. Figure 9.15 reports the test performance of the single best policy from four independent MTRL experiments for the task group 5.3, each with a different p_{atomic} setting. Note that even runs in which policies were restricted to a single team of programs ($p_{atomic} = 1.0$) managed to produce one policy capable of playing the game Kung-Fu Master[11]. However, without the ability to build multi-team policy graphs, as is the case when $p_{atomic} = 1.0$, no multi-task policies emerged from any of the runs. By contrast, all runs with the capability to build policy graphs where able to discovered multi-task policies capable of playing at least 3 game titles.

[11] This single-team policy for Kung-Fu Master was simple but surprisingly effective. Kung-Fu Master is a side-scrolling fighting game. In addition to moving Left and Right, the player's actions include kick, punch, crouch, and jump. This policy simply crouched down continuously (avoiding the objects being thrown at it) and punched all opponent players as they approached. Eventually, enough objects are thrown toward the player at a low enough angle that crouching doesn't help, and since the policy does not jump to avoid them, the player eventually dies.

Complexity of Multi-Task Policy Graphs

Table 9.6 reports the average number of teams, instructions, and proportion of state space contributing to each decision for the multi-task champion during test. Interestingly, even for an evolved multi-task policy graph (i.e post-training), the number of instructions executed depends on the game in play, for example, ranging from 200 in Kangaroo to 512 in Kung-Fu Master for the Group 3.4 champion. While the complexity/cost of decision-making varies depending on the game in play, the average number of instructions per decision for the group 5.3 champion is 610, not significantly different from the average of 602 required by task-specific policies when playing the same games (See Table 9.3). Furthermore, the group 5.3 champion multi-task policy averaged 1832 - 2342 decisions per second during test, which is significantly faster than single-task policies from both DQN and Blob-PROST (See Table 9.2). Finally, as the parametrization for TPG under MTRL is identical to task-specific experiments with a significantly smaller population size (90 vs. 360), and the number of generations is similar in both cases[12], we can conclude that the cost of development is not significantly greater under MTRL.

9.7 Summary

Applying RL directly to high-dimensional decision-making tasks has previously been demonstrated using both neuro-evolution and multiple deep learning architectures. To do so, neuro-evolution assumed an a priori parameterization for model complexity whereas deep learning had the entire architecture pre-specified. Moreover, both previous approaches assume that all the state space should be indexed in order to compose solutions. In this work, an entirely emergent approach to evolution, or Tangled Program Graphs, is proposed in which the overall decision-making policy, including its topology and state space indexing, are evolved in an open-ended manner.

This chapter has demonstrated that TPG is able to evolve solutions to a suite of 49 Atari game titles that match the quality of those discovered by deep learning at a fraction of the model complexity. To do so, TPG begins with single teams of programs and incrementally discovers a graph of interconnectivity, potentially linking

[12] MTRL runs lasted 200 - 750 generations, which is roughly the range of generations reached for the single-task runs (See Figure9.2(a))

Table 9.6: Complexity of champion multi-task policy graphs from each game group in which all tasks were covered by a single policy. The cost of making each decision is relative to the average number of teams visited per decision (Tm), average number of instructions executed per decision (Ins), and proportion of state space indexed per decision (%SP). TPG wall-clock time is measured on a 2.2GHz Intel Xeon E5-2650 CPU.

Group	Title	Tm	Ins	%SP	Decisions per sec
3.2	Battle Zone	3	413	11	2687
	Bowling	4	499	15	2922
	Centipede	2	595	15	2592
3.4	Kangaroo	2	200	6	3141
	Krull	2	394	11	2502
	Kung-Fu Master	2	512	12	2551
3.5	Ms. Pac-Man	3	532	14	2070
	Private Eye	4	804	18	1857
	Time Pilot	5	869	19	1982
5.3	Krull	5	782	18	1832
	Kung-Fu Master	2	455	13	2342
	Ms. Pac-Man	5	673	16	1989
	Private Eye	3	481	13	2192
	Time Pilot	4	657	16	2306

hundreds of teams by the time competitive solutions are found. However, as each team can only have one action (per state), very few of the teams composing a TPG solution are evaluated in order to make each decision. This provides the basis for efficient real-time operation without recourse to specialized computing hardware. A simple methodology for multi-task learning with the TPG representation has also been demonstrated. Champion multi-task TPG agents can potentially play multiple games titles from direct screen capture, all at the level of deep learning, without incurring any additional training cost or solution complexity.

Chapter 10

Conclusions and Future Work

10.1 Summary of Goals, Methods, and Observations

Broadly speaking, the goal of this research is to investigate methods of applying machine learning, and GP in particular, to the construction of behavioural agents that scale to complex sequential decision-making tasks without extensive prior (task-specific) knowledge, and while also maintaining minimal model complexity. To this end, the utility of modularity and open-endedness / emergence has been considered through two related but independent approaches to hierarchical team-based GP: Policy Trees and Tangled Program Graphs. This section will summarize the goals and methodology behind both algorithms, and discuss how the experimental observations from each approach relate to one another.

In the case of Policy Trees, a methodology for transfer learning was proposed and evaluated in two challenging task environments. A suitable task decomposition is known a priori for both task environments. This study represents the first application of transfer learning within the SBB architecture. The general approach exploited the modular nature of SBB Policy Trees, and their incremental development over two phases of evolution. In testing this proposed methodology (detailed in Chapters 4, 5, and 6), the following observations are made:

- In the specific case of RoboCup, in which the goal was to transfer behaviours from the Keepaway and Scoring tasks to the more challenging task of Half-Field Offense, the transfer methodology outlined in Section 5.6 was shown to be a critical factor in scaling Policy Trees to match state of the art levels of play in HFO. Conversely, Policy Trees without transfer were not able to reach equivalent performance.

- Also in the case of RoboCup, explicit diversity maintenance (Section 5.4.1) during the development of source task policies played a critical role in successful transfer. In particular, two task-agnostic metrics for quantifying the difference between policies, one specific to team GP and one purely behavioural (i.e. applicable to any type of behavioural agent), were proposed. Their combination within a diversity switching scheme implies that both types of novelty might be maintained within an evolving population without requiring a parameter to explicitly balance any potential trade-offs. Indeed, while neither type of novelty was especially effective alone, their combination made the difference between success or failure for Policy Trees under the HFO task.

- Under the Ms. Pac-Man task, the same diversity maintenance mechanism (Section 5.4.1) was applied during the evolution of source task policies. In this case, the diversity mechanism effectively removed the need for human-defined subtasks. That is, assuming the proposed diversity mechanism during the development of independent, single-team policies during phase 1 of evolution resulted in a useful degree of specialization among 'source' policies. The policy trees developed during phase 2 of evolution were then able to identify a useful division of labour over source policies, ultimately discovering generalist policies that exceeded the current state of the art from neuro-evolution.

- Champion policy trees in both HFO and Ms. Pac-Man were shown to be significantly more efficient to deploy post-training than respective comparator algorithms in each domain. As a population-based (evolutionary) model, the sample complexity of policy tree development (i.e. the total number of observation-action-reward cycles experienced during training, Figure 1.1) is significantly greater than the Sarsa temporal difference method that represented the state of the art in HFO. However, champion policy trees for HFO were shown to be significantly more efficient to deploy *post-training*. This is a direct result of their modular structure, and the specific property that only a portion of the full decision-making structure of the policy is required for each individual decision. Likewise, final policy trees in Ms. Pac-Man were shown to be significantly simpler than solutions from neuro-evolution.

In summary, modularity (and by extensions hierarchy) in SBB Policy Trees is the single most important property that facilitates transfer learning through the recombination of diverse building blocks. Furthermore, modularity was shown to be the basis for efficient decision-making in policy trees. However, the nature of modularity in policy trees is constrained by two limitations outlined in Section 2.3.4:

1. Emergent events, in this case hierarchical transitions, are anticipated and parameterized by the human designer. This implies that the algorithm is not free to discover the appropriate degree of hierarchical complexity appropriate for a given task.

2. Hierarchical development in Policy Trees is strictly bottom-up. That is, only root-level policies are subject to variation in any phase of evolution, while lower-level structures represent a 'frozen' library of reusable code. Furthermore, the decision-making path from root team to atomic action must operate through every level of the hierarchy. Thus, without the ability to modify the library, the Policy Tree algorithm would not easily support the sequential learning of multiple tasks.

These limitations motivate the second approach explored in this thesis, or Tangled Program Graphs. In this case the basic SBB approach to teaming (Chapter 4) is extended through simple modifications to program variation operators (Section 8.2) and team-level decision-making (Algorithm 9), such that the degree of hierarchical modularity becomes an entirely open-ended, emergent property. TPG is benchmarked under the Arcade Learning Environment, in which agents observe their environment through a high-dimensional, visual sensory interface. The major observations from this study include:

- TPG produced high-quality behavioural agents in the Arcade Learning Environment over a diverse set of 49 game titles, all from direct screen capture state and without any task-specific tuning of learning parameters. From the perspective of game-playing ability, the quality of TPG policies were broadly equivalent to a set of deep learning and neuro-evolution approaches from the ALE literature (Section 9.5.2).

- Emergent events (hierarchical transitions) in TPG can happen at any time, and while development is bottom-up (i.e. solutions start simple and incrementally complexify), hierarchical development is not monotonic. That is, hierarchical structures grow and break down throughout evolution. Ultimately, this allows the degree of modularity in TPG to be adaptively scaled based on interaction with the task environment. Section 9.5.3 provides empirical evidence of this property.

- An implication of the emergent modularity described in the previous point is that TPG policies scale to high-dimensional visual inputs by learning to ignore regions of the state space that are not relevant to decision-making. Section 9.5.4 provides an analysis of this property.
- Emergent modularity in the TPG representation supports multi-task learning in the ALE. Specifically, an MTRL methodology is proposed in Section 9.6 through which TPG develops multi-task policies capable of playing up to 5 game titles at the level of a recent deep learning approach (DQN).
- Finally, and most importantly, TPG policies are shown to be significantly less computationally demanding than solutions from deep learning. This is true from the perspective of development (i.e. training, Section 9.5.5) and deployment post-training (Section 9.5.6).

In summary, this thesis has made contributions to evolutionary reinforcement learning by providing a working example for the theoretical / conceptual models of modularity and open-ended evolution reviewed in Chapter 2. From the broader perspective of reinforcement learning in general, this work represents a different perspective from the neural network models that are currently receiving widespread attention. In adopting a GP approach, the focus has been on emergent model building and, more specifically, the ability to adapt the behaviour *and* complexity of solutions through interaction with the task environment. In doing so, this work has proposed methodologies for important research topics in RL, namely domain independent AI, transfer learning, and multi-task learning.

10.2 Future Work

The following list of open questions represents possible directions for future research, primarily with respect to the TPG representation:

- The empirical evaluation of TPG in a visual reinforcement learning domain (Chapter 9) employed a screen quantization procedure that reduced the dimensionality of state observations prior to being presented to the agent. While a similar preprocessing step is employed by most RL approaches in the ALE, recent work has suggested that TPG may be capable of scaling to significantly higher-dimension RL tasks [123]. Determining the limits of this scalability in visual RL domains is one avenue for future work.

- In the case of high-dimensional visual input, there may be utility in enforcing different spatial constraints on the state variables indexed by different programs at initialization. At present any program may index any state variable. However, programs/teams might instead be initialized under a spatial constraint such that they only index state variables within a certain locality. Would such biases be useful for specific image processing tasks?

- The agents developed in this work have been purely reactive. That is, they represent a sensory-motor mapping from state to action. While the resulting policies were robust to some amount of partial observability (e.g. Section 7.1 describes partial observability in the ALE), addressing tasks with more significant, longer time-scale partial observability will likely require some form of temporal memory mechanism. Neural network research has made extensive use of 'Long Short Term Memory (LSTM) to address this issue (e.g. [8]) and proposed Neural Turing Machines to equip neural networks with external memory [46]. TPG faces unique opportunities and challenges with respect to supporting memory. One possibility is to make program registers stateful by not clearing register content prior to each execution. This will result in local memory, specific to each program. However, it does not address how to share memory between different teams (nodes) within the same graph. Given that TPG structures are emergent, what memory architectures would scale most gracefully with the emergent properties? In short, TPG carries spatial decomposition to new levels, what is the analogue for the decomposition of (temporal) memory?

- It would be interesting to know what happens to TPG under tasks described purely in terms of novelty as an objective. Would evolution converge to one ultimately novel individual graph that subsumes all other attempts at novelty, or would the result be a population of unique policy graphs?

- With respect to the suite of ALE tasks considered in Chapter 9, the complexity of champion TPG policies tended to plateau at roughly the same point as their fitness (See Figures 9.2 and 9.3). However, the team population size as a whole often continued to grow beyond this point. This implies that the search space for constructing policy graphs would also increase, and may eventually present a barrier to making further progress on the task. Future work could investigate this property.

- TPG's capacity for the sequential learning of multiple tasks was demonstrated in Section 9.6. Might the representation also help with non-stationary tasks? Teaming (or ensembles) has been widely recognized as particularly appropriate for streaming applications with non-stationary processes [53]. Does the extension to graphs provide further utility under streaming applications?

Bibliography

[1] Alexandros Agapitos, Julian Togelius, and Simon Mark Lucas. Evolving controllers for simulated car racing using object oriented genetic programming. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1543–1550. ACM, 2007.

[2] Manu Ahluwalia and Lawrence Bull. Co-evolving functions in genetic programming: Dynamic adf creation using glib. In *Proceedings of the 7th International Conference on Evolutionary Programming VII*, pages 809–818. Springer-Verlag, 1998.

[3] Atif M. Alhejali and Simon M. Lucas. Using a training camp with Genetic Programming to evolve Ms Pac-Man agents. In *IEEE Symposium on Computational Intelligence and Games*, pages 118–125, 2011.

[4] Atif M. Alhejali and Simon M. Lucas. Using genetic programming to evolve heuristics for a Monte Carlo Tree Search Ms Pac-Man agent. In *IEEE Symposium on Computational Intelligence and Games*, pages 1–8, 2013.

[5] David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*, pages 119–125. American Association for Artificial Intelligence, 2002.

[6] Peter Angeline. Advances in genetic programming. chapter Genetic Programming and Emergent Intelligence, pages 75–97. MIT Press, 1994.

[7] Peter Angeline and Jordan Pollack. Evolutionary module acquisition. In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163. MIT Press, 1993.

[8] Bram Bakker. Reinforcement learning with long short-term memory. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 1475–1482. MIT Press, 2001.

[9] Wolfgang Banzhaf. Genetic programming and emergence. *Genetic Programming and Evolvable Machines*, 15(1):63–73, 2014.

[10] Wolfgang Banzhaf, Bert Baumgaertner, Guillaume Beslon, René Doursat, James A. Foster, Barry McMullin, Vinicius Veloso de Melo, Thomas Miconi, Lee Spector, Susan Stepney, and Roger White. Defining and simulating open-ended novelty: requirements, guidelines, and challenges. *Theory in Biosciences*, 135(3):131–161, 2016.

[11] André da Motta Salles Barreto, Douglas Adriano Augusto, and Helio JC Barbosa. On the Characteristics of Sequential Decision Problems and Their Impact on Evolutionary Computation and Reinforcement Learning. In *Artificial Evolution*, pages 194–205. Springer, 2009.

[12] Samuel Barrett and Peter Stone. Cooperating with unknown teammates in robot soccer. In *AAMAS Autonomous Robots and Multirobot Systems Workshop (ARMS 2014)*, 2014.

[13] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[14] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 4148–4152. AAAI Press, 2015.

[15] Marc G. Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 864–871. AAAI Press, 2012.

[16] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[17] Josh Bongard. Behavior chaining - incremental behavior integration for evolutionary robotics. In *Artificial Life XI: International Conference on the Synthesis and Simulation of Living Systems*, pages 64–71, 2008.

[18] Matthew M. Botvinick, Yael Niv, and Andrew C. Barto. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, 113(3):262–280, 2009.

[19] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1986.

[20] Markus Brameier and Wolgang Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[21] Matthias F. Brandstetter and Samad Ahmadi. Reactive control of Ms. Pac Man using information retrieval based on genetic programming. In *2012 IEEE Conference on Computational Intelligence and Games*, pages 250–256, 2012.

[22] Alexander Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Reuse of neural modules for general video game playing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 353–359. AAAI Press, 2016.

[23] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1):139 – 159, 1991.

[24] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[25] John Cartlidge and Seth Bullock. Combating cevolutionary disengagement by reducing parasite virulence. *Evolutionary Computation*, 12(2):193–222, 2004.

[26] Kumar Chellapilla. Evolving computer programs without subtree crossover. *Transactions on Evolutionary Computation*, 1(3):209–216, 1997.

[27] William J. Clancey. *Situated Cognition: On Human Knowledge and Computer Representations*. Cambridge University Press, 1997.

[28] Giuseppe Cuccu and Faustino Gomez. When novelty is not enough. In *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation - Volume Part I*, pages 234–243. Springer-Verlag, 2011.

[29] Edwin D. de Jong, Dirk Thierens, and Richard A. Watson. Hierarchical genetic algorithms. In *Parallel Problem Solving from Nature - PPSN VIII*, pages 232–241. Springer Berlin Heidelberg, 2004.

[30] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1):1–30, 2006.

[31] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13(1):227–303, 2000.

[32] Bruce Digney. Emergent hierarchical control structures: Learning reactive / hierarchical relationships in reinforcement environments. In *Proceedings of the Fourth Conference on the Simulation of Adaptive Behavior*, January 1996.

[33] Bruce Digney. Learning hierarchical control structure for multiple tasks and changing environments. In *Proceedings of the Fifth Conference on the Simulation of Adaptive Behavior: SAB 98*, January 1998.

[34] Stephane Doncieux and Jean-Baptiste Mouret. Behavioral diversity with multiple behavioral distances. In *IEEE Congress on Evolutionary Computation*, pages 1427–1434, 2013.

[35] John A. Doucette, Peter Lichodzijewski, and Malcolm I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 97–104, 2012.

[36] Bruce Edmonds. Meta-genetic programming: co-evolving the operators of variation. Technical report, CPM Report 9832, Centre for Policy Modelling, Manchester Metropolitan University, 1998.

[37] Stefan Elfwing, Eiji Uchibe, Kenji Doya, and Henrik I. Christensen. Co-evolution of shaping rewards and meta-parameters in reinforcement learning. *Adaptive Behavior*, 16(6):400–412, 2008.

[38] Stepfan Elfwing, Eiji Uchibe, Kenji Doya, and Henrik I. Christensen. Evolutionary development of hierarchical learning structures. *IEEE Transactions on Evolutionary Computation*, 11(2):249–264, 2007.

[39] Carlos Espinosa-Soto and Andreas Wagner. Specialization can drive the evolution of modularity. *PLoS Computational Biology*, 6(3):e1000719, 2010.

[40] Anestis Fachantidis, Ioannis Partalas, Matthew E. Taylor, and Ioannis Vlahavas. *An Autonomous Transfer Learning Algorithm for TD-Learners*, pages 57–70. Springer International Publishing, 2014.

[41] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[42] James J. Gibson. *The ecological approach to visual perception.* Psychology Press, 2nd edition, 1986.

[43] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[44] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[45] Faustino Gomez. Sustaining diversity using behavioral information distance. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 113–120, 2009.

[46] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[47] Patricia M. Greenfield. Language, tools and brain: The ontogeny and phylogeny of hierarchically organized sequential behavior. *Behavioral and Brain Sciences*, 14(4):531551, 1991.

[48] Steven M. Guastafson and William H. Hsu. Layered learning in genetic programming for a cooperative robot soccer problem. In *European Conference on Genetic Prgramming*, volume 2038, pages 291–301, 2001.

[49] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[50] Matthew Hausknecht, Prannoy Mupparaju, Sandeep Subramanian, Shivaram Kalyanakrishnan, and Peter Stone. Half field offense: An environment for multiagent learning and ad hoc teamwork. In *AAMAS Adaptive Learning Agents (ALA) Workshop*, May 2016.

[51] Matthew Hausknecht and Peter Stone. The impact of determinism on learning atari 2600 games. In *Workshop at the AAAI Conference on Artificial Intelligence*, 2015.

[52] M. I. Heywood and P. Lichodzijewski. Symbiogenesis as a mechanism for building complex adaptive systems: A review. In *EvoApplications – Part I*, volume 6024 of *LNCS*, pages 51–60, 2010.

[53] Malcolm I. Heywood. Evolutionary model building under streaming data for classification tasks: opportunities and challenges. *Genetic Programming and Evolvable Machines*, 16(3):283–326, 2015.

[54] Philip Hingston. A Turing Test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):169–177, 2009.

[55] John H. Holland. Properties of the bucket brigade. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 1–7, 1985.

[56] William H. Hsu, Scott J. Harmon, Edwin Rodriguez, and Christopher Zhong. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer. In *Genetic and Evolutionary Computation Conference – Late Breaking Papers*, 2004.

[57] K. Imamura, T. Soule, R. B. Heckendorn, and J. A. Foster. Behavioural diversity and probabilistically optimal GP ensemble. *Genetic Programming and Evolvable Machines*, 4(3):235–254, 2003.

[58] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms*. Cambridge University Press, 2011.

[59] Edwin D. de Jong. A monotonic archive for pareto-coevolution. *Evolutionary Computation*, 15(1):61–93, 2007.

[60] Nicholas K. Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 752–757, August 2005.

[61] Tobias Jung and Daniel Polani. Learning RoboCup-Keepaway with kernels. *Journal of Machine Learning Research - Proceedings Track*, 1:33–57, 2007.

[62] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *arXiv preprint arXiv:1708.07902*, 2017.

[63] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[64] S. Kalyanakrishnan and P. Stone. An empirical analysis of value function-based and policy search reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 749–756, 2009.

[65] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in robocup soccer: A multiagent reinforcement learning case study. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Artificial Intelligence*, pages 72–85. Springer, 2007.

[66] Wolfgang Kantschik, Peter Dittrich, Markus Brameier, and Wolfgang Banzhaf. Meta-evolution in graph gp. In *Proceedings of the Second European Workshop on Genetic Programming*, pages 15–28. Springer-Verlag, 1999.

[67] Nadav Kashtan, Elad Noor, and Uri Alon. Varying environments can speed up evolution. *Proceedings of the National Academy of Sciences*, 104(34):13711–13716, 2007.

[68] Stephen Kelly. On developmental variation in hierarchical symbiotic policy search. Master's thesis, Faculty of Computer Science, Dalhousie University, 2012.

[69] Stephen Kelly and Malcolm I. Heywood. Genotypic versus behavioural diversity for teams of programs under the 4-v-3 keepaway soccer task. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3110–3111, 2014.

[70] Stephen Kelly and Malcolm I. Heywood. On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In *European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 75–86. Springer, 2014.

[71] Stephen Kelly and Malcolm I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis: Building simple programs for a complex task. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1143–1150, 2015.

[72] Stephen Kelly and Malcolm I. Heywood. Emergent tangled graph representations for atari game playing agents. In *European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 64–79, 2017.

[73] Stephen Kelly and Malcolm I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, 2017.

[74] Stephen Kelly and Malcolm I. Heywood. Discovering agent behaviors through code reuse: Examples from half-field offense and ms. pac-man. *IEEE Transactions on Games*, 10(2):195–208, 2018.

[75] Stephen Kelly, Peter Lichodzijewski, and Malcolm I. Heywood. On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pages 3245–3252, 2012.

[76] John F. C. Kingman. A simple model for the balance between selection and mutation. *Journal of Applied Probability*, 15(1):112, 1978.

[77] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *arXiv preprint 1612.00796*, 2016.

[78] Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. In M. Wiering and M. van Otterio, editors, *Reinforcement Learning*, pages 579–610. Springer, 2012.

[79] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[80] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[81] John R. Koza, David Andre, Forrest H. Bennett, and Martin A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., 1st edition, 1999.

[82] Krzysztof Krawiec and Bir Bhanu. Visual Learning by Evolutionary and Coevolutionary Feature Synthesis. *IEEE Transactions on Evolutionary Computation*, 11(5):635–650, 2007.

[83] Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *arXiv preprint arXiv:1604.06057*, 2016.

[84] Mateusz Kurek and Wojciech Jakowski. Heterogeneous team deep q-learning in low-dimensional multi-agent environments. In *IEEE Conference on Computational Intelligence and Games*, pages 201–208. IEEE, 2016.

[85] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

[86] John Levine, Clair B. Congdon, Marc Ebner, Graham Kendall, Simon M. Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General Video Game Playing. In *Artificial and Computational Intelligence in Games*, volume 6, pages 77–83. 2013.

[87] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael Bowling. State of the art control of Atari games using shallow reinforcement learning. In *Proceedings of the ACM International Conference on Autonomous Agents and Multiagent Systems*, pages 485–493, 2016.

[88] Peter Lichodzijewski. *A symbiotic bid-based framework for problem decomposition using Genetic Programming.* PhD thesis, Faculty of Computer Science, Dalhousie University, 2011.

[89] Peter Lichodzijewski and Malcolm I. Heywood. Coevolutionary bid-based genetic programming for problem decomposition in classification. *Genetic Programming and Evolvable Machines*, 9:331–365, 2008.

[90] Peter Lichodzijewski and Malcolm I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 863–870, 2008.

[91] Peter Lichodzijewski and Malcolm I. Heywood. Symbiosis, complexification and simplicity under GP. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 853–860, 2010.

[92] Peter Lichodzijewski and Malcolm I. Heywood. The Rubik Cube and GP temporal sequence learning: an initial study. In *Genetic Programming Theory and Practice VIII*, chapter 3, pages 35–54. Springer, 2011.

[93] Mee Hong Ling, Kok-Lim Alvin Yau, Junaid Qadir, Geong Sen Poh, and Qiang Ni. Application of reinforcement learning for security enhancement in cognitive radio networks. *Applied Soft Computing*, 37(C):809–829, 2015.

[94] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *arXiv preprint arXiv:1709.06009*, 2017.

[95] John Maynard-Smith and Eors Szathmary. *The Major Transitions in Evolution.* New York: Oxford University Press, 1997.

[96] Elizabeth A. McGovern. *Autonomous Discovery of Temporal Abstractions from Interaction with An Environment.* PhD thesis, University of Massachusetts, 2002.

[97] Jan Hendrik Metzen, Mark Edgington, Yohannes Kassahun, and Frank Kirchner. Performance evaluation of EANT in the robocup keepaway benchmark. In *IEEE International Conference on Machine Learning and Applications*, pages 342–347, 2007.

[98] Jan Hendrik Metzen, Mark Edgington, Yohannes Kassahun, and Frank Kirchner. Analysis of an evolutionary reinforcement learning method in a multiagent domain. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pages 291–298, 2008.

[99] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 1928–1937, 2016.

[100] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel. Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[101] David E. Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5:4, 373–399.

[102] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999.

[103] Jean-Baptiste Mouret and Staphane Doncieux. Encouraging behavioral diversity in evolutionary robotics: an empirical study. *Evolutionary computation*, 20(1):91–133, 2012.

[104] Yavar Naddaf. *Game-independent AI agents for playing Atari 2600 console games*. Masters thesis, University of Alberta, 2010.

[105] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[106] Aurora M. Nedelcu, Richard E. Michod, and I. Overview. Evolvability, modularity, and individuality during the transition to multicellularity in volvocalean green algae. In *In Modularity in development and evolution*. Chicago Press, 2002.

[107] Stefano Nolfi. Using emergent modularity to develop control systems for mobile robots. *Adaptive behavior*, 5(3-4):343–363, 1997.

[108] Scott E. Page. *Diversity and Complexity*. Princeton University Press, 1st edition, 2010.

[109] Emilio Parisotto, Lei Jimmy Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[110] M. Parter, N. Kashtan, and U. Alon. Facilitated variation: How evolution learns from past environments to generalize to new environments. *PLoS Computational Biology*, 4(11):e1000206, 2008.

[111] T. Pepels and M. H. M. Winands. Enhancements for monte-carlo tree search in ms pac-man. In *IEEE Symposium on Computational Intelligence in Games*, pages 265–272, 2012.

[112] Diego Perez-Liebana, Samothrakis Samothrakis, Julian Togelius, Tom Schaul, and Simon M. Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[113] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[114] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[115] Justinian Rosca. Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85. AAAI, 1995.

[116] Jrgen Schmidhuber. Adaptive confidence and adaptive curiosity. Technical report, Institut fur Informatik, Technische Universitat Munchen, 1991.

[117] Jacob Schrum and Risto Miikkulainen. Discovering multimodal behavior in Ms. Pac-Man through evolution of modular neural networks. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):67–81, 2016.

[118] Siang Yew Chong, P. 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.

[119] Herbert A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.

[120] Satinder Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Trans. on Auton. Ment. Dev.*, 2(2):70–82, 2010.

[121] Moshe Sipper. *Evolved to Win*. Lulu, 2011.

[122] Robert J. Smith and Malcolm I. Heywood. Coevolving deep hierarchies of programs to solve complex tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1009–1016. ACM, 2017.

[123] Robert J. Smith and Malcolm I. Heywood. Scaling tangled program graphs to visual reinforcement learning in vizdoom. In *European Conference on Genetic Programming*, 2018.

[124] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 215–223. MIT Press, 1996.

[125] Lee Spector, Brian Martin, Kyle Harrington, and Thomas Helmuth. Tag-based modules in genetic programming. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1419–1426. ACM, 2011.

[126] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[127] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[128] Susan Stepney and Tim Hoverd. Reflecting on open-ended evolution. In *ECAL '11, Proceedings of the 11th European Conference on Artificial Life*, pages 781–788. MIT Press, 2011.

[129] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223. Springer-Verlag, 2002.

[130] P. Stone. *Layered learning in multiagent systems*. MIT Press, 2000.

[131] Peter Stone. Learning and multiagent reasoning for autonomous agents. In *The 20th International Joint Conference on Artificial Intelligence*, pages 13–30, 2007.

[132] Peter Stone, Gregory Kuhlmann, Matthew Taylor, and Yaxin Liu. Keepaway soccer: From machine learning testbed to benchmark. In *RoboCup 2005: Robot Soccer World Cup IX*, pages 93–105. 2006.

[133] Peter Stone and Richard S. Sutton. Scaling reinforcement learning toward robocup soccer. In *The Eighteenth International Conference on Machine Learning*, pages 537–544, 2001.

[134] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

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

[136] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[137] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[138] Istvan. Szita. Reinforcement learning in games. In *Reinforcement Learning*, pages 539–577. Springer, 2012.

[139] Amir Tavafi and Wolfgang Banzhaf. A hybrid genetic programming decision making system for robocup soccer simulation. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1025–1032. ACM, 2017.

[140] Matthew E. Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 283–290, 2008.

[141] Matthew E. Taylor and Peter Stone. Cross-domain transfer for reinforcement learning. In *International Conference on Machine Learning*, pages 879–886, 2007.

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

[143] Matthew E. Taylor and Peter Stone. An introduction to inter-task transfer for reinforcement learning. *AI Magazine*, 32(1):15–34, 2011.

[144] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007.

[145] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1321–1328, 2006.

[146] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *Proceedings of the ACM International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1–37, 2007.

[147] Tim Taylor. *From Artificial Evolution to Artificial Life*. PhD thesis, The University of Edinburgh, 1999.

[148] Astro Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, 1996.

[149] Russell Thomason and Terence Soule. Novel ways of improving cooperation and performance in ensemble classifiers. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1708–1715, 2007.

[150] Lisa Torrey, Jude Shavlik, Trevor Walker, and Richard Maclin. Skill acquisition via transfer learning and advice taking. In *European Conference on Machine Learning*, pages 425–436. Springer, 2006.

[151] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2094–2100, 2016.

[152] Sjoerd van Steenkiste, Jan Koutník, Kurt Driessens, and Jürgen Schmidhuber. A wavelet-based encoding for neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 517–524, 2016.

[153] 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.

[154] Phillip Verbancsics and Kenneth O. Stanley. Evolving static representations for task transfer. *Journal of Machine Learning Research*, 11:1737–1769, 2010.

[155] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[156] Gunter P. Wagner and Lee Altenberg. Perspective: complex adaptations and the evolution of evolvability. *Evolution*, 50:967–976, 1996.

[157] M. Waibel, L. Keller, and D. Floreano. Genetic team composition and level of selection in the evolution of cooperation. *IEEE Transactions on Evolutionary Computation*, 13(3):648–660, 2009.

[158] Richard A. Watson and Jordan B. Pollack. Modular interdependency in complex dynamical systems. *Artificial Life*, 11(4):445–457, 2005.

[159] Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59(1):5–30, 2005.

[160] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Journal of Autonomous Agents and Multi-Agent Systems*, 21(1):1–27, 2009.

[161] Shelly Xiaonan Wu and Wolfgang Banzhaf. Rethinking multilevel selection in genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1403–1410, 2011.

[162] Andrew S. Yang. Modularity, evolvability, and adaptive radiations: a comparison of the hemi- and holometabolous insects. *Evolution and Development*, 3(2):59–72, 2001.

[163] G. N. Yannakakis and J. Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2015.

Appendices

Appendix A

RoboCup Soccer Server Parameters

Table A.1: Soccer server parameter settings used for all RoboCup experiments in this thesis. Detailed parameter descriptions are provided in the soccer server manual at `https://sourceforge.net/projects/sserver/files/rcssserver/`

Parameter	**Value**	**Parameter**	**Value**
half_time	-1.0	forbid_kick_off_offside	0.0
use_offside	0.0	stamina_inc_max	3500
stamina_capacity	-1.0	visible_angle	360
quantize_step	0.1	quantize_step_l	0.01
team_actuator_noise	false	wind_random	false
kick_rand	0.05	kick_rand_factor_l	1.0
kick_rand_factor_r	1.0	player_rand	0.1
prand_factor_l	1.0	prand_factor_r	1.0
tackle_rand_factor	2.0	wind_rand	0.0
back_passes	0.0	ball_rand	0.0

Appendix B

Robocup Sensors

Table B.1: Sensor inputs (state variables) for the 4v3 Half-field Keepaway task and 4v4 Half-field Offense task. P_p is the player in possession of the ball. The rest of the players on P_p's team are numbered relative to their distance from P_p with P_1 being the closest. 'Opponent' refers to any taker in the Keepaway task and any 'defense' player in the HFO task. All sensor inputs are real-valued. Angle values range from 0° to 360° and distance values range from 0 to approximately 85 meters.

			Sensor Input	**Description**
Keepaway	Half-field Offense	a	$dist(P_p, P_1)$	Distance from P_p to closest teammate.
		b	$dist(P_p, P_2)$	Distance from P_p to second-closest teammate.
		c	$dist(P_p, P_3)$	Distance from P_p to third-closest teammate
		d	$dist(P_1, O_{min})$	Distance from P_1 to closest opponent.
		e	$dist(P_2, O_{min})$	Distance from P_2 to closest opponent.
		f	$dist(P_3, O_{min})$	Distance from P_3 to closest opponent.
		g	$min_{angle}(P_1, O_p, O_{min})$	Min. angle, with vertex at P_p, between P_1 and any opponent.
		h	$min_{angle}(P_2, O_p, O_{min})$	Min. angle, with vertex at P_p, between P_2 and any opponent.
		i	$min_{angle}(P_3, O_p, O_{min})$	Min. angle, with vertex at P_p, between P_3 and any opponent.
		j	$min_{dist}(P_p, O_{Dcone})$	Min. distance from P_p to any player in the dribble cone. The dribble cone is a cone with half angle 60° with its vertex at P_p and axis passing through the centre of the goal. D_{cone} is the set of defenders in the dribble cone.
		k	$dist(P_p, O_{min})$	Distance from P_p to closest opponent.
		l	$dist(P_p, GL)$	Distance from P_p to goal line.
		m	$dist(P_1, GL)$	Distance from P_1 to goal line.
		n	$dist(P_2, GL)$	Distance from P_2 to goal line.
		o	$dist(P_3, GL)$	Distance from P_3 to goal line.
		p	$max_{angle}(P_p, Goal)$	Max. angle with vertex at P_p, formed by rays connecting P_p and goalposts or opponents in the goal cone, which is the triangle formed by P_p and the two goalposts.
		q	$dist(P_p, Goalie)$	Distance from P_p to the goalie.

Appendix C

Ms. Pac-Man Sensors

Table C.1: Undirected sensor inputs (state variables) for the Ms. Pac-Man task. Binary sensors are marked by shaded rows. All other sensors are real-valued and scaled to the range $[0, 1]$.

Sensor	**Description**
0	Proportion of power pills left in maze
1	Proportion of regular pills left in maze
2	Proportion of ghosts that are edible
3	Proportion ghost edible time remaining
4	1 if any ghost is edible, 0 otherwise
5	1 if four threats are outside the lair, 0 otherwise
6	1 if Ms. Pac-Man is within 10 steps of a power pill, 0 otherwise

Table C.2: Directed sensor inputs (state variables) for the Ms. Pac-Man task. Binary sensors are marked by shaded rows. All other sensors are real-valued and scaled to the range $[0, 1]$.

Sensor	**Description**
7	Distance to nearest regular pill (in given direction)
8	Distance to nearest power pill
9	Distance to nearest maze junction
10	Distances to the closest ghost
11	1 if closest ghost is approaching, 0 otherwise
12	1 if a directional path to closest ghost contains no junctions, 0 otherwise
13	1 if closest ghost is edible, 0 otherwise
14	Distances to the second closest ghost
15	1 if second closest ghost is approaching, 0 otherwise
16	1 if a directional path to second closest ghost contains no junctions, 0 otherwise
17	1 if second closest ghost is edible, 0 otherwise
18	Distances to the third closest ghost
19	1 if third closest ghost is approaching, 0 otherwise
20	1 if a directional path to third closest ghost contains no junctions, 0 otherwise
21	1 if third closest ghost is edible, 0 otherwise
22	Distances to the fourth closest ghost
23	1 if fourth closest ghost is approaching, 0 otherwise
24	1 if a directional path to fourth closest ghost contains no junctions, 0 otherwise
25	1 if fourth closest ghost is edible, 0 otherwise
26	Proportion of pills on the path in the given direction that has the most pills
27	Proportion of junctions on the path in the given direction that has the most junctions
28	Proportion of junctions reachable from next nearest junction that Ms. Pac-Man is closer to than a threat ghost

Appendix D

ALE Comparator tables

Test performance over all 49 game titles is split between two comparator groups (Section 9.5.2), those assuming screen capture state information (Table D.1) and those assuming hand crafted state (Table D.2). TPG uses screen capture state in both cases.

Table D.1: Average game score of best agent under test conditions for TPG along with comparator algorithms in which *screen capture* represent state information. Figures in bold represent best score on each game title. Source information for comparator algorithms is as follows: DQN ([100]), Gorila ([105]), Double DQN ([151]), Hyper-NEAT ([51]).

Game	**TPG**	**DQN**	**Gorila**	**Double DQN**	**Hyper-NEAT**
Alien	**3,382.7**	3,069.3	2,621.53	2,907.3	1,586
Amidar	398.4	739.5	**1,189.7**	702.1	184.4
Assault	2,422	3,359.6	1,450.4	**5,022.9**	912.6
Asterix	2,400	6,011.7	6,433.3	**15,150**	2,340
Asteroids	**3,050.7**	1,629.3	1,047.7	930.3	1,694
Atlantis	89,653	85,950	**100,069**	64,758	61,260
Bank Heist	**1,051**	429.7	609	728.3	214
Battle Zone	**47,233.4**	26,300	25,266.7	25,730	36,200
Beam Rider	1,320.8	6,845.9	3,302.9	**7,654**	1,412.8
Bowling	**223.7**	42.4	54	70.5	135.8
Boxing	76.5	71.8	**94.9**	81.7	16.4
Breakout	12.8	401.2	**402.2**	375	2.8
Centipede	**34,731.7**	8,309.4	8,432.3	4,139.4	25,275.2
C. Command	**7,070**	6,686.7	4,167.5	4,653	3,960
Crazy Climber	8,367	**114,103.3**	85,919.1	101,874	0
Demon Attack	2,920.4	9,711.2	**13,693.1**	9,711.9	3,590
Double Dunk	**2**	-18.1	-10.6	-6.3	**2**
Enduro	125.9	301.8	114.9	**319.5**	93.6
Fishing Derby	**49**	-0.8	20.2	20.3	-49.8
Freeway	28.9	30.3	11.7	**31.8**	29
Frostbite	**8,144.4**	328.3	605.2	241.5	2,260
Gopher	581.4	**8,520**	5,279	8,215.4	364
Gravitar	786.7	306.7	**1,054.6**	170.5	370
H.E.R.O	16,545.4	19,950.3	14,913.9	**20,357**	5,090
Ice Hockey	10	-1.6	-0.6	-2.4	**10.6**
James Bond	3,120	576.7	605	438	**5,660**
Kangaroo	**14,780**	6,740	2,547.2	13,651	800
Krull	**12,850.4**	3,804.7	7,882	4,396.7	12,601.4
Kung-Fu Master	**43,353.4**	23,270	27,543.3	29,486	7,720
M's Revenge	0	0	**4.3**	0	0
Ms. Pac-Man	**5,156**	2,311	3,233.5	3,210	3,408
Name This Game	3,712	**7,256.7**	6,182.2	6,997.1	6,742
Pong	6	18.9	18.3	**21**	-17.4
Private Eye	**15,028.3**	1,787.6	748.6	670.1	10,747.4
Q*Bert	2,245	10,595.8	10,815.6	**14,875**	695
River Raid	3,884.7	8,315.7	8,344.8	**12,015**	2,616
Road Runner	27,410	18,256.7	**51,008**	48,377	3,220
Robotank	22.9	**51.6**	36.4	46.7	43.8
Seaquest	1,368	5,286	**13,169.1**	7,995	716
Space Invader	1,597.2	1,975.5	1,883.4	**3,154**	1,251
Star Gunner	1,406.7	57,996.7	19,145	**65,188**	2,720
Tennis	0	-1.6	**10.9**	1.7	0
Time Pilot	**13,540**	5,946.7	10,659.3	7,964	7,340
Tutankham	128	186.7	**245**	190.6	23.6
Up and Down	34,416	8,456.3	12,561.6	16,769.9	**43,734**
Venture	576.7	380	**1,245**	0	1,188
Video Pinball	37,954.4	42,684.1	**157,550.2**	70,009	0
Wizard of Wor	5,196.7	3,393.3	**13,731.3**	5,204	3,360
Zaxxon	6,233.4	4,976.7	7,129.3	**10,182**	3,000
Avg. Rank (R_i)	2.74	3.11	2.63	2.64	3.87

Table D.2: Average game score of best agent under test conditions for TPG (screen capture) along with comparator algorithms based on *prior object/feature identification.* Figures in bold represent best score on each game title. Source information for comparitor algorithms is as follows: Blob-PROST ([87]), Hyper-NEAT ([51]), NEAT ([51]), Conti-Sarsa ([100])

Game	**TPG**	**Blob-PROST**	**Hyper-NEAT**	**NEAT**	**Conti-Sarsa**
Alien	3,382.7	**4,886.6**	2,246	4,320	103.2
Amidar	398.4	**825.6**	218.8	325.2	183.6
Assault	2,422	1,829.3	2,396	**2,717.2**	537
Asterix	2,400	**2,965.5**	2,550	1,490	1,332
Asteroids	3,050.7	2,229.9	220	**4,144**	89
Atlantis	89,653	42,937.7	44,200	**126,260**	852.9
Bank Heist	1,051	793.6	**1,308**	380	67.4
Battle Zone	**47,233.4**	37,850	37,600	45,000	16.2
Beam Rider	1,320.8	**2,965.5**	1,443.2	1,900	1,743
Bowling	223.7	91.1	**250.4**	231.6	36.4
Boxing	76.5	**98.3**	91.6	92.8	9.8
Breakout	12.8	**190.3**	40.8	43.6	6.1
Centipede	**34,731.7**	21,137	33,326.6	22,469.6	4,647
C. Command	7,070	4,898.9	**8,120**	4,580	16.9
Crazy Climber	8,367	**81,016**	12,840	25,060	149.8
Demon Attack	2,920.4	2,166	3,082	**3,464**	0
Double Dunk	2	-4.1	4	**10.8**	-16
Enduro	125.9	**299.1**	112.8	133.8	159.4
Fishing Derby	**49**	-28.8	-37	-43.8	-85.1
Freeway	28.9	**32.6**	29.6	30.8	19.7
Frostbite	**8,144.4**	4,534	2,226	1,452	180.9
Gopher	581.4	**7,451.1**	6,252	6,029	2,368
Gravitar	786.7	1,709.7	1,990	**2,840**	429
H.E.R.O	16,545.4	**20,273.1**	3,638	3,894	7,295
Ice Hockey	10	**22.8**	9	3.8	-3.2
James Bond	3,120	1,030.5	**12,730**	2,380	354.1
Kangaroo	**14,780**	9,492.8	4,880	12,800	8.8
Krull	12,850.4	**33,263.4**	23,890.2	20,337.8	3,341
Kung-Fu Master	43,353.4	51,007.6	47,820	**87,340**	29,151
M's Revenge	0	**2,508.4**	0	340	259
Ms. Pac-Man	5,156	**5,917.9**	3,830	4,902	1,227
Name This Game	3,712	7,787	**8,346**	7,084	2,247
Pong	6	**20.5**	4	15.2	-17.4
Private Eye	15,028.3	100.3	**15,045.2**	1,926.4	86
Q*Bert	2,245	**14,449.4**	810	1,935	960.3
River Raid	3,884.7	**14,583.3**	4,736	4,718	2,650
Road Runner	27,410	**41,828**	14,420	9,600	89.1
Robotank	22.9	34.4	**42.4**	18	12.4
Seaquest	1,368	2,278	**2,508**	944	675.5
Space Invader	**1,597.2**	889.8	1,481	1,481	267.9
Star Gunner	1,406.7	1,651.6	4,160	**9,580**	9.4
Tennis	0	0	0.2	**1.2**	0
Time Pilot	13,540	5,429.5	**15,640**	14,320	24.9
Tutankham	128	**217.7**	110	142.4	98.2
Up and Down	34,416	**41,257.8**	6,818	10,220	2,449
Venture	576.7	**1,397**	400	340	0.6
Video Pinball	3,794.4	21,313	82,646	**253,986**	19,761
Wizard of Wor	5,196.7	5,681.2	3,760	**17,700**	36.9
Zaxxon	6233.4	**11,721.8**	4,680	6,460	21.4
Avg. Rank (R_i)	2.81	2.16	2.81	2.47	4.76

Appendix E

Complexity of Deep Q-Network for Reinforcement Learning in the Arcade Learning Environment

Complexity of the Deep Q-Network use for comparison in Chapter 9 is calculated as follows (See Methods Section in [100]). Each Atari screen frame was down-sampled prior to being presented to the DQN agent. This screen preprocessing map converts each Atari screen frame from the original size of 210 × 160 pixels (with a 128-colour palette) to an 84 × 84 pixel map. The 4 most recent frames are stacked, producing a final preprocessed image size of $84 \times 84 \times 4$. The neural network has three convolution layers, followed by a Multi-Layer Perceptron with two fully connected layers and one output for each action.

The convolution layers of the network transform 3-dimensional input volumes (i.e. image data) to 3-dimensional output volumes. The depth, or the number of ”filters“, in each output volume is specified a priori for each convolutional layer. The size (width/height) of the output from each convolution layer can be computed as a function of the input size (W), the size of each neuron's receptive field (F), the stride with which neurons slide across the input (S), and the amount of zero padding (P) used around the image border. Given this information, the formula for calculating the output size (O) from each layer is:

$$O = \frac{W - F + 2P}{S} + 1 \tag{E.1}$$

The original DQN paper made no mention of zero padding, and it is therefore assumed here that none was used. Calculating the output size (width/height) from each convolution layer of the network proceeds as follows:

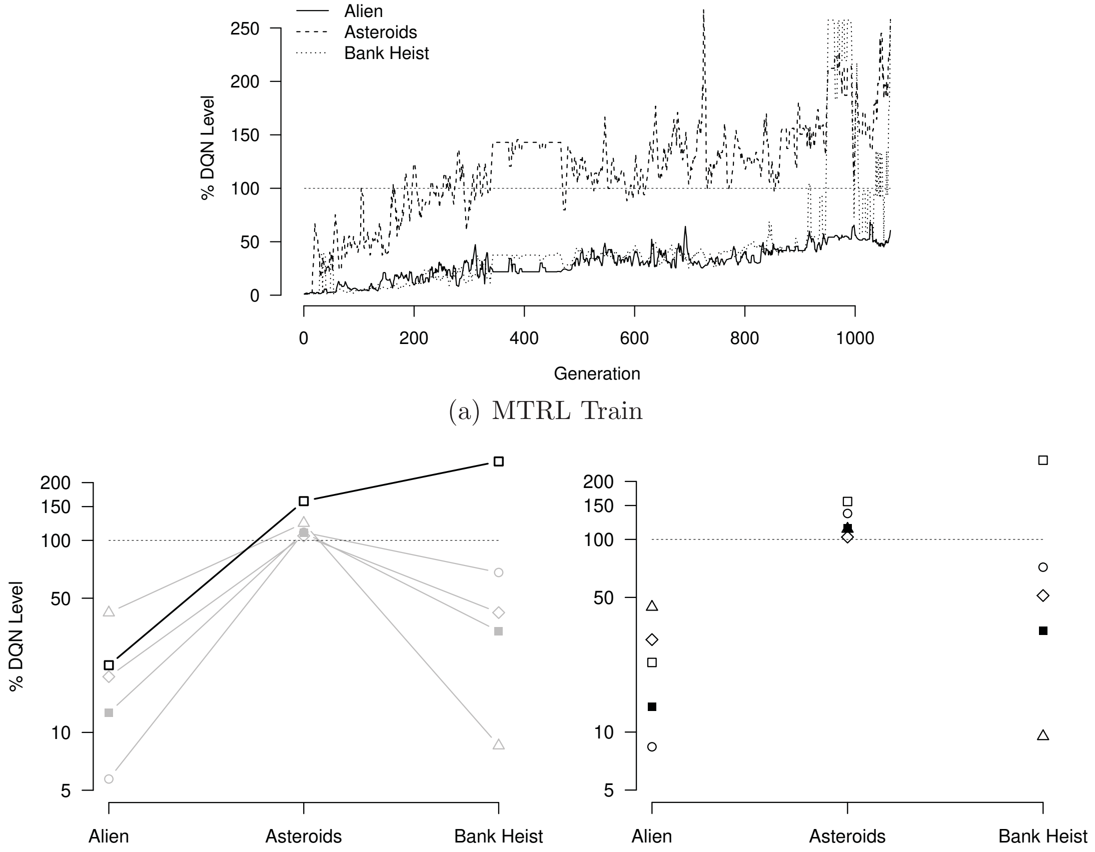
- The first convolution layer consists of 32 filters of 8×8 with stride 4. Applying Equation E.1 gives a layer 1 output size of $\frac{84-8}{4} + 1 = 20$, for a final 3D volume of $20 \times 20 \times 32$.

- The second convolution layer consists of 64 filters of 4×4 with stride 2. Applying Equation E.1 gives a layer 2 output size of $\frac{20-4}{2}+1=9$, for a final 3D volume of $9\times9\times64$.
- The third convolutional layer consists of 64 filters of 3×3 with stride 1. Applying Equation E.1 gives a layer 3 output size of $9-3+1=7$, for a final 3D volume of $7\times7\times64$.

Thus, the output dimensionality of the final convolution layer is $7\times7\times64=3136$. Fully connecting this volume to the first MLP hidden layer (512 neurons) requires $3136\times512=1,605,632$ connections (weight parameters). For the purposes of the comparison made in this thesis, these parameters will be used to represent a snapshot of the complexity of DQN when estimating its computational cost in the Arcade Learning Environment. Specifically, each neuron in the MLP layer computes the dot product over its inputs, and we can therefore assume one calculation per input for each forward pass through the network. The total cost of processing 50 million frames (the frame budget used in [100] for training on each game title) is now at least 8.03×10^{13} calculations. This characterization of DQN's complexity is simple, accurate given the information available from [100], and very conservative, since it does not account for the cost of the convolution layers or the computational costs associated with back propagation, i.e. this is the cost associated with performing the forward pass, at a single layer of the DQN architecture (associated with the input to the MLP), for the 50 million training frames. For a tutorial on estimating the size of filters in deep learning architectures see `http://cs231n.github.io/convolutional-networks/`.

Appendix F

Additional TPG Multi-Task Learning Results

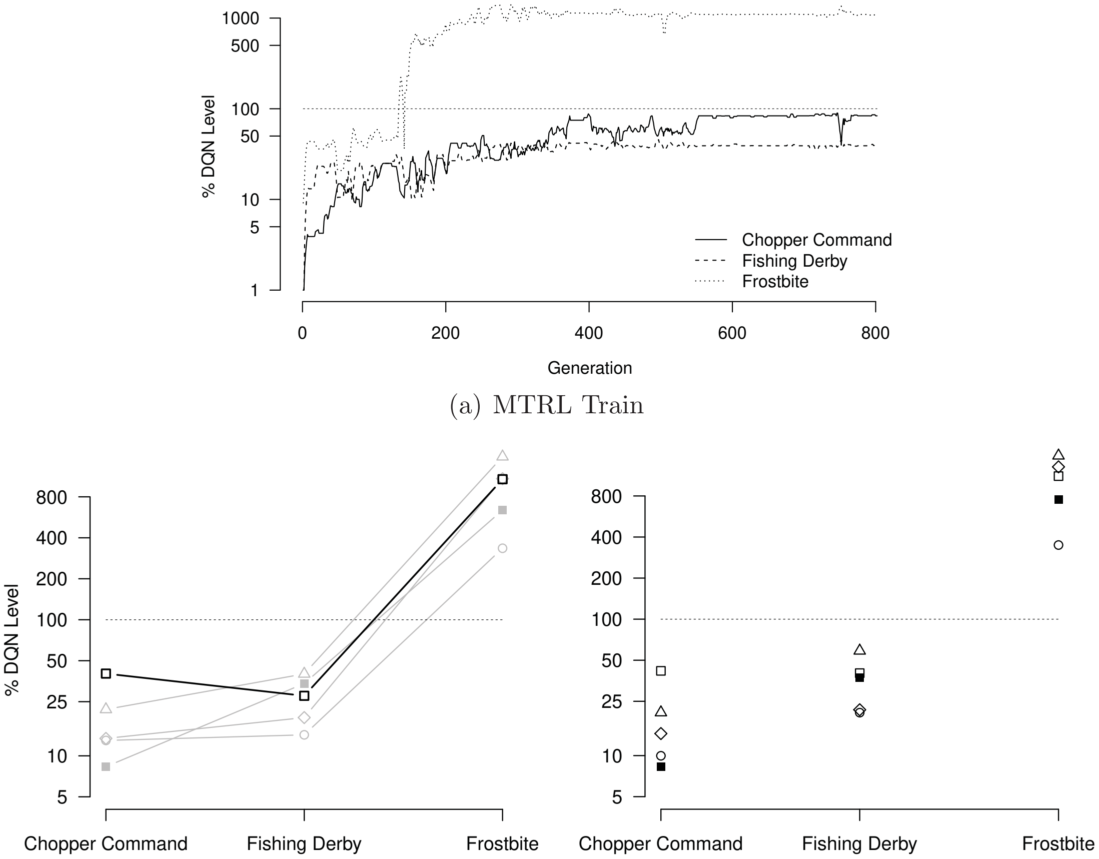


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.1: TPG multi-task reinforcement learning results for game group 3.1. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

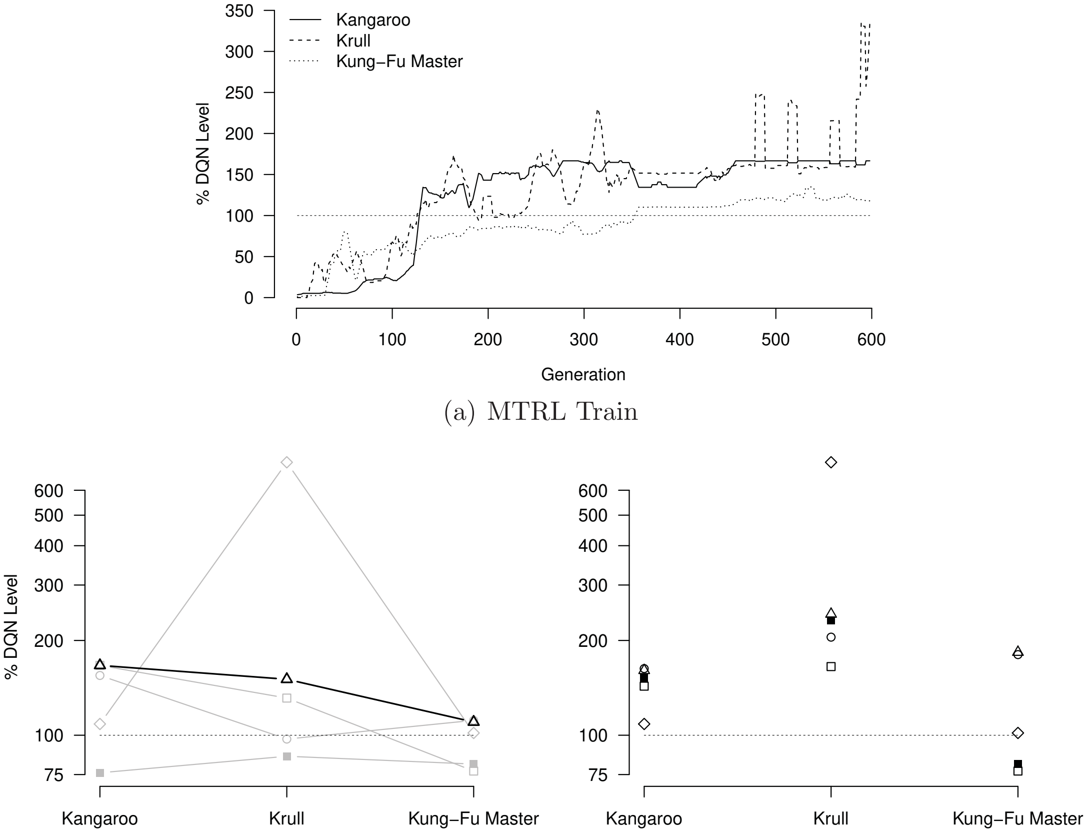


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.2: TPG multi-task reinforcement learning results for game group 3.3. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

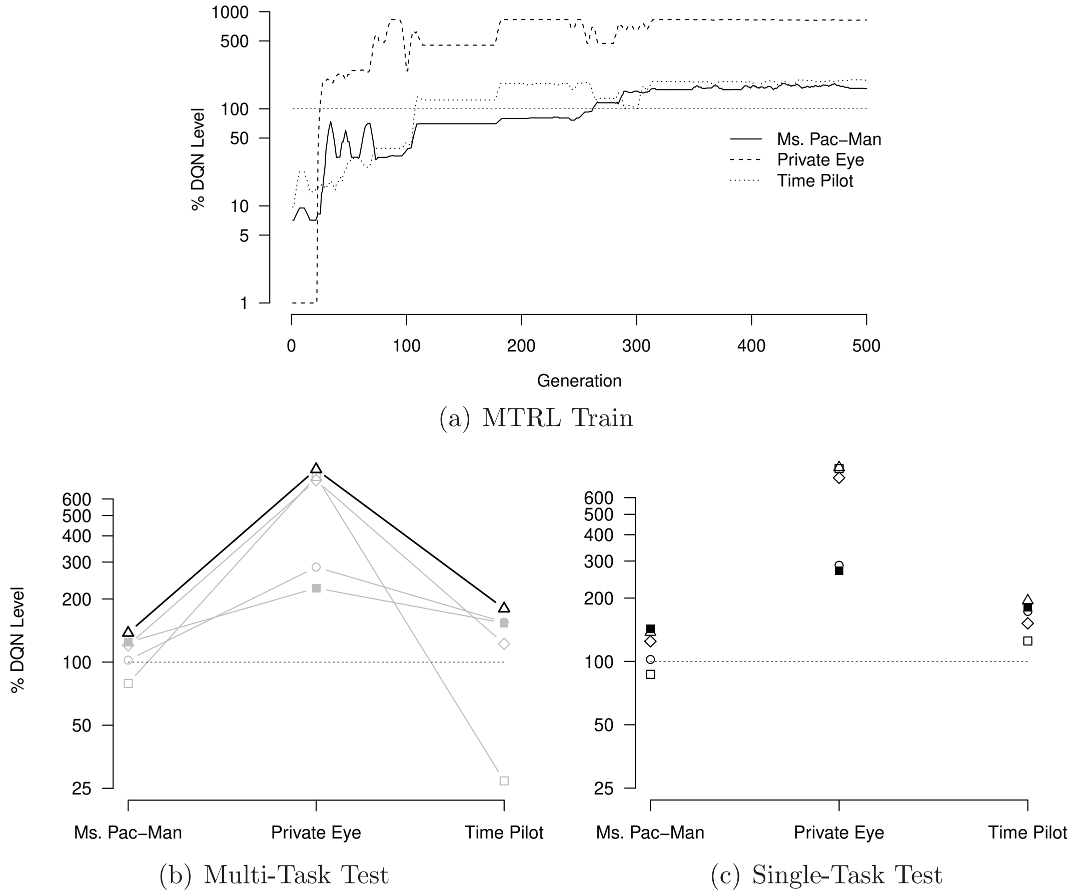


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.3: TPG multi-task reinforcement learning results for game group 3.4. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

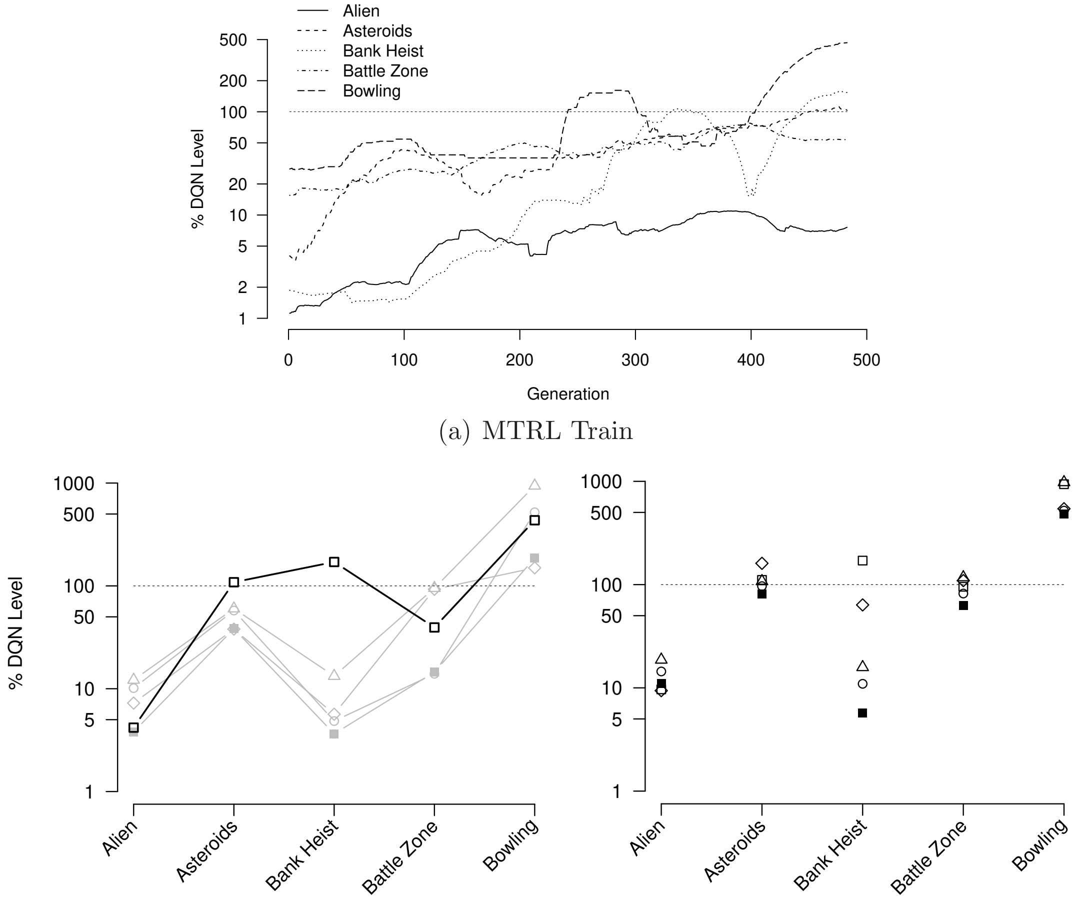


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.4: TPG multi-task reinforcement learning results for game group 3.5. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

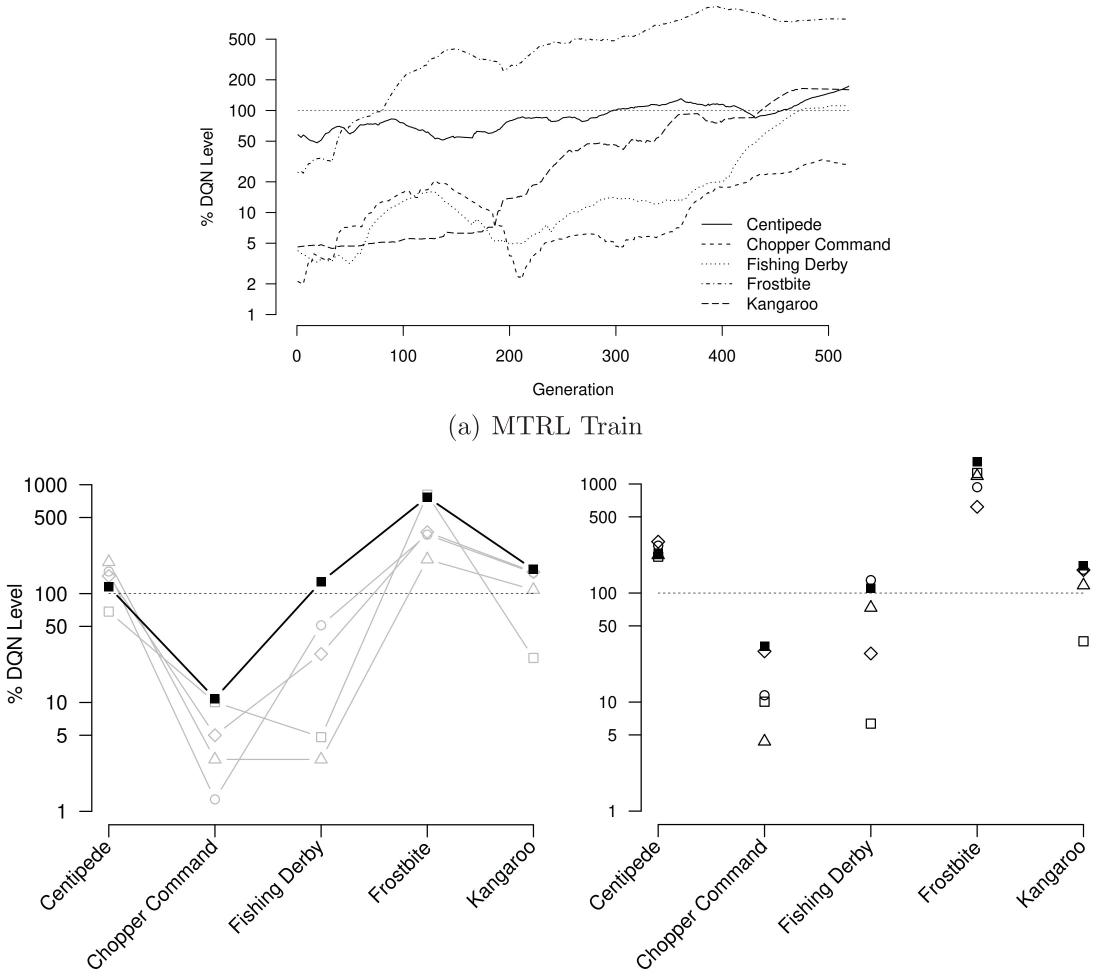


(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.5: TPG multi-task reinforcement learning results for game group 5.1. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].



(a) MTRL Train

(b) Multi-Task Test

(c) Single-Task Test

Figure F.6: TPG multi-task reinforcement learning results for game group 5.2. Each run identifies one elite multi-task policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multi-task policy over all 5 independent runs. Note that *multi-task* implies that the scores reported at each generation are all from the same policy. Test scores for the final multi-task champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title (The line connecting points in (b) emphasizes that scores are from the same multi-task policy). DQN scores are from [100].

Appendix G

Additional Multi-Task Policy Graphs

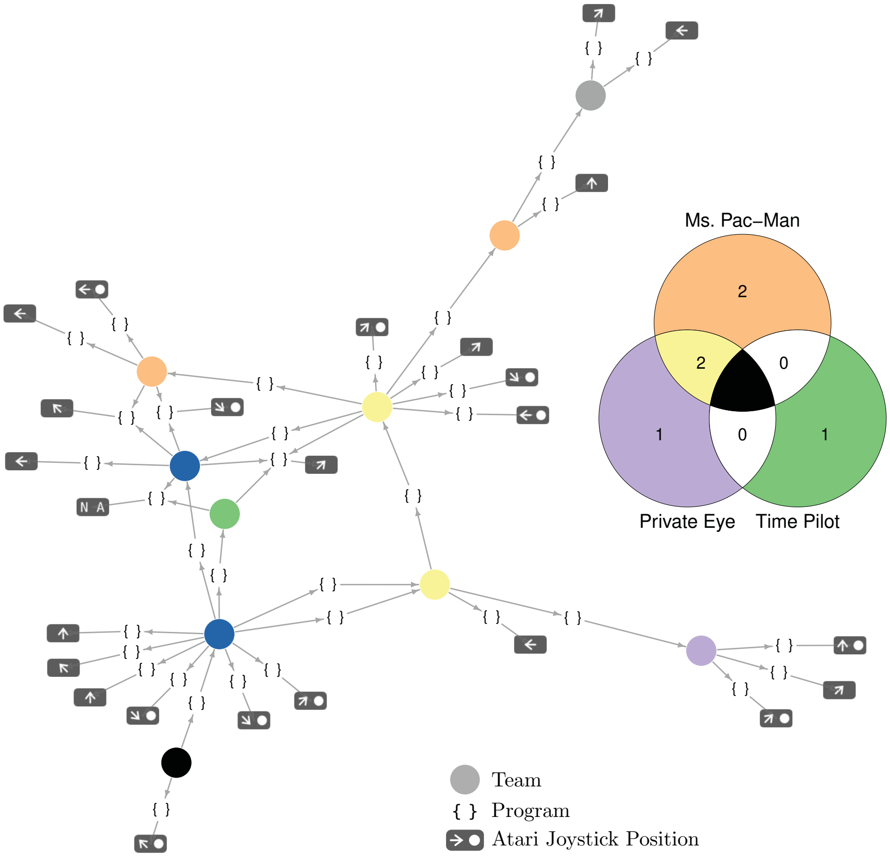


Figure G.1: Champion multi-task TPG policy graph from the group 3.5 experiment. Decision-making in a policy graph begins at the root node (black circle) and follows *one* path through the graph until an atomic action (joystick position) is reached (See Algorithm 9). Venn diagram indicates which teams are visited while playing each game, over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision-making during test are shown. Figure requires viewing in colour.

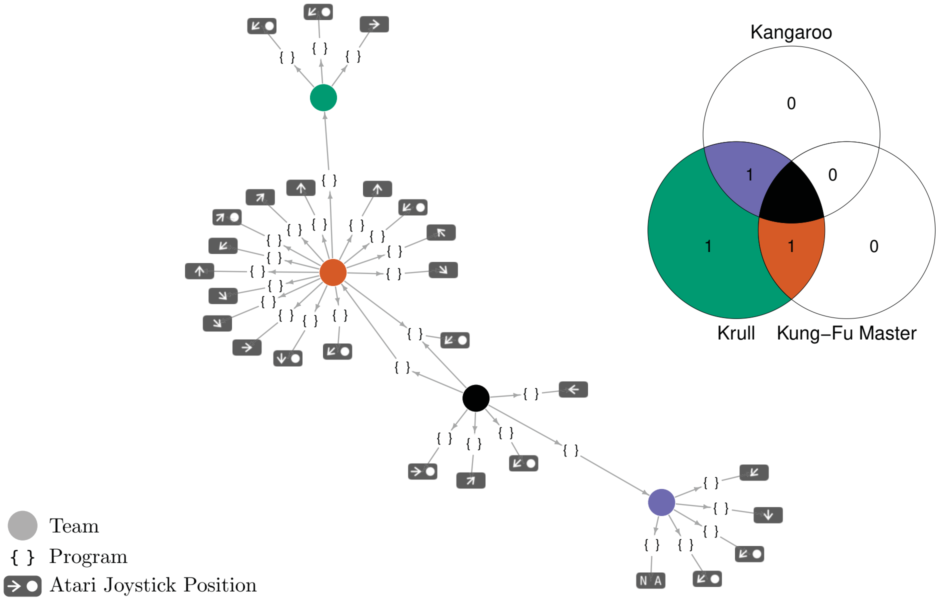


Figure G.2: Champion multi-task TPG policy graph from the group 3.4 experiment. Decision-making in a policy graph begins at the root node (black circle) and follows *one* path through the graph until an atomic action (joystick position) is reached (See Algorithm 9). Venn diagram indicates which teams are visited while playing each game, over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision-making during test are shown. Figure requires viewing in colour.