Evolving Multimodal Behavior Through Modular Multiobjective Neuroevolution

Jacob Benoid Schrum

Report TR-14-07 May 2014

schrum2@cs.utexas.edu
http://www.cs.utexas.edu/users/nn/

Artificial Intelligence Laboratory
The University of Texas at Austin
Austin, TX 78712

The Dissertation Committee for Jacob Benoid Schrum
certifies that this is the approved version of the following dissertation:

Evolving Multimodal Behavior Through Modular Multiobjective Neuroevolution

Committee:

Risto Miikkulainen, Supervisor

Dana Ballard

Joydeep Ghosh

Raymond Mooney

Peter Stone

Evolving Multimodal Behavior Through Modular Multiobjective Neuroevolution

by

Jacob Benoid Schrum, B.S.; M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2014

To Elissa, for her patience

Acknowledgments

I would like to start by thanking my research supervisor, Risto Miikkulainen, from whom I have learned much about how to present ideas, be it textually, visually, or verbally. I have also benefitted from his research advice: Of the many meetings I have had with him over the years, I have left each one with many new ideas to try and hypotheses to test.

I would also like to thank my committee, Peter Stone, Raymond Mooney, Dana Ballard, and Joydeep Ghosh, for their support and suggestions. I would like to thank Peter Stone in particular for giving me a chance to be something more than a TA for his honors Discrete Math course by designing some of the course content that distinguished the honors version of the course from the standard course.

Additionally, I would like to thank current and former members of the Neural Networks Research Group at UT that have listened to my many talks and given me constructive criticism that has helped me both find ways to make my research work, and to present my ideas effectively to others. In particular, I would like to thank Igor Karpov, with whom I worked many hours to create and test the software agent that eventually won the BotPrize 2012 competition.

I would also like to thank my friends who have known me since my undergraduate days before I started at UT: Chris Tanguay, Amy Tanguay, Shane Baumgartner, Rebecca Baumgartner, Larcy Levins, and most importantly my wife, Elissa Schrum, who has gracefully tolerated my addiction to work over the years, while also encouraging me to take a break every once in a while. Thank you all for encouraging me to have a life outside of my teaching and research.

Lastly, I would like to thank my Mom, who has wanted to tell people that I'm a "doctor" for years. I finally made it!

This research was supported in part by NSF grants DBI-0939454, IIS-0915038, SBE-0914796, IIS-0757479, and EIA-0303609, by NIH grant R01-GM105042, by Texas Higher Education Coordinating Board grant 003658-0036-2007, and by Google, Inc.

JACOB BENOID SCHRUM

The University of Texas at Austin
May 2014

Evolving Multimodal Behavior Through Modular Multiobjective Neuroevolution

Publication No. _____

Jacob Benoid Schrum, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Risto Miikkulainen

Intelligent organisms do not simply perform one task, but exhibit multiple distinct modes of behavior. For instance, humans can swim, climb, write, solve problems, and play sports. To be fully autonomous and robust, it would be advantageous for artificial agents, both in physical and virtual worlds, to exhibit a similar diversity of behaviors. This dissertation develops methods for discovering such behavior automatically using multiobjective neuroevolution. First, sensors are designed to allow multiple different interpretations of objects in the environment (such as predator or prey). Second, evolving networks are given ways of representing multiple policies explicitly via modular architectures. Third, the set of objectives is dynamically adjusted in order to lead the population towards the most promising areas of the search space.

These methods are evaluated in five domains that provide examples of three different types of task divisions. Isolated tasks are separate from each other, but a single agent must solve each of them. Interleaved tasks are distinct, but switch back and forth within a single evaluation. Blended tasks do not have clear barriers, because an agent may have to perform multiple behaviors at the same time, or learn when to switch between opposing behaviors. The most challenging of the domains is Ms. Pac-Man, a popular classic arcade game with blended tasks. Methods for developing multimodal behavior are shown to achieve scores superior to other Ms. Pac-Man results previously published in the literature. These results demonstrate that complex multimodal behavior can be evolved automatically, resulting in robust and intelligent agents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An impressive quality of the human brain is the way it is applied to solving a wide variety of tasks, from the physical (running, swimming, climbing, sports) to the mental (solving puzzles, reading, communicating, planning). Though there is occasional overlap between some of these activities, the fact that our capacity to do any one of them generally does not interfere with our ability to do the others is impressive.

Such abilities do not begin with humans. All but the most simplistic of organisms share our capacity for a diverse array of behaviors: Birds must fly, forage, and build nests in order to propagate their species, and leopards stalk their prey, chase it down, and then hide leftovers in trees in order to survive. Multimodal behavior — the ability to exhibit multiple different, even dissimilar behavioral modes — is a fundamental feature of intelligence.

Therefore, it is natural to desire artificial agents to exhibit multimodal behavior, be they physical robots or software agents. Many complex, highly engineered systems already exhibit multimodal behavior that depends on human-specified divisions of tasks into subtasks. However, the performance of such agents always depends on the accuracy of the assumptions made by the human designers: Did the designers pick the right number of subtasks? Is the division between tasks the best possible division for the problem? Are the sub-behaviors used in each task optimal? Are both the sub-behaviors and aggregate behavior robust? The answers to such questions are often uncertain, which is why the prospect of learning multimodal behavior, rather than designing it, is appealing.

This dissertation develops an approach to such learning, using evolution of neural networks. This chapter motivates multimodal learning, discusses some of the challenges, introduces the approach developed in this dissertation, and provides an outline for the rest of the dissertation.

1.1 Motivation

A primary goal of Artificial Intelligence is to construct intelligent artificial agents. Such agents are needed in many contexts. RoboCup Soccer requires agents that strategically maneuver, pass the ball to teammates, and try to block kicks from opponents (Chen et al., 2013). Agents in First-Person Shooter video games must explore levels, take cover, and coordinate attacks with teammates (Hingston, 2012). Autonomous vehicles must handle a variety of driving circumstances: driving on highways, driving in traffic within cities, parking, and responding to unexpected obstacles such as debris or careless pedestrians (Buehler et al., 2009). Search-and-rescue robots are needed to find people in need of help at disaster sites, to remove or shore up rubble, and deliver supplies (Murphy et al., 2008).

Although it would be useful for search-and-rescue robots to be completely autonomous, they are currently teleoperated. However, over 50% of the failures of such robots in the field are due to human error (Murphy, 2014). Another example of human error as the primary cause is traffic accidents. In the United States from 2005 to 2007, the critical reason for over 90% of car crashes was the driver (National Highway Traffic Safety Administration, 2008). It would therefore be extremely beneficial to have intelligent agents that perform better than humans in such situations.

Although there are many challenges to overcome in these domains, one feature they all have in common is the need for multimodal behavior: An artificial agent must exhibit multiple behavioral modes depending on the circumstances in order to succeed.

1.2 Challenge

The intelligent agents in the above domains need to exhibit multimodal behavior. Developing such behavior is not easy. Sometimes the border between behaviors is clear (e.g. a driving agent knows when it is getting onto a highway via an on-ramp), and at other times it is unclear (a car may be maneuvering around traffic while also trying to dodge debris). Sometimes humans have a good idea of what a good policy should do, but at other times we either do not know what is best, or our preconceptions of what constitutes good behavior prevent us from discovering even better behaviors.

These domains can all be modeled as Reinforcement Learning (RL) problems in which an agent interacts with its environment and learns from numeric reward signals (Michie and Chambers, 1968; Sutton and Barto, 1998; Deisenroth et al., 2012; Szita and Lőrincz, 2006; Moriarty et al., 1999; Kohl and Miikkulainen, 2009). Specifically, a control policy is learned. This policy maps an agent's observations of the state space to actions.

Most RL algorithms explicitly rely on the formalism of Markov Decision Processes (MDPs), which is discussed in Section 4.1. However, in a domain requiring multimodal behavior, it is difficult for a single policy to handle the different modes of behavior. For this reason, there has been much research in Hierarchical RL, in which agents have access to high-level, temporally-extended

actions. Access to such actions can make tasks easier to solve, if they effectively solve subtasks within the domain. There are several approaches to defining such temporally-extended actions: options (Sutton et al., 1999), skills (Konidaris and Barto, 2009; Konidaris et al., 2010), activities (Barto and Mahadevan, 2003), modes (Alur et al., 2000), and behaviors (Huber and Grupen, 1997; Brooks, 1986). These approaches offer different perspectives on the same Hierarchical RL problem. This dissertation presents a different perspective; these temporally-extended actions are behavioral modes encapsulated within modules, and an agent exhibiting such behavior is said to exhibit multimodal behavior. This perspective does not depend on the Hierarchical RL formalism, so a comparison with these other methods is postponed until Chapter 11.

How best to learn multiple modes of behavior is an interesting open challenge, the impact of which is magnified by other challenges common in domains requiring intelligent behavior: partial observability (Sutton and Barto, 1998), continuous state and action spaces (Gaskett et al., 1999), and noisy evaluations. Taking on these challenges, this dissertation develops methods specifically aimed at discovering multimodal behavior.

1.3 Approach

A variety of methods exist for learning agent behavior in RL and Hierarchical RL problems. Two major paradigms are Value-Function Learning and Policy Search. Value-Function Learning attempts to define a function that estimates the long-term expected return (accumulation of rewards) of either reaching specific states or taking particular actions in each state (Bellman, 1957). The resulting value-function is then used to define a policy that favors the actions leading to the highest expected returns. In contrast, Policy Search methods (de Boer et al., 2005; Stanley and Miikkulainen, 2002) search the space of policies directly. These methods aggregate reinforcement signals, and can often achieve better results, particularly when function approximation is necessary, or when the domain is partially observable (Kalyanakrishnan and Stone, 2009). A prominent example of the Policy Search approach is Evolutionary Computation, specifically Neuroevolution.

Neuroevolution simulates the process of evolution by natural selection to construct neural networks that serve as control policies for agents (Stanley and Miikkulainen, 2002; Gomez et al., 2006). Neural networks are powerful function approximators that can learn robust behavior (Haykin, 1999). Additionally, recurrent neural networks can also be evolved, which makes it possible to discover policies that retain a memory of past states.

This dissertation builds on pre-existing Neuroevolution algorithms in order to produce multimodal behavior reliably. First, techniques for evolving neural networks are combined with evolutionary multiobjective optimization, so that behavioral modes that correspond to different objectives each receive individual focus during the search process. Second, networks are allowed to evolve modular architectures that associate particular modules with distinct behavioral modes. Third, the multiobjective search process is enhanced by Targeting Unachieved Goals, an approach that dy-

namically manages which objectives are used during selection so that objectives that need it most receive more focus.

1.4 Outline

This dissertation is organized as follows:

Chapter 2 presents the methodological foundations upon which multimodal evolution is built. These pre-existing methods are Pareto-based Evolutionary Multiobjective Optimization and Constructive Neuroevolution. Specifically, the Non-Dominated Sorting Genetic Algorithm II (NSGA-II; Deb et al., 2002) is used to perform multiobjective optimization, and Neuro-Evolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen, 2002) is the basis for the constructive neuroevolution method used in this dissertation.

Chapter 3 builds upon these foundational methods to discover multimodal behavior. The main technical contributions are presented in this chapter. They are (1) a set of general design principles for sensors/features (Conflict sensors vs. Split sensors) that make learning particular task divisions easier, (2) a collection of methods for designing modular network architectures (Multitask Learning, Preference Neurons, Module Mutation) that can learn unexpected and advantageous task divisions, and (3) a means of manipulating multiobjective evolutionary search (Targeting Unachieved Goals) so that discovering multiple modes of behavior is more reliable.

Chapter 4 gives a high-level overview of the types of domains that require multimodal behavior. Specifically, three types of task divisions are identified: (1) isolated tasks that are completely independent from each other, (2) interleaved tasks that alternate along a shared time line, and (3) blended tasks where the division between one task and another is unclear.

Chapter 5 describes the BREVE simulation environment used to create several domains for the experiments in later chapters. Two of these domains consist of isolated tasks: Front/Back Ramming involves two tasks where evolved agents must exhibit two different types of aggressive behavior, and Predator/Prey consists of one task requiring aggressive behavior and another requiring defensive behavior. The third domain is the Battle Domain, which consists of blended offensive and defensive tasks.

Chapter 6 describes experiments in Front/Back Ramming and Predator/Prey. These experiments demonstrate the ability of modular networks to succeed in domains with isolated tasks.
Specifically, isolnde9(ma-sSpecited)-312(tasisolnd(di)25(vision)-312((Mult(tasisolndLlearning)-312(and)-Ind0Mult(ne
task division discovededvolution Module Mutation opes Predator/Preye
isolated tasksmernical in dif

Chapter describes expert in the Battle Domai(.)-519(These)-642(e)15(xperiments)-642(demonstrate)-642(th

iors not discovered without TUG.

Chapter 8 describes the classic arcade video game of Ms. Pac-Man, which is the second simulation environment in which experiments are conducted, to demonstrate that the methods scale up to real-world games. Ms. Pac-Man is well-suited for demonstrating multimodal behavior because the enemies in the game can be in either a threat state or vulnerable state, each of which requires a different response from the player. This chapter explains why Ms. Pac-Man is challenging, reviews previous research in the domain, and then describes both a variant with interleaved tasks and variants with blended tasks, in which experiments are conducted in the following chapters.

Chapter 9 describes experiments in the Imprison variant of Ms. Pac-Man, which has interleaved tasks. Several types of modular network architectures are evaluated, showing that while there is little difference between modular and non-modular approaches when using split sensors, all modular methods are superior to a non-modular approach when the more general conflict sensors are used. Additionally, the best modular networks succeed by discovering a novel, unexpected task division that dedicates a module to a clever luring behavior.

Chapter 10 presents experiments in the full Ms. Pac-Man game. In experiments where Ms. Pac-Man has only a single life, the best modular networks triumph once again by discovering a luring module. Then experiments with multiple lives, as in the complete game, demonstrate how modular networks and TUG can be combined to consistently reach a high level of performance. These results are also compared to previous results in the literature, and shown to be superior in nearly all cases.

Chapter 11 puts the methods developed in the dissertation into the context of the research literature. In particular, previous approaches to developing multimodal behavior are sorted into three categories: (1) hand-designed hierarchical approaches, (2) methods that learn or learn parts of an explicit hierarchy, and (3) methods that create modular structures that split the domain in a hierarchical way. The modular network architectures developed in this dissertation fall into the third category.

Chapter 12 uses the related work of the previous chapter and the results from earlier in the dissertation as a jump-off point for a deeper discussion of the results. This discussion leads to directions for future work, such as methods for automatically adapting the sensors used in a domain, ways of combining various types of modular architectures from this dissertation, ideas for improving TUG, and similar but different ways of generating multimodal behavior via Generative and Developmental Systems.

Chapter 2

Foundations

The two main tools on which this dissertation builds in order to discover multimodal behavior are Pareto-based multiobjective optimization and constructive neuroevolution. Each of these tools is discussed within this chapter.

2.1 Multiobjective Optimization

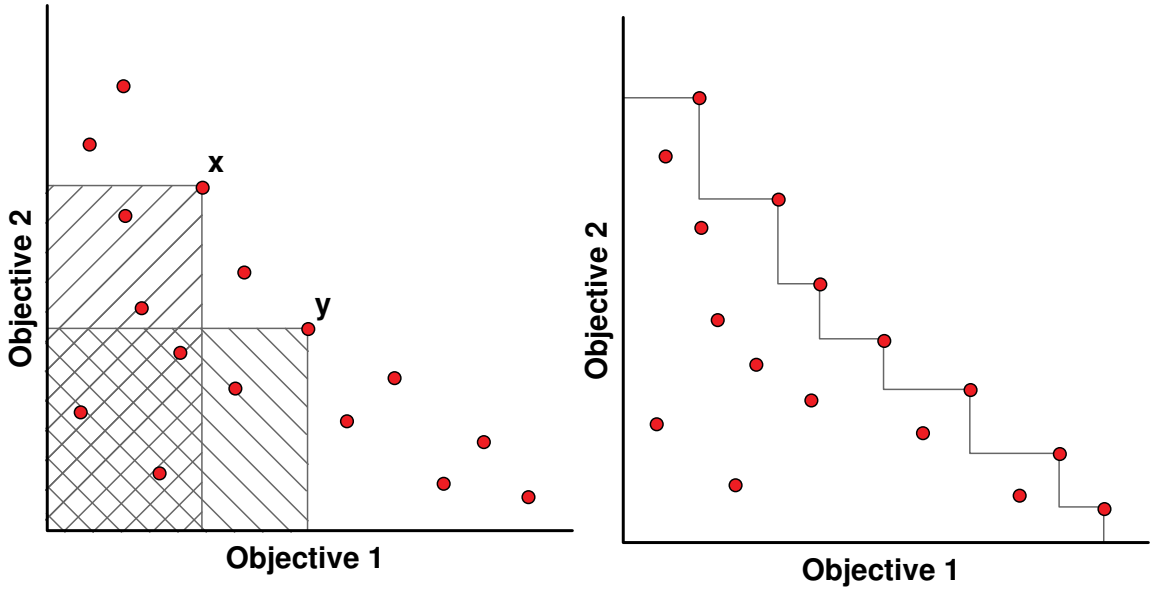
Domains are likely to require multimodal behavior when they have multiple, conflicting objectives. For example, in a scenario where both offensive and defensive behaviors are needed at different times, there are likely to be separate objectives favoring each of these behaviors. In the specific case of Ms. Pac-Man, which is discussed in depth in Chapter 8, rewards are given both for eating pills and eating ghosts, which means that different behaviors are required to satisfy each of these objectives.

Therefore a principled way of dealing with multiple objectives is needed. Such an approach is offered by the framework of Pareto optimality.

2.1.1 Pareto Optimality

The concept of Pareto optimality is an approach to dealing with multiple objectives that allows one to avoid the design pitfalls inherent in aggregating objectives (e.g. how to weight objectives). It also has theoretical benefits with respect to the types of solutions that are attainable. Specifically, the alternative weighted-sum approach cannot capture Pareto optimal points on non-convex surfaces (Coello, 1999; Fonseca and Fleming, 1995). In practice, a multiobjective approach can often find better overall performance than simply optimizing a single combined objective (Schrum and Miikkulainen, 2008; Lochtefeld and Ciarallo, 2012). The concepts of Pareto dominance and optimality provide the framework for multiobjective optimization[1] (Figure 2.1):

[1] These definitions assume a maximization problem. Objectives to be minimized can simply be multiplied by -1.



(a) Pareto Dominance

(b) Pareto Front

Figure 2.1: **Pareto Optimality.** The concept of Pareto optimality is demonstrated in an example domain consisting of two objectives. Each point in the space represents an individual solution that has earned the corresponding scores in objectives 1 and 2. (a) Point **x** dominates solutions in the area contained within the shaded box that extends down and to the left of **x**. Similarly, point **y** dominates points within the shaded box associated with it. The area where the shaded boxes overlap is dominated by both solutions. (b) The same points are shown with a line connecting the Pareto front. The region below and to the left of this line is dominated by the Pareto front. These points are non-dominated, which makes them all Pareto optimal. These points are the best in terms of the two objectives shown. None are strictly better than the others, but collectively they are better than all other points in the space.

Definition 1 (Pareto Dominance) *Vector* $\vec{v} = (v_1, \ldots, v_n)$ *dominates* $\vec{u} = (u_1, \ldots, u_n)$ *if and only if the following conditions hold:*

1. $\forall i \in \{1, \ldots, n\} : v_i \geq u_i$, *and*
2. $\exists i \in \{1, \ldots, n\} : v_i > u_i$.

The expression $\vec{v} \succ \vec{u}$ *denotes that* $\vec{v}$ *dominates* $\vec{u}$.

Definition 2 (Pareto Optimal) *A set of points* $\mathcal{S} \subseteq \mathcal{F}$ *is Pareto optimal if and only if it contains all points such that* $\forall \vec{x} \in \mathcal{S}$: $\neg\exists \vec{y} \in \mathcal{F}$ *such that* $\vec{y} \succ \vec{x}$. *The points in* $\mathcal{S}$ *are non-dominated, and make up the non-dominated Pareto front of* $\mathcal{F}$.

The above definitions indicate that one solution is better than (i.e. dominates) another solution if it is strictly better in at least one objective and no worse in the others. The best solutions are not dominated by any other solutions, and make up the Pareto front of the search space. The next best individuals are those that would be in a recalculated Pareto front if the actual Pareto front were removed first. Layers of Pareto fronts can be defined by successively removing the front and recalculating it for the remaining individuals. Solving a multiobjective optimization problem involves approximating the *first* Pareto front as well as possible.

There are usually several points in the search space that are optimal in this sense, which makes it sensible to to search for these points using a population-based approach, namely evolutionary computation. Several evolutionary approaches to multiobjective optimization have been developed (Corne et al., 2000, 2001; Zitzler and Thiele, 1999), but the most popular is the Non-Dominated Sorting Genetic Algorithm II (NSGA-II; Deb et al., 2002).

2.1.2 Non-Dominated Sorting Genetic Algorithm II

NSGA-II uses $(\mu + \lambda)$ elitist selection favoring individuals in higher Pareto fronts (i.e. closer to the true Pareto front) over those in lower fronts. In the $(\mu + \lambda)$ paradigm, a parent population of size μ is evaluated, and then used to produce a child population of size λ. Selection is performed on the combined parent and child population to give rise to a new parent population of size μ. NSGA-II typically uses $\mu = \lambda$.

When performing selection based on which Pareto layer an individual occupies, a cutoff is often reached such that the layer under consideration holds more individuals than there are remaining slots in the next parent population. These slots are filled by selecting individuals from the current layer based on a metric called "crowding distance," which encourages the selection of individuals in less-explored areas of the trade-off surface between objectives.

The crowding distance for a point p in objective space is the average distance between all pairs of points on either side of p along each objective. Points having an objective score that is the maximum or minimum for the particular objective are considered to have a crowding distance of infinity, though if there is a tie then only one such point is (randomly) chosen. For other points, the crowding distance tends to be bigger the more isolated the point is. NSGA-II favors solutions with high crowding distance during selection, because the more isolated points in objective space are filling a niche in the trade-off surface with less competition.

By combining the notions of non-dominance and crowding distance, a total ordering of the population is obtained: Individuals in different layers are sorted based on the dominance criteria, and individuals in the same layer are sorted based on crowding distance. The resulting comparison operator for this total ordering is also used by NSGA-II: Each new child population is derived from the parent population via binary tournament selection based on this comparison operator.

A run of NSGA-II creates an approximation to the true Pareto front, i.e. an approximation set. This approximation set potentially contains multiple solutions, which must be analyzed in order

to determine which solutions fulfill the needs of the user. However, in order to assess the overall performance of a method for generating approximation sets, special multiobjective performance metrics have been developed.

2.1.3 Assessing Multiobjective Performance

Individual objective scores and statistics based on them are misleading because high scores in one objective can be combined with low scores in other objectives. Comparing approximation sets directly reveals whether one dominates another, but this approach does not scale to a large number of comparisons. Furthermore, if different approximation sets cover non-intersecting regions of objective space, it is still unclear which one is better. Multiobjective performance metrics help by reducing an approximation set to a single number that gives some indication of its quality.

All of these measures involve first normalizing the scores achieved within the Pareto fronts to the range $[0, 1]$ with respect to minimum and maximum objective scores. The specific minimums and maximums used depend on the domain. The role of normalization in interpreting the results of each metric is very important, and is explained below. The normalized objective scores are used to calculate two types of metrics: hypervolume (Zitzler et al., 2007) and unary epsilon indicator values (Knowles et al., 2006).

Hypervolume

Hypervolume (HV) is the primary performance measure used in Chapters 6 and 7. It measures the region dominated by all points in an approximation set with reference to some point that is dominated by all points in the set. For example, if an approximation set consisted of a single solution, and the reference point were the zero vector, its hypervolume would be the product of all normalized objective scores, i.e. the volume of the hypercube between the solution and the reference point. When more points are in the approximation set, hypervolume measures the size of the union of the hypercubes between each solution and the reference point.

Because each objective is scaled to the range $[0, 1]$, hypervolume is also restricted to this range. A hypervolume close to 0 thus has nearly minimum performance in all objectives, while a hypervolume close to 1 has nearly maximum performance in all objectives. For a domain with strongly conflicting objectives, hypervolumes close to 1 are unlikely, since high performance in some objectives is traded for low performance in others. Solutions that have high scores in multiple objectives will contribute more to hypervolume than solutions with high scores in some objectives but low scores in the others.

Hypervolume is particularly useful because it is Pareto-compliant (Zitzler et al., 2007), meaning that an approximation set that completely dominates another approximation set will have a higher hypervolume. The opposite is not true: An approximation set with higher hypervolume does not necessarily dominate one with lower hypervolume, since each set could dominate non-

intersecting regions of objective space. In fact, it is provably impossible to construct a unary indicator that tells when one approximation set dominates another (Zitzler et al., 2002). Therefore, it is important to compare results using other metrics as well, to assure that these results corroborate rather than contradict the hypervolume results. The additional metrics used are two variants of the unary epsilon indicator.

Epsilon Indicators

Like hypervolume, both epsilon indicators are Pareto-compliant (Knowles et al., 2006). For epsilon indicators an approximation set that dominates another approximation set will have a $lower$, rather than a higher, epsilon indicator score.

The two flavors of unary epsilon indicator are multiplicative and additive. The multiplicative indicator I_ϵ^1 measures by how much each objective for each solution in a set would have to be multiplied such that each solution in a reference set R would be dominated by or equal to a point in the resulting set. The set R should be chosen such that it dominates all fronts under consideration. Therefore, an I_ϵ^1 value of 1 corresponds to R itself, which is in turn the best/lowest value possible. The additive indicator $I_{\epsilon+}^1$ measures how much would have to be added to each objective in each solution such that each point in R would be dominated by or equal to a point in the modified set. In this case, the best/lowest $I_{\epsilon+}^1$ value is 0, the value for R again. For both indicators, smaller values are better because they indicate that a smaller adjustment is needed to dominate the reference set R. As suggested by Knowles et al. (2006), when using epsilon indicators to measure performance in a domain, the reference set R used is the super Pareto front (Pareto front of several Pareto fronts) of all fronts for which epsilon values are being calculated and compared.

As with hypervolume, epsilon indicator values are calculated for normalized objective scores. For the epsilon indicators, the purpose of normalization is to make the different objective score ranges comparable. For example, imagine a two-objective problem where the objective ranges are $[0, 1]$ and $[100, 200]$. Consider two approximation sets consisting of one point each: $A = \{(0.1, 200)\}$ and $B = \{(1.0, 110)\}$. With respect to the scales for each objective, these points are trade-offs at exact opposite ends of objective space, and therefore of equal quality (assuming no objective preferences). However, if the epsilon indicator values are calculated without normalizing first, the following results are obtained: $I_\epsilon^1(A) = 10$, $I_{\epsilon+}^1(A) = 0.9$, $I_\epsilon^1(B) = 1.8181$, and $I_{\epsilon+}^1(B) = 90$. Not only are differences between like metrics inappropriately large, but results across indicators are inconsistent: Front A has a better $I_{\epsilon+}^1$ value, but B has a better I_ϵ^1 value. Had normalization been used, each Pareto front would have equal scores in like metrics.

Combined with hypervolume, the epsilon indicators provide a thorough analysis of how the approximation sets discovered in a given domain cover the space of all objectives. For each of these metrics, a better score does not guarantee a superior approximation set, but superior scores in all metrics gives confidence that a given approximation set actually is better.

Up to this point, solutions to a multiobjective problem have been discussed only in terms

of the objective scores they achieve. How these scores are achieved does not matter for NSGA-II, and neither does how solutions are represented. The next section describes an effective method for automatically generating complex control policies.

2.2 Neuroevolution

In order to learn multimodal behavior, a means of representing complex control policies is required. Artificial neural networks are in theory capable of approximating any behavior with arbitrary accuracy if the weights and topology are correctly set (Haykin, 1999). Also, evolutionary computation has proven to be a useful method for learning how to configure the weights and topologies of neural networks in order to solve many problems (Floreano and Urzelai, 2000; Yao and Liu, 1997; Stanley, 2003; Miikkulainen et al., 2012). When evolutionary computation is used to evolve artificial neural networks, it is called neuroevolution.

2.2.1 Constructive Neuroevolution

A particularly successful approach to evolving neural networks is to start with simple genotypes that gradually complexify throughout the course of evolution. This approach is called constructive neuroevolution, and was popularized by the Neuro-Evolution of Augmenting Topologies (NEAT) algorithm (Stanley and Miikkulainen, 2002; Stanley, 2003).

In this approach, networks start with minimum structure and become more complex from mutations across several generations. The initial population consists of networks with no hidden layers, i.e. only input and output neurons. Inputs to read sensor values and outputs to determine the actions of an agent are the minimal structure necessary to define a control policy. It is common to begin with every input connected to every output, but a feature-selective approach, in which each output starts out connected to only a single input, is also possible (Whiteson et al., 2005).

Three mutation operators were used to change network behavior. Weight mutation perturbs the weights of existing network connections, link mutation adds new (potentially recurrent) connections between existing nodes, and node mutation splices new nodes along existing connections. Recurrent connections are particularly useful in partially observable domains. An environment is partially observable if the current observed state cannot be distinguished from other observed states without memory of past states (Sutton and Barto, 1998). Recurrent connections help in these situations because they encode and transmit memory of past states; this property could help a network determine which mode of behavior is most appropriate for the current situation.

The algorithm described so far is very similar to NEAT (Stanley and Miikkulainen, 2002). However, since NEAT was designed to use only a single fitness function, in this dissertation it made more sense to reimplement these features of NEAT in NSGA-II than to modify NEAT to use a Pareto-based multiobjective approach. Another key innovation of NEAT that is used in this dissertation is topological crossover based on historical markers. Every new link and neuron that

is introduced by mutation is given a unique innovation number that identifies it. The genotype that encodes each neural network stores these innovations linearly in an order that is consistent across all members of the population. This representation makes it easy to align components with a shared origin within different genotypes, thus making crossover between networks computationally inexpensive.

Although components with a shared origin can be aligned, not all components in two different networks will share common origins. Crossover is only happening between synaptic weights that can be aligned, but components that do not align are left alone. To maintain topological diversity, both children of each crossover are kept (provided there is room in the new population). Each child will have the topology of a different parent, but the crossover of aligned weights will make the children different from the parents, even if no further mutations occur. Parents are chosen via binary tournament selection.

Although topological crossover via historical markers was an impressive innovation in NEAT, it is not used in all experiments in this dissertation. In particular, experiments using feature-selective networks do not use crossover (Chapters 6 and 7). Preliminary experiments indicated that crossover was not helpful in these situations. A potential reason is that feature-selective networks will tend to have fewer links that actually align, thus making crossover superfluous at best, and potentially detrimental if it disrupts the fragile weight structure of the few weights that align.

The next section gives an example of how constructive neuroevolution, modelled after NEAT, can sometimes give rise to multimodal behavior by taking advantage of simple tricks.

2.2.2 Multimodal Behavior via Neuroevolution

The form of neuroevolution described so far has been used to solve many challenging problems (Stanley and Miikkulainen, 2002; Kohl and Miikkulainen, 2009; Schrum and Miikkulainen, 2010). Although this approach does not explicitly encourage the development of multimodal behavior, it has produced agents that can be claimed to exhibit multiple modes of behavior. One particularly interesting example comes from Stanley, Bryant, and Miikkulainen's work in what they called the Dangerous Foraging Domain (2003).

This domain models the situation faced by a foraging animal entering a new geographical region. The foraging animal cannot know in advance whether the food in this region is safe to eat or not, therefore it must taste one food item first before it has a chance of making appropriate decisions. In the domain, all food in a given trial is either safe or unsafe, so the optimal behavior is to try one piece of food, and then take different actions according to whether the food was safe or not. If it was safe, then all food should be eaten. Otherwise, all food should be avoided. This task is made challenging by the fact that the network only receives brief pain and pleasure signals upon eating unsafe and safe food respectively. The fact that this signal is brief rather than persistent makes the domain partially observable.

Networks using recurrent connections did surprisingly well in this domain by taking advan-

tage of a simple trick: A recurrent connection activated by the pain sensor caused the robot to spin until it was facing away from food. This example demonstrates how useful recurrent connections are in helping a network change its output behavior. It also shows how multimodal behavior may sometimes evolve without the aide of methods specifically aimed at doing so, by using simple tricks.

The Dangerous Foraging Domain thus underlines how important it is to evaluate the methods developed in this dissertation in domains that require true multimodal behavior and are not solvable by simple trick solutions. Other previous attempts to learn true multimodal behavior, both with and without neuroevolution, are discussed in the related work chapter at the end of this dissertation (Chapter 11).

2.3 Conclusion

Pareto-based multiobjective optimization and neuroevolution, particularly constructive neuroevolution, have been demonstrated previously to be useful tools for solving challenging problems. However, in order to reliably learn multimodal behavior in challenging domains, these methods need to be extended in ways described in the next chapter. In particular, neural networks need architectures that support multiple policies via multiple modules, and multiobjective evolutionary search needs a way to focus search on the most relevant areas of objective space on any given generation of evolution.

Chapter 3

Evolving Multimodal Behavior

In order to evolve multimodal behavior reliably, the tools described in the previous chapter need to be extended. This chapter describes several such extensions. They constitute the main technical contributions of this dissertation, and will be thoroughly evaluated in later chapters.

Three such contributions are described in this chapter. The first is a set of general design principles for sensors/features that make learning particular task divisions easier. The second is a collection of methods for designing modular network architectures that can learn unexpected and advantageous task divisions even when the sensors are not configured in such a beneficial way. The third is a means of manipulating multiobjective evolutionary search so that discovering multiple modes of behavior is more reliable.

3.1 Role of Sensors

The hardest of problems can be made trivially easy with the right choice of sensors. In the extreme case, imagine a complex sensor that simply provides the optimal output. Any learning method would simply need to develop an easy-to-learn identity function in order to have perfect behavior. In this case, no matter how complex or multimodal the observed behavior was, the policy itself would actually be simple.

Such a scenario is not common, and even in the rare cases where a perfect oracle is available, such problems are not of interest to researchers of Artificial Intelligence. However, the lesson to be learned from this example is that the choice of sensors can obscure any contributions made by learning algorithms. Algorithms of widely varying complexity would likely reach perfect performance quickly in any task with such a sensor configuration.

Although it makes sense to construct complex sensors to solve challenging problems, this dissertation provides methods for developing multimodal behavior even when intelligent sensor configurations are not available. Additionally, this dissertation points out some general properties of sensors that can make learning multimodal behavior more challenging. Such knowledge can

help in the design of more learning-friendly sensors in those cases where some degree of sensor manipulation is possible.

3.1.1 Conflict Sensors

A nasty property of some sensors that has an important impact on results in Ms. Pac-Man (Chapters 9 and 10) is that they can sometimes have different, conflicting interpretations.

At the lowest level, sensor values are simply numbers. As these values are propagated through a neural network, their influence may be magnified or reduced. These values may also be swapped from positive to negative or negative to positive as a result of negative network weights. Sometimes there are clear correlations between sensor values and advantageous actions: Agents in nearly any domain should generally move towards nearby entities that will increase fitness and away from nearby entities that will decrease fitness. If there are obstacles or other extenuating circumstances, then sensors that detect these circumstances should temper the general attraction or repulsion that the agent's policy exerts.

For example, an agent moving forward to a goal may need to turn slightly to the left or reduce its speed to get around an obstacle. Learning such adjustments is not trivial, but it is certainly not outside the reach of most modern learning algorithms. In this case, the sensors that result in the most obvious attraction toward fitness-increasing entities will have a strong influence on behavior, and other sensors whose importance is more circumstantial will simply refine that general behavior.

However, a problem arises when certain entities can alternate between being fitness-enhancing and fitness-decreasing. A sensor that, for example, detects the distance to such entities, will be difficult to learn how to use properly, because it needs to influence the agent's behavior in conflicting ways: encouraging both approach and retreat in different circumstances. Even if both of these opposing impulses travel through different neural pathways, when they eventually reach the output layer of the network, one will need to be completely canceled out in order for the agent to behave optimally.

If the right network topology and synaptic weights are learned, in which hidden neurons properly produce useful aggregate features, then it is possible to completely cancel out competing influence in this manner, but such structures are sufficiently complicated that they are not learned easily. Fortunately, the learning task can be made much easier by using networks with a modular structure that allows these opposing influences to be handled in different ways. Such modular networks are a major topic of this dissertation that will be discussed momentarily (Section 3.2), but first another type of sensor configuration is discussed that contributes towards the same goal.

3.1.2 Split Sensors

If a sensor has multiple different interpretations, then one way to make it easier to learn with it is to replace it with one sensor for each possible interpretation.

Definition 3 (Split Sensors) *Suppose $\phi : \mathcal{S} \rightarrow \mathbb{R}$ is a conflict sensor mapping states from the state space $\mathcal{S}$ to numeric sensor values. This sensor has n different interpretations that are identified by $\beta : \mathcal{S} \rightarrow \{1, \ldots, n\}$, a function that indicates how ϕ should be interpreted in each state. Then n split sensors, $\theta_1, \ldots, \theta_n : \mathcal{S} \rightarrow \mathbb{R}$, can be defined using n constants, $C_1, \ldots, C_n$, as follows.*

For $i \in \{1, \ldots, n\}$, and $s \in \mathcal{S}$:

$$\theta_i(s) = \begin{cases} \phi(s), & \text{if } \beta(s) = i \\ C_i, & \text{otherwise} \end{cases}$$

It is up to the sensor designer to provide a β and values for $C_1, \ldots, C_n$ that are appropriate to the domain.

For example, a single sensor returning the distance to an entity that is sometimes fitness-enhancing and sometimes fitness-decreasing can be replaced with two sensors: one for the fitness-enhancing case and the other for the fitness-decreasing case.

Such sensors are split sensors, because the potentially conflicting ways of interpreting one sensor's values have been split across two sensors. However, several issues must be considered: Is it clear whether each entity is currently fitness-enhancing vs. fitness-decreasing? Are these the only opposing interpretations of the sensor's values? What values will the resulting split sensors return when the entity they sense is not in the right state?

The answers to the first two questions depend on the specific domain and sensor. The third question can be answered more generally: If, for example, there are no fitness-enhancing entities, then a sensor pertaining to such entities should return a constant value. Assuming all sensor values are scaled to the range $[0, 1]$, then a value of 0 or 1 will usually be most appropriate. For example, a distance sensor should return a value of 1 for entities that are not present, because they are effectively an infinite distance away. Other sensors may more appropriately return a 0 in such circumstances. So, some knowledge about how the sensor works still needs to be incorporated, but the need for the sensor to return a constant value when entities of interest are absent is universally important. It enables the learning system to ignore the sensor when necessary. Sometimes it is obvious how to define such useful sensors, and at other times extra effort may be required to design such sensors; such issues are domain-dependent.

For each interpretation of sensor values encoded by the split, split sensors can make it easier to learn a separate mode of behavior. However, such sensors also bias evolution towards that particular task division. If a better set of behavioral modes exists that depends on splitting up the domain in a different way, then the introduction of such sensors will not help, and may even hurt learning.

In order to learn novel, unexpected modes of behavior, networks need to have the freedom to interpret sensors as the domain dictates. Such freedom is easily attained if networks have multiple output modules.

3.2 Network Modules

A network module is a set of output neurons that are capable of defining the behavior of an agent. These neurons are called policy neurons because one set defines one policy. A single network can consist of multiple modules, and therefore multiple policies. The overall policy of the agent is defined according to an arbitration mechanism that chooses which module to use on each time step.

This section introduces several ways to design such network modules, along with a means of arbitration between modules. The Multitask Learning approach uses a human-specified number of modules and task division, preference neurons allow the division to be learned, and Module Mutation allows the number of modules to be learned as well.

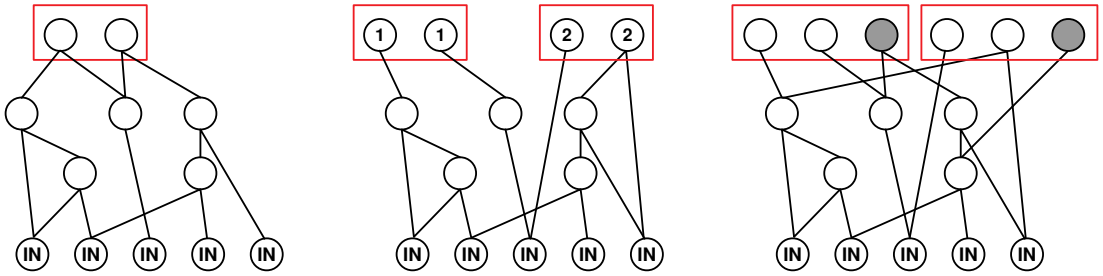
3.2.1 Multitask Learning

Multitask networks were first proposed by Caruana (1993, 1997) in the context of supervised learning using neural networks and backpropogation. One network has multiple modules, where each module corresponds to a different, yet related, task (Figure 3.1b). Each module is trained on the data for the task to which it corresponds, but because hidden-layer neurons are shared by all outputs, knowledge common to all tasks can be stored in the weights of the hidden layer. This approach speeds up supervised learning of multiple tasks, or even just a single task of interest, because knowledge shared across tasks is only learned once and shared, rather than learned independently multiple times. Multitask Learning has been expanded on in many ways (Bakker and Heskes, 2003; Liu et al., 2009; Collobert and Weston, 2008), but using this general architecture to evolve agent behavior is a new approach.

Although Multitask Learning is a powerful technique, there are known problems with it. The first restriction is the need to identify the individual tasks to learn. The appropriate decision is not always obvious, and divisions that seem obvious may not actually be so. In fact, inappropriate task choices can hurt learning. In the supervised learning contexts where Multitask Learning is commonly applied, even when there is certainty regarding how to divide the tasks, there can be uncertainty regarding which tasks are related enough to benefit from sharing information. For this reason, methods have been developed to learn how tasks should share information (Thrun and O'Sullivan, 1998; Kang et al., 2011).

In the context of constructive neuroevolution, Multitask Learning imposes a strict task division and assumes that evolving agents are always aware of the task they currently face, but imposes no bias in the degree of information sharing between tasks. Each network has a module for each task, but these modules are initially connected only to input neurons; the modules can only share information if they evolve to share hidden neurons.

As an example of the multitask architecture, consider a domain where two policy neurons are required to define the behavior of an agent, and the agent must solve two tasks. The networks have two policy neurons for each task, for a total of four outputs. When performing a given task,



(a) Single-module Network (b) Multitask Network (c) Preference Neuron Network

Figure 3.1: **Network Architectures With a Fixed Number of Modules.** The neural networks in this figure are intended for a task with five input sensors (bottom) and behavior defined by two policy neurons. Each complete module of output neurons is contained within its own red box. (a) A neural network with only one module has only two output neurons at the top. (b) Multitask network with two modules, each consisting of two policy neurons. The numbers indicate which module is used with each task. (c) Network with two modules that use preference neurons (grey). Each module has its own preference neuron, and the module with the highest preference neuron output defines the network's behavior on each time step. These network architectures demonstrate different ways that a single network can have multiple modules, but the number of modules in each case is fixed (cf. Figure 3.2).

the agent bases its behavior on the outputs corresponding to the current task, and ignores the other outputs.

Multitask Learning can supply a learning system with a helpful bias, but this bias will only be useful if it is appropriate for the domain at hand. The task split imposed by Multitask Learning is very similar to the bias imposed by split sensors (Section 3.1.2). In both cases, even when the human-specified division is helpful, it may not be the best division. In order to discover potentially better task divisions, a means of learning how to arbitrate between tasks is needed.

3.2.2 Preference Neurons

Preference neurons allow a task division to be learned. This approach requires that each network module possess an additional neuron called a preference neuron (Figure 3.1c). The preference neurons in a network indicate the relative preference for using the corresponding module. On each time step, the network module whose preference neuron output is highest is used to define the behavior of the agent (Algorithm 1).

For example, assume a domain requires two outputs to designate the behavior of an agent, and a network has two modules. Then the network has six outputs: two policy neurons and one preference neuron for Module 1, and two policy neurons and one preference neuron for Module 2.

Algorithm 1: Execution of a Network With Preference Neurons

Data: n = number of policy neurons per module, m = number of modules, networkOutputs = array of values from all network output neurons from all modules; for each module, the outputs of the policy neurons come first, followed by the preference neuron output.

Result: behaviorOutputs = array containing only the outputs of policy neurons for the module chosen by the preference neurons.

for $i \leftarrow 1$ **to** m **do**
  modulePreferences [i] $\leftarrow$ networkOutputs [$i(n+1)$];
chosenModule $\leftarrow \arg\max_i$ modulePreferences [i];
for $i \leftarrow 1$ **to** n **do**
  policyIndex $\leftarrow$ (chosenModule $-1)(n+1)+i$;
  behaviorOutputs [i] $\leftarrow$ networkOutputs [policyIndex];

Whenever the output of Preference Neuron 1 is higher than the output of Preference Neuron 2, the two policy neurons of Module 1 define the behavior of the agent. Otherwise, the policy neurons of Module 2 are used.

The use of preference neurons assumes that there is enough information in the state space of the problem to determine which module should be used. Technically, Multitask Learning does not require this information to be available in the sensors. However, it does assume that a human designer knows how to determine the current task, which is actually a more complicated requirement. Still, the use of modules dictated by the preference neurons will depend on the available sensors.

The architecture described so far still assumes that a human designer specifies the number of modules to use. If a good guess at the number of modules cannot be made, one option is to simply give a network lots of modules, and hope that it learns to ignore those it does not need. However, adding many extra modules needlessly increases the size of the search space, which also removes some of the benefits of constructive neuroevolution (Section 2.2.1). Therefore, this dissertation proposes a new way to introduce network modules gradually, called Module Mutation.

3.2.3 Module Mutation

Module Mutation (known as Mode Mutation in previous work; Schrum and Miikkulainen, 2009, 2011, 2012) refers to any structural mutation operator that adds a new module to a neural network. Because an unknown number of modules may be added in this way, such networks depend on preference neurons for module arbitration. New populations start with a single module and a preference neuron that only becomes relevant after more modules are added. Each new module from Module Mutation adds a new set of policy neurons and a new preference neuron.

Calabretta et al. (2000) proposed similar ideas: Neural networks were evolved to control robots using a "duplication operator," which created a copy of one output neuron with all of its

connections and weights. The network then had two output neurons that corresponded to the same actuator on the robot. Arbitration was accomplished by "selector units," which were essentially the same as the preference neurons described above. This duplication operator differs from Module Mutation in two major ways: (1) The number of neural modules per output neuron was limited to two, so for any given output, only one duplication operation was allowed. (2) The duplication operation works at the level of individual output neurons, whereas Module Mutation works at the level of groups of output neurons.

There are several ways in which Module Mutation can be implemented. Two of them, evaluated in the BREVE simulation environment (Chapter 6), are Module Mutation Previous (MM(P); Figure 3.2b) and Module Mutation Random (MM(R); Figure 3.2c).

The intention behind MM(P) is to create a new output module with minimal additional structure that is similar in behavior, at least initially, to a pre-existing module (Algorithm 2). Therefore, each neuron within the newly added module starts with one input synapse that comes from the corresponding neuron of the previous output module. These connections are lateral, from left to right in the same layer, but are treated as feed-forward connections (i.e. they transmit on the same time step). The weights of these connections are set to 1.0, but the new module is not identical to the previous module because the $\tanh$ activation function is used on each neuron, and acts as a squashing function (which is a common design in neural networks). Therefore, the new module is a similar but slightly diminished (in terms of activation) version of the previous module. Future mutations can further differentiate the new module from its source module such that both modules exhibit distinct behavior.

Algorithm 2: Module Mutation Previous

Data: n = number of policy neurons per module, m = number of modules, links = array of all synaptic links in network stored in arbitrary order, neurons = array of all neurons in the network stored in strict order: all input neurons come first, followed by all hidden neurons, followed by all output neurons for each module; for each module, the policy neurons come first, followed by the preference neuron, outputStart = index of the first output neuron in neurons.

sourceModule $\leftarrow$ `rand` (m);
for $i \leftarrow 1$ **to** $(n+1)$ **do**
  $o \leftarrow$ `newOutputNeuron` ();
  $l \leftarrow$ `newLink` ();
  l.weight $\leftarrow$ 1.0;
  sourceLocation $\leftarrow$ outputStart $+$(sourceModule $-1)(n+1) + (i-1)$;
  l.source $\leftarrow$ neurons [sourceLocation];
  l.target $\leftarrow o$;
  neurons.`addToEnd` (o);
  links.`addToEnd` (l);
$m \leftarrow m+1$;

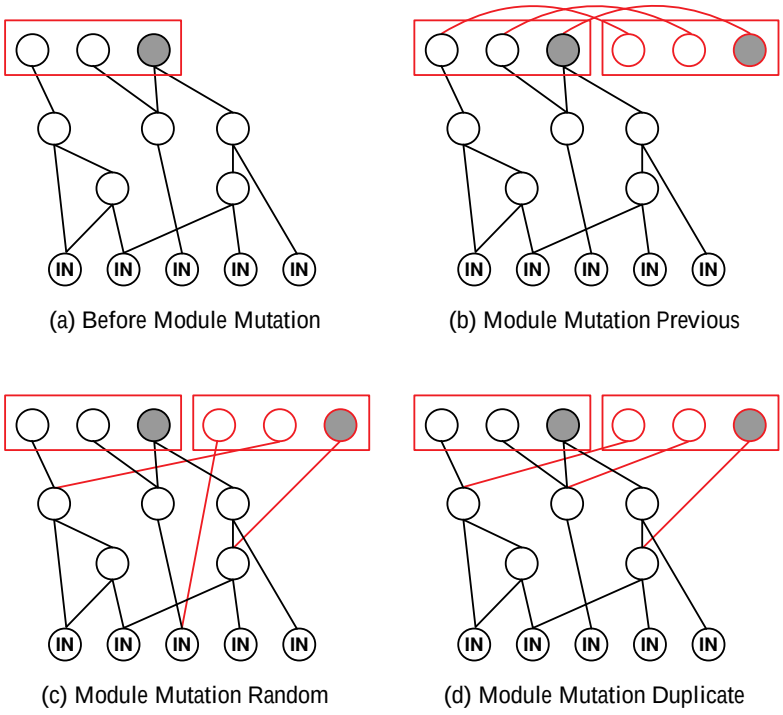


Figure 3.2: **Types of Module Mutation.** As in Figure 3.1, the neural networks shown here are intended for a task with five input sensors (bottom) and behavior defined by two policy neurons. All preference neurons are grey, and new structure that results from Module Mutation is drawn in red. Each complete module of output neurons is contained within its own red box. (a) Neural network at the start of a run using Module Mutation. The lone preference neuron becomes relevant after more modules are added. (b) MM(P) creates a new module whose inputs come directly from the previous module. (c) MM(R) creates a new module whose inputs come from random sources, and have random weights. (d) MM(D) creates a new module whose behavior exactly duplicates that of a previous module. Notice that the inputs to the new policy neurons in MM(D) come from the same sources that lead to the policy neurons of the pre-existing module. Their weights are the same as well. However, the new preference neuron has an input from a random source with a random weight. Each of these mutation operators provides a different way to create new modules in a neural network, which can then learn specialized behavior corresponding to interesting behavioral modes.

However, such differentiation is not guaranteed to occur. Because new modules are similar to old modules, there may not be enough selection pressure for them to change, meaning that they may persist indefinitely. Furthermore, despite the capacity for evolved connections to differentiate each module, older modules will always have some influence over later modules via the lateral connections created along with each new module, thus making it hard to evolve lasting modular behavior.

The second method of Module Mutation addresses this problem. In new MM(R) modules, each neuron receives randomly weighted inputs from random hidden and input neurons in the network (Algorithm 3). For policy neurons, the number of these inputs equals the number of inputs each output neuron had in the original network from the start of evolution. Connecting the new outputs in this way assures that new modules start out as complex as the module with which the network started evolution. In contrast, preference neurons only ever start with a single random input, since this design allows evolution to more easily experiment with different ways of using the modules. This whole approach is risky since a new random module could cause fitness scores to plummet, but it has the advantage of more quickly introducing *distinct* network modules.

MM(R) also makes it feasible to delete network modules. Deleting an MM(P) module is often infeasible, because the modules are tightly interconnected and a deletion would often disconnect neurons from the network. Specifically, since new modules start out connected only to the previous module, deletion of an MM(P) module can potentially disconnect all modules added after that module. Even if links had evolved that would prevent these newer modules from becoming disconnected, the deletion of the lateral links connecting the deleted module to the next module could drastically change the behavior of all of these modules, which would usually be undesirable.

However, modules can be safely deleted in MM(R) networks without disconnecting other modules. The ability to delete old modules after better ones evolve can be advantageous. Therefore, this module-deletion mutation will be evaluated in some of the experiments below.

The third form of Module Mutation is inspired by the work of Calabretta et al. (2000): Module Mutation Duplicate (MM(D); Figure 3.2d). In MM(D), the sources and synaptic weights of inputs to the policy neurons in the new module are chosen so that it exactly duplicates the behavior of a randomly chosen previous module (Algorithm 4). This approach improves upon MM(P) in two ways: (1) New module behavior exactly copies a previous module rather than merely approximating it, making it impossible for the new mutation to decrease fitness, and (2) the new module is not directly linked to the module it duplicates, which means that future changes to the source module will not damage any useful behaviors learned by the newer module. As with MM(R), the preference neurons in MM(D) are initialized with a single random input, to encourage the new module to be used in different situations than its predecessor.

The different forms of Module Mutation are powerful because they search the space of policies in ways that are not possible with a single module. MM(P) elaborates on past modules to refine behavior, MM(R) quickly discovers completely new modules, and MM(D) makes it easy

Algorithm 3: Module Mutation Random

```
Data: n = number of policy neurons per module, m = number of modules,
      links = array of all synaptic links in network stored in arbitrary order,
      neurons = array of all neurons in the network stored in strict order: all input
      neurons come first, followed by all hidden neurons, followed by all output
      neurons for each module; for each module, the policy neurons come first,
      followed by the preference neuron, linksPerOutput = number of new random
      links that each new policy neuron receives.
newModule ← [];
for i ← 1 to n do
    o ← newOutputNeuron ();
    for j ← 1 to linksPerOutput do
        l ← newLink ();
        l.weight ← rand ();
        l.source ← neurons [rand (neurons.length)];
        l.target ← o;
        if ¬∃p ∈ links (p.source = l.source ∧p.target = l.target ) then
            links.addToEnd (l);
    newModule.addToEnd (o);
o ← newOutputNeuron ();
l ← newLink ();
l.weight ← rand ();
l.source ← neurons [rand (neurons.length)];
l.target ← o;
links.addToEnd (l);
newModule.addToEnd (o);
for i ← 1 to (n + 1) do
    neurons.addToEnd (newModule [i]);
m ← m + 1;
```

Algorithm 4: Module Mutation Duplicate

```
Data: n = number of policy neurons per module, m = number of modules, links = array
      of all synaptic links in network stored in arbitrary order, neurons = array of all
      neurons in the network stored in strict order: all input neurons come first,
      followed by all hidden neurons, followed by all output neurons for each module;
      for each module, the policy neurons come first, followed by the preference
      neuron, outputStart = index of the first output neuron in neurons.
sourceModule ← rand (m);
newModule ← [];
for i ← 1 to n do
    sourceLocation ← outputStart +(sourceModule −1)(n + 1) + (i − 1);
    o ← newOutputNeuron ();
    for j ← 1 to links.length do
        if links [j].target = neurons [sourceLocation ] then
            l ← newLink ();
            l.weight ← links [j].weight;
            l.source ← links [j].source;
            l.target ← o;
            links.addToEnd (l);
    newModule.addToEnd (o);
o ← newOutputNeuron ();
l ← newLink ();
l.weight ← rand ();
l.source ← neurons [rand (neurons.length)];
l.target ← o;
links.addToEnd (l);
newModule.addToEnd (o);
for i ← 1 to (n + 1) do
    neurons.addToEnd (newModule [i]);
m ← m + 1;
```

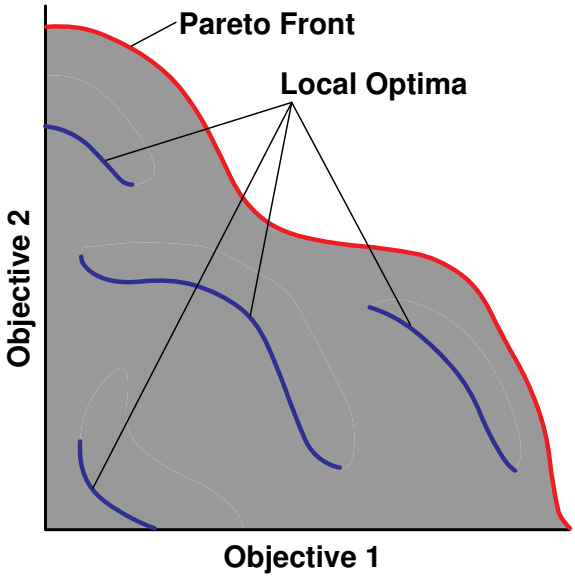


Figure 3.3: **Locally Optimal Fronts.** The gray region defines the feasible region of objective space in this two-objective problem. Any feasible point can potentially be achieved within the given domain. In addition to the Pareto front, there are also locally optimal fronts where points may converge. In order to reach the Pareto front, search must either get around or leap over these local optima. Regular NSGA-II may sometimes have trouble following such trajectories, so a way of shaping the search process is needed to focus on the best solutions.

for different modules to branch off from a common starting point. This dissertation evaluates the benefits of each approach in the experiments below.

However, the survival of individuals within an evolving population depends on fitness values and how evolution deals with them. Straightforward multiobjective evolution will preserve the best solutions when they are discovered, but may not always take the most direct route through the search space needed to reach the best solutions. Therefore, a means of improving the navigation of search spaces in conjunction with multiple objectives is introduced in the next section.

3.3 Targeting Unachieved Goals

NSGA-II is designed to search for solutions in domains with multiple objectives. The upper bounds on the scores that can be achieved in each objective are defined by the Pareto front, i.e. the collection of optimal points in objective space. However, as with single-objective search, local optima can exist in objective space. These locally optimal fronts can pose a problem by acting as road blocks on the path to the Pareto front (Figure 3.3).

Intelligent ways of navigating the multiobjective search space can help get around these lo-

cal optima. Specifically, focusing on objectives in which the population as a whole is performing poorly is a way of pushing past sub-optimal regions to find the Pareto front. This is what Targeting Unachieved Goals (TUG) does: Only objectives in which the population is having trouble performing should be part of the selection process; objectives in which the entire population is performing well can be ignored.

To know when to deactivate an objective, a numeric goal is defined for each objective. These values represent the currently desired levels of performance, and indicate how well an agent would have to perform to be considered successful. Initial goal values can be set at minimal starting points, or can be chosen based on domain knowledge. Additionally, goals increase over time as the population improves across all objectives.

A goal is considered achieved once the average performance of the population in that objective has *persisted* long enough at a level above the value of the goal. Persisting above the goal means both the average performance and a recency-weighted average of that average have surpassed their objective's goal value. The persistence requirement is important because the fitness values often fluctuate significantly, particularly in domains with noisy evaluations.

Formally, a recency-weighted average r_t at time t is updated according to

$$r_{t+1} \leftarrow r_t + \alpha(\bar{x}_{t+1} - r_t), \tag{3.1}$$

where α defines the portion of the distance between r_t and the current actual average, denoted by $\bar{x}_{t+1}$, by which r_t should be increased. Thus the recency-weighted average moves slightly closer to the most recent average of every generation.

Note that an objective can be reactivated if performance drops back below its goal. Goals reactivate as soon as the *actual* average drops below the goal, since the recency-weighted average catches up too slowly in comparison. As a result, any level of performance should be maintained once it is achieved.

To push the level of performance higher and higher, the goal values increase once they are all achieved. This step can be accomplished in a variety of ways. Obviously, the amount by which each objective increases could be set based on domain knowledge. A more general approach can be used if each objective has a maximum value: Choose a number of steps n and divide the maximum achievable score in each objective by n to get the step size for that objective. This approach is evaluated in Ms. Pac-Man (Chapter 10).

An alternative approach that can be used without knowledge of maximum values or deciding on a number of steps is the following: When increasing goals for each objective, move the current goal closer to the current maximum score in that objective. Formally, the update rule is

$$g_o \leftarrow g_o + \eta(o_{\max} - g_o), \tag{3.2}$$

where g_o is the goal for objective o, $o_{\max}$ is the current maximum score within the population for objective o, and η defines the portion of the distance between the current goal and the current

Algorithm 5: TUG in a Multiobjective Evolutionary Algorithm

Data: α = step-size parameter for recency-weighted averages, n = number of objectives, η = optional step-size parameter for goal values, $\Delta_1, \ldots, \Delta_n$ = alternative fixed goal increments for each objective, $\min_1, \ldots, \min_n$ = minimum possible score in each objective (or reasonable estimates), $g_1, \ldots, g_n$ = initial goal values for each objective.

```
population ← initialPopulation();
rwas ← (min_1, ..., min_n);
goals ← (g_1, ..., g_n);
while evolving do
    evaluate(population);
    for i ← 1 to n do
        maxes [i] ← population's maximum score in objective i;
        avgs [i] ← population's average score in objective i;
        rwas [i] ← rwas [i] + α(avgs [i] - rwas [i]);
        achieved [i] ← (rwas [i] > goals [i]) ∧ (avgs [i] > goals [i]);
    if all entries in achieved are True then
        for i ← 1 to n do
            if using fixed goal increments then
                goals [i] ← goals [i] + Δ_i;
            else
                goals [i] ← goals [i] + η(maxes [i] - goals [i]);
            useObjective [i] ← True;
            rwas [i] ← min_i;
    else
        for i ← 1 to n do
            useObjective [i] ← ¬achieved [i];
    for each individual's scores s = (s_1, ..., s_n) in population do
        for i ← 1 to n do
            if ¬useObjective [i] then
                s[i] ← −∞;
    population ← selectNextGeneration(population);
```

maximum by which the goal should be increased. This approach increases the goal values without overshooting the capabilities of agents within the domain. The maximum score in an objective will always be above the average, which will always be above the goal at the moment it is achieved, so this update rule will always increase goals. Because goals are moved towards the current maximum score, no goal will ever be set at a level that is unattainable. This approach is evaluated in Chapter 7.

Regardless of *how* goals are increased, each increase only occurs when all goals are achieved.

Chapter 4

Domains Requiring Multimodal Behavior

Multimodal behavior is required in many domains, some more complicated than others. This chapter first describes the Markov Decision Process (MDP) formalism that is the basis of Reinforcement Learning (RL) problems (Sutton and Barto, 1998). The neuroevolution approach to RL domains functions independently from this formalism — it is not a component of the algorithms developed in Chapter 3 — but including it provides a basis of comparison to the work from the non-evolutionary RL community. This formalism is then used to define three ways in which a domain may divide into tasks. They may be isolated, interleaved, or blended, depending on how clear the border between tasks is. The resulting order of increasing difficulty provides a logical sequence in which methods for learning multimodal behavior will be evaluated in this dissertation.

4.1 Markov Decision Process

An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ is the state space of the environment, $\mathcal{A}$ is the set of actions that the agent is able to perform, $\mathcal{P}$ is the transition function, and $\mathcal{R}$ is the reward function. The transition function is defined as $\mathcal{P} : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \rightarrow [0, 1]$ where $\mathcal{P}(s, a, s')$ returns the probability of reaching state s' on the next time step after performing action a in state s. The reward function provides the agent with feedback on how well it is performing in the environment. Typical RL formulations use a scalar reward function, but the domains in this dissertation explicitly use multiple objectives. Thus, $\mathcal{R} : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathbb{R}^N$ is defined such that $\mathcal{R}(s, a)$ returns a tuple of the expected immediate rewards in each objective for performing action a in state s. This tuple has length N, which is the number of objectives. In fact, this N will be included in the definition of the resulting multiobjective MDP to make the discussion below easier to follow: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, N)$

All domains in this dissertation are episodic, meaning that they all terminate in finite time. Because evolutionary computation is being used, the rewards from $\mathcal{R}$ must be translated into fit-

ness. Therefore, the fitness for a given objective is the sum of the reward accrued in the course of an episode[1]. Because every episode has a start and end point, specific start and end states can be identified in $\mathcal{S}$. Strictly speaking, there can be multiple start and end states, but these can be abstracted away by using a single master start state that randomly transitions to one of the actual start states, and a master end state to which all actual end states involuntarily transition. These master start and end states will simplify the discussion of how tasks are divided.

It is worth pointing out that many interesting domains (such as those in this dissertation) are actually Partially Observable MDPs (POMDPs; Sutton and Barto, 1998), meaning that states cannot be distinguished without a perfect memory of all past states visited. The extra challenges of POMDPs slightly complicate the formalism provided so far, but these complications can be safely ignored in the discussion below, because the observability of states does not directly impact whether or not the domain consists of separate tasks. All POMDPs have underlying MDPs, even though agents in such domains may not be aware of them. Therefore, when a domain is actually a POMDP, it can still be categorized according to the following definitions if they apply to the domain's underlying MDP.

The first type of domain consists of multiple, isolated tasks, as discussed next.

4.2 Isolated Tasks

In the simplest multimodal domains, each task is completely isolated from the others (Figure 4.1): Nothing that an agent does in one task affects the other tasks.

Domains with isolated tasks can be constructed by taking any set of T distinct domains $(\mathcal{S}_1, \mathcal{A}_1, \mathcal{P}_1, \mathcal{R}_1, N_1), \ldots, (\mathcal{S}_T, \mathcal{A}_T, \mathcal{P}_T, \mathcal{R}_T, N_T)$ and combining them into one MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, N)$. In this combined MDP,

$$\mathcal{S} = \bigcup_{i=1}^{T} \mathcal{S}_i. \tag{4.1}$$

Note that $\{\mathcal{S}_1, \ldots, \mathcal{S}_T\}$ is a partition of $\mathcal{S}$. This requirement is imposed even if states from different MDPs are seemingly identical, since the source MDP of a given state is part of what defines that state: Two states from different MDPs are different by virtue of being from different MDPs. Of course, an agent may not know which MDP a state came from. This issue relates to POMDPs: States from different tasks may look identical to an agent, but are distinct because the tasks are distinct. The action space is defined similarly as

$$\mathcal{A} = \bigcup_{i=1}^{T} \mathcal{A}_i, \tag{4.2}$$

[1] Rewards are also often discounted such that early rewards are higher than later rewards. Such discounting is not necessary in episodic domains, and is not used in this dissertation.

Figure 4.1: **Isolated Tasks.** An abstract representation of a domain with two isolated tasks. In such a domain, nothing that occurs in any task can affect any other task. Each task has its own time line of events. Evaluation order could be changed, or evaluations could be done in parallel (assuming multiple copies of a software agent) without the outcome of evaluation being any different (ignoring any noise in evaluation). In fact, if a learning agent is aware of the task split, one could even learn completely separate policies for each task.

but in this case there is no strict partition requirement. In fact, in a situation where one would actually want to have one agent learn multiple isolated tasks, it makes sense for the action spaces of each task to be the same. Such is the case for all domains in this dissertation, but is not a requirement of the definition. The transition function for the combined MDP is defined as

$$\mathcal{P}(s, a, s') = \mathcal{P}_i(s, a, s') \text{ if } s, s' \in \mathcal{S}_i \text{ and } a \in \mathcal{A}_i. \tag{4.3}$$

However, if $s \in \mathcal{S}_i$ and $s' \in \mathcal{S}_j$ for $i \neq j$, then $\mathcal{P}(s, a, s') = 0$ regardless of a, except in a few special cases. According to the formalism defined above, each of the constituent task MDPs has exactly one start state and one end state. So, if s^i_{start} is the start state for Task i, and s^i_{end} is the end state for Task i, then

$$\mathcal{P}(s^i_{end}, a, s^{i+1}_{start}) = 1 \text{ for all } a \in \mathcal{A} \text{ and } 1 \leq i < T. \tag{4.4}$$

Therefore, a single episode in the combined MDP consists of one episode in each task, in sequence. The last components that needs to be defined are $\mathcal{R}$ and N. The objectives from each task remain separate in the combined MDP, so

$$N = \sum_{i=1}^{T} N_i. \tag{4.5}$$

In order to define the reward function, a vector concatenation operator needs to be defined:

$$(v_1, \ldots, v_n) \oplus (u_1, \ldots, u_m) = (v_1, \ldots, v_n, u_1, \ldots, u_m), \text{ and} \tag{4.6}$$

$$\bigoplus_{i=1}^{n} \overrightarrow{v}_i = \overrightarrow{v}_1 \oplus \cdots \oplus \overrightarrow{v}_n. \tag{4.7}$$

Given this operator, the reward function is defined as

$$\mathcal{R}(s, a) = (\bigoplus_{i=1}^{k-1} \overrightarrow{0}_{N_i}) \oplus \mathcal{R}_k(s, a) \oplus (\bigoplus_{i=k+1}^{T} \overrightarrow{0}_{N_i}) \text{ when } s \in \mathcal{S}_k. \tag{4.8}$$

In this definition, $\overrightarrow{0}_x$ represents the zero vector of length x. $\mathcal{R}$ is getting the rewards for Task k according to $\mathcal{R}_k : (\mathcal{S}_k \times \mathcal{A}_k) \rightarrow \mathbb{R}^{N_k}$, and then padding the vector with zeroes in the places for objectives from the other tasks. An interesting property resulting from this definition is that the sets of objectives used by all isolated tasks partition the set of all objectives used in the combined MDP.

The above example shows how to construct a domain with isolated tasks by combining several MDPs, but there exist MDPs with similar properties that are not constructed. In order to identify an MDP as consisting of isolated tasks, it must have the following properties identified in the example above:

1. The state space can be partitioned into separate state spaces for each task.
2. The transition function is guaranteed to take the agent from one task to the next in fixed sequence, visiting each task once with no returning to earlier tasks.
3. Events in one task do not affect which states are reachable in other tasks. Specifically, each task has only one start and end state. If a task had multiple exit states that transitioned to multiple start states in another task, but with different transition probabilities, then the agent's choices in one task could influence its starting position in the next task. Such is not the case with isolated tasks.
4. The set of objectives is partitioned according to each task.

For example, the sport of baseball is a domain that has these properties. A baseball inning is partitioned into offensive and defensive tasks. When one team is at bat (offense), the other team is on the defense. When the batting team receives three outs, they trade places with the defensive team. The actions available to players at bat does not depend on actions performed while on defense, and vice versa[2]. The game pauses and resets as teams switch roles. In the offensive task, the team's goal is to maximize its own score by earning runs, and in the defensive task the team tries to minimize the score of the opposing team, thus the objectives are partitioned.

One approach to isolated tasks is to simply learn separate controllers for each task. Such an approach makes sense if the different tasks are easily identified and can be learned in isolation. Professional baseball teams use this approach to a limited extent, because (in certain leagues) the pitcher is not required to bat. Therefore, the pitcher can focus on the pitching aspect of the defensive task, and ignore the offensive (batting) task. However, such information is not always available, and no learning system will ever be fully general if it cannot deal with separate isolated tasks. After all, the batters are still required to fill a role while the team is on the defensive, so even a player that focuses on batting will have to deal with isolated tasks to some degree.

[2]Strictly speaking, substituting players in the defensive task affects batting order in the offensive task. However, the actions available to each player are in no way affected. If the policy of the team is being learned, then the notion of substituting players becomes meaningless, because the learning agent could chose to have any player behave as it wants, thus removing distinctness from individual players.

Intelligent agents perform many different tasks, and many tasks are unrelated. Therefore, intelligent agents should be able to learn how to handle isolated tasks using the methods for learning multimodal behavior developed in this dissertation. These methods will be evaluated in two domains with isolated tasks in Chapter 6. However, multiple tasks within a domain are not always completely isolated, which makes the relationships between tasks more complicated.

4.3 Interleaved Tasks

If an agent must perform multiple tasks to succeed in a single domain, but performance in certain tasks has an impact on what happens in the other tasks, then the tasks are no longer isolated. When one task directly leads into the next task, then the tasks are interleaved (Figure 4.2).

Figure 4.2: **Interleaved Tasks.** A domain consisting of two interleaved tasks. The tasks of this domain share a common time line. Tasks switch between time steps, meaning that the current state before the switch directly affects the state and available actions at the beginning of the new task. Good behavior in a domain with interleaved tasks involves acting in a way that leaves the agent prepared for these switches.

Isolated tasks can be thought of as an extreme example of interleaving: Each task is visited once before shifting to the next. However, the definition of a domain with interleaved tasks is more permissive; some of the requirements of isolated tasks must be relaxed to arrive at the definition of interleaved tasks.

First, the requirement that the state spaces for the individual tasks create a partition of the combined state space will be retained. Every state is a member of exactly one task. However, more ways of transitioning between different tasks are now allowed. The state space $\mathcal{S}_i$ for any given task can have transitions to any other task $\mathcal{S}_j$ where $i \neq j$. However, each Task i will contain a subset of states $\mathcal{S}_i^{\text{core}}$, called the core, that are only able to transition back to that task. Formally,

$$\forall a \in \mathcal{A} \forall s \in \mathcal{S}_i^{\text{core}} \forall s' \in \mathcal{S}_j \text{ where } i \neq j, \mathcal{P}(s, a, s') = 0 \tag{4.9}$$

Because each task now potentially has multiple exit and entry points, potentially with different transition probabilities, an agent's actions in one task now have consequences in other tasks. Different entry points to a task can lead to different areas of its core, which can influence the rewards generated by $\mathcal{R}$.

The reward function in a domain with interleaved tasks is still made up of separate reward functions $\mathcal{R}_i$ for each Task i, but the sets of objectives used by each task need no longer be disjoint. In fact, each function $\mathcal{R}_i : (\mathcal{S}_i \times \mathcal{A}_i) \rightarrow \mathbb{R}^N$ is defined on all N objectives of the combined domain with interleaved tasks. Certain objectives may always have a 0 reward in certain tasks, but this is not required of the definition. $\mathcal{R}$ is now simply defined as $\mathcal{R}(s, a) = \mathcal{R}_i(s, a)$ if $s \in \mathcal{S}_i$.

The definition provided so far is very general; nearly any domain can be contorted to fit the definition of interleaved tasks. Any state space can be partitioned into arbitrarily many tasks with very small cores. For any arbitrary subset of the state space, the portion of it that happens to transition back to the same subset can be labelled the core, and the rest of the states are simply those that go to other tasks.

Therefore, an additional restriction is needed to make the definition of interleaved tasks useful. Specifically, a task needs to be a stable entity, and different tasks need to be distinguishable from each other. Note that for distinct behavioral modes to be useful, the proportion of task switches to total time spent in the domain should be small. In this case, a large majority of state transitions must be confined to one task, which in turn means that the cores of each task are relatively large. Therefore, having a low rate of thrashing back and forth between tasks is an indication that a domain consists of a few important tasks that partition the state space. Formally, a domain with interleaved tasks should have a low thrashing rate, where the thrashing rate is the average number of task transitions per episode divided by the average number of time steps per episode.

Calculating the thrashing rate with precision may not always be possible, but a rough estimate can be enough to establish suitable bounds. There is no precise boundary where a domain suddenly goes from being interleaved to not being interleaved, but the lower the thrashing rate is, the more likely a domain can be usefully labelled as having interleaved tasks. At the extreme end, a domain with isolated tasks has the lowest thrashing rate possible for a given number of tasks and a given number of time steps per episode. In this dissertation, Imprison Ms. Pac-Man (Chapter 9) serves as the prime example of a domain with interleaved tasks, and a conservative estimate indicates that it has a thrashing rate less than 0.01 (as will be explained in Section 8.3).

In summary, a domain with interleaved tasks has the following properties:

1. The state space can be partitioned into separate state spaces for each task, as with isolated tasks.

2. The transition function can go between tasks, but the majority of states are in the cores of each task.

3. Actions in one task can affect other tasks because there may be multiple ways of transitioning between tasks. The agent's actions in one task determine its entry state in other tasks.

4. The domain has a low thrashing rate.

Interleaved tasks can also be understood by looking at an example from the world of real-life sports: American football interleaves the tasks of offensive and defensive play. The offensive team tries to advance the ball toward the defensive team's end zone in order to score touchdowns. The roles of each team are well defined, but on any play there are turnover events (fumble, interception) that can result in the ball changing hands, which switches the roles of the teams. Turnovers and touchdowns do not happen on every play; the thrashing rate is low[3].

When a defensive player does intercept the ball, his team suddenly becomes the offensive team. Furthermore, the options available to all players after an interception depend directly on where the players were at the time of the interception. If the offensive team is widely dispersed when their quarterback throws a ball that gets intercepted, then there are many gaps available through which the interceptor of the ball can run. The actions of agents in one task have a direct impact on the state of the environment after the task switches, which in turn affects the actions available to all agents.

Notice that in this example, not only can the task switch in the middle of play, but the agents themselves determine when the task switches. The task switch could be caused by a bad pass from the offensive quarterback, or a skilled maneuver by a defensive player that intercepts the ball. Even if the ball is not intercepted during play, if the offensive team has nearly run out of downs, it will often choose to punt the ball to the opposition, thus putting the opponent at a disadvantage when the task switches. The ability to control when tasks switch can be very important to success in a domain, and is yet another important behavior that policies must learn in order to generate successful multimodal behavior. In fact, learning to control precisely when the task switches will prove to be very important in Imprison Ms. Pac-Man (Chapter 9).

Note also that when the task switches, say from defensive to offensive, there is no strict property of the domain's state space demanding that a previous offensive possession be included in the same task as the current offensive possession. Each time the ball changes hands, the agents could be said to be in yet another new task. Since American football games are timed, the current time is part of the state, meaning that there are no transitions from the core of one possession to the core of another possession across large gaps of time; in terms of how the state space is structured, each possession is like a separate task. Modelling the domain this way does not change the fact that each task affects the next, so the domain is still interleaved rather than isolated, but in terms of the semantics of the game, it seems strange for separate possessions to be separate tasks. One possession is semantically similar to another: The offensive team always has the ball.

Grouping all offensive actions into one task and all defensive actions into another task is a sensible division. Such a division could be used to tell a Multitask Learning agent (Section 3.2.1) how to split up a domain, or could be used to define the β function for split sensors (Section 3.1.2) that indicates when conflict sensors can be interpreted in different ways. Alternatively, preference neurons (Section 3.2.2), can learn their own task division, and could potentially favor a division

[3] A degenerate case in which the ball fumbles back and forth every few seconds without end is in theory possible, but is too unlikely to consider.

between different offensive plays, or at least different types of offensive plays (kicking, running, passing).

The concept of interleaved tasks is defined in terms of the structure of the state space, which is in turn defined by the transition function. However, the football example above shows how high-level semantic properties of a domain can be useful in grouping separate task cores into a single semantic task, e.g. an offensive task or a defensive task. It is sometimes hard to partition the state space of a domain with respect to such semantic properties. In fact, sets of states in which different semantic properties hold may not be disjoint. The region where such properties overlap blends the properties of different tasks. The challenging case of blended tasks is described next.

4.4 Blended Tasks

In the American football example above, the task switches at the point where the ball is intercepted, or otherwise changes possession. Possession of the ball is the semantic property that unifies all offensive activity in opposition to all defensive activity. However, in some domains it is not always clear what the current task is (Figure 4.3).

Figure 4.3: **Blended Tasks.** A domain with two tasks that are blended. The tasks share a common time line; at certain times it is clear what the current task is, but near the border where one task becomes another it is unclear. Is there actually a border that is simply not obvious to a human observer? Or is there a region in time during which both tasks are occurring simultaneously? Or does the area where tasks seem to overlap actually represent a third, totally different task? Domains with blended tasks are challenging because the answers to these questions are unclear.

A domain with blended tasks is a special case of a domain with interleaved tasks. To identify a domain with blended tasks, one must be able to identify important semantic properties of that task, specifically those that are most likely to result in significantly different goals. If structurally breaking up the state space into separate cores results in tasks that have multiple semantic properties associated with other tasks, then the core with multiple semantic properties represents a blended region between tasks. For example, imagine a version of football in which both teams could sometimes have a ball at the same time. This task would be different from the regular tasks of offense and defense. Each team's efforts would need to be divided between advancing their own ball down the field and preventing the opponents from advancing their ball in the other direction. The semantics of defense and offense are blended in this new task.

Formally, assume a domain has T interleaved tasks, whose state spaces $\mathcal{S}_1, \ldots, \mathcal{S}_T$ partition the global state space $\mathcal{S}$. Then let $\psi_1, \ldots, \psi_m : \mathcal{S} \rightarrow \{\top, \bot\}$ be predicates that indicate whether a state has one of m important semantic properties. Some expert knowledge is needed to identify these predicates, but useful semantic properties should be linked to the structure of the domain. Therefore, it should be true that $m \leq T$. In fact, in any domain with interleaved tasks (not just blended tasks), each task should have a semantic predicate that is true in all states of that task:

$$\forall i \in \{1, \ldots, T\} \exists j \in \{1, \ldots, m\} \forall s \in \mathcal{S}_i[\psi_j(s)]. \tag{4.10}$$

To improve readability, the predicates $\Psi_1, \ldots, \Psi_m : \{1, \ldots, T\} \rightarrow \{\top, \bot\}$ are used to indicate if a given semantic predicate is true for all states in a task, i.e.

$$\forall i \in \{1, \ldots, T\} \forall j \in \{1, \ldots, m\} (\Psi_j(i) \leftrightarrow \forall s \in \mathcal{S}_i[\psi_j(s)]). \tag{4.11}$$

This notation makes it easy to define a domain with blended tasks: It is a special type of domain with interleaved tasks in which there are tasks where two semantic predicates apply, that are also true individually in other tasks. Formally, there exist distinct values of i, j, and k in $\{1, \ldots, T\}$, and distinct values of p and q in $\{1, \ldots, m\}$, such that

$$\Psi_p(i) \wedge \Psi_q(j) \wedge \Psi_p(k) \wedge \Psi_q(k). \tag{4.12}$$

Domains with blended tasks have the above property, and Task k in particular is a specific blended task that blends the semantics of Task i and Task j. In contrast, tasks for which only one of the m semantic predicates applies are pure tasks.

The formal details above indicate that a domain with blended tasks has the following properties:

1. The domain has all the properties of domains with interleaved tasks.
2. The domain contains at least one blended task, i.e. a task in which multiple semantic predicates are true, meaning that multiple, potentially conflicting goals are available to be pursued at the same time.

This definition establishes the set of blended tasks as a strict subset of the set of interleaved tasks. However, for simplicity, whenever interleaved tasks are mentioned in this dissertation, they are assumed to be pure interleaved tasks (not blended). Whenever a domain is blended, this will be stated explicitly.

Real-world examples of domains with blended tasks include one-on-one combat, such as boxing, fencing, or martial arts. In any kind of combat, the participants must focus on attack and defense. Punches, kicks, and sword thrusts are all offensive actions, and one can defensively block punches and kicks, or parry a sword thrust. A skilled fighter is always prepared to follow up a block

with a counter-attack, and is ready to defend after launching a failed attack; this back-and-forth indicates that the roles of attacker and defender can swap very rapidly. However, sometimes these roles are not only swapping, but become blended.

For example, many actions are both offensive and defensive. A boxer may move slightly to the side to dodge an opponent's punch while simultaneously punching back. A proper fencing parry both deflects an opponent's sword (defensive), and points the defender's blade at the opponent (offensive). One may also defend against a kick by grabbing it, which is an offensive action. Many techniques have a dual role. As such, it does not make sense to say that the fighters are exclusively attacking or defending at all times.

In these domains with blended tasks, it becomes more important for the agent to decide what role to take on at any given time, and as a result, the distinction between tasks becomes even less clear. For example, if two fighters throw punches at the same time, but one is slightly faster, then that punch will land first, and perhaps disrupt the opponent's punch so that it does not land. Both fighters took on an offensive role at the same time, but one punch was canceled out because that fighter made a bad judgment. In this case, the slower fighter should have taken a more defensive role, but this conclusion is only clear in hindsight. The importance of timing the execution of offensive vs. defensive actions will be explored in the Battle Domain (described in Section 5.4).

In terms of the formal definition given earlier, the ideal goal of an agent becomes unclear when in a blended task, because multiple semantic predicates are active; multiple goals are within reach simultaneously, or at least seem to be, so whether the agent decides to pursue all goals, no goals, or any one specific goal, the consequences of actions taken in the blended task are magnified.

In summary, there are many reasons why domains with blended tasks are challenging: (1) Agents have more responsibility in determining what the current task is, (2) sometimes there are situations when multiple tasks overlap, and (3) it is not clear whether the blended situations are just a combination of tasks or new tasks in their own right. The challenges of these domains will be explored in both the Battle Domain (Chapter 7) and Ms. Pac-Man (Chapter 10).

4.5 Conclusion

This chapter formally described several types of domains in which multimodal behavior is required in order to succeed. Examples of each type of domain will appear later in this dissertation. Domains with isolated tasks are discussed in Chapter 6. A domain with blended tasks is discussed in Chapter 7. These domains are all implemented in BREVE, a simulation environment described in Chapter 5 next. Intelligent behavior is evolved in a domain with interleaved tasks in Chapter 9. Then in Chapter 10, experiments in another domain with blended tasks are conducted. These domains are both variants of the challenging classic video game Ms. Pac-Man (Chapter 8). The experiments in these challenging domains will show the benefits of the methods developed in Chapter 3 for learning multimodal behavior.

Chapter 5

Domain Description: BREVE

The BREVE simulation environment[1] by Klein (2003) is an open-source tool that supports the development of multi-agent simulations, particularly for the study of decentralized behavior and Artificial Life. Since BREVE was used to develop several of the domains in which multimodal behavior is evolved and evaluated, this chapter gives an overview of BREVE, and describes the specific domains that were created using this tool.

5.1 BREVE Basics

BREVE is a 3D simulation environment that allows the user to make complex simulations by programming the behavior of the agents in a simple interpreted scripting language called `steve`. The `steve` language is object-oriented, and each `steve` entity has a method that is called on every time step of the simulation. Therefore, the programmer can define each agent's behavior in isolation, and let the system determine the consequences of how the agents interact.

BREVE is available for Mac OS X, Linux, and Windows. Though control scripts are written in `steve` (or optionally `Python`), the code for BREVE itself is written in `C++`. OpenGL is used for visualization, but it is also possible to run BREVE in non-visual mode in order to speed up execution. BREVE supports the simulation of realistic physics, but this feature is not needed for the domains studied in this dissertation.

All of the domains developed in BREVE for this dissertation have a similar setup. They are adversarial multi-agent scenarios that start with a cooperating team of four evolved agents surrounding an opponent that uses scripted behavior to oppose them. To distinguish between the two types of agents, the evolved agents are called "monsters," and the scripted opponent is called a "bot."

Each evolved monster gets its own copy of the same evolved neural network policy to control its behavior. Therefore, the evolved team can be said to be homogeneous. Work by Waibel et al. (2009) and preliminary work done for this dissertation indicate that homogeneous teams of agents

[1] `http://www.spiderland.org/`

are easier to evolve than heterogeneous teams, i.e. those where each agent has a different policy. Proper credit assignment is difficult with heterogeneous teams, because it is not always clear to what extent the actions of a teammate led to a particular agent's success or failure. In contrast, the behavior of homogeneous agents can always be credited unambiguously to the one policy shared by all agents. Therefore, homogeneous teams can more easily evolve teamwork, which will be vital in the domains described below.

The environment used in all domains is an infinite plane in continuous space. Evaluations have a limited duration of 600 time steps per task, and are independent from each other. All agents are shaped as pentagonal-base pyramids that point in the direction they are currently facing (Figure 5.1), which matters for sensor definition. Agents can move forward and backward and can turn left and right with respect to their current heading.

Monsters start facing the bot, but the initial heading of the bot is always random, which requires the monsters to learn situational behavior and makes it detrimental to memorize bot trajectories. As a result, evaluation is noisy, and multiple evaluations are needed to get reasonable estimates of monster performance.

Monsters and the bot interact in different ways in each task, but the monsters never physically interact with each other (i.e. they can occupy the same space). Each domain is adversarial in that agents have various ways of dealing damage to each other. However, regardless of how

(a) Front Ramming Task (b) Back Ramming Task

Figure 5.1: **Front/Back Ramming Domain.** In both (a) Front Ramming and (b) Back Ramming, the monsters start pointed at the bot in the center. The rams are depicted by white orbs attached to the monsters. In the Front Ramming task, monsters can start attacking the bot immediately, but in the Back Ramming task they must turn around first. Learning which behavior to exhibit is difficult because different behavior is needed in the different tasks, even though the sensor readings are the same (monsters cannot sense where on their bodies the rams are affixed). However, by remembering the history of the bot's movements and the consequences of interacting with it, monsters can learn to exhibit the appropriate behavior in each situation.

The two tasks differ in where the ram is affixed on the body of the monsters. The two tasks are Front Ramming and Back Ramming. In Front Ramming, the rams are attached to the fronts of monsters' bodies, and an evaluation starts with rams pointed at the surrounded bot. In Back Ramming, the rams are attached to the rear ends of the monsters, so the rams start facing away from the bot. In this task, monsters must execute a 180 degree turn before being in a good position for ramming. However, in order to know which task they are facing, the monsters must be able to track how the bot behaves across time using some form of memory, such as recurrent network links.

Bot behavior is essentially the same in both tasks: It will try to circle around the rams to hit the monsters from the unprotected side if possible, but if threatened by the rams, it will prefer to run away and avoid damage.

This domain has six objectives. Each task has its own instance of the same three objectives: deal damage to the bot, avoid damage from the bot, and stay alive as long as possible. Damage dealt to the bot is shared by all monsters on the team. The damage-avoidance and staying-alive

objectives are assessed individually, and the average across team members is assigned to the team. Although damage received and time alive are both affected by taking damage, each one provides valuable feedback when the other does not: If all monsters die, then time alive indicates how long each avoided death, but if no monsters die, then damage received indicates which team is better.

Even though the monsters cannot sense where their rams are attached, they need to be alternately offensive and defensive in each task, which makes this domain very challenging. Dealing with six objectives is also a significant challenge. However, because the monsters start off surrounding the bot, optimal behavior should allow the monsters to focus mainly on offensive strategies, which is an important way in which the Front and Back Ramming tasks each individually contrast with the Battle Domain (Section 5.4). Defensive behavior is only an issue when the vulnerable sides of the agent are exposed. Ultimately the monsters are doing the same thing in Front Ramming and Back Ramming, but have to accomplish it in different ways because of how the ram is affixed.

In contrast, the next domain discussed is challenging because monsters are required to exhibit opposite behaviors in different tasks in order to succeed.

5.3 Predator/Prey

Predator/Prey is another domain with isolated tasks, but in contrast to FBR, offensive and defensive behaviors are needed in separate tasks within this domain. Monsters are either predators or prey depending on the task, and the bot takes on the opposite role (Figure 5.2). In other words, the dynamics of the environment and the behavior of the bot change depending on the task.

Predator/prey scenarios have long been of interest in Reinforcement Learning, multi-agent systems, and evolutionary computation (Tan, 1993; Stone and Veloso, 2000b; Rajagopalan et al., 2011; Yong and Miikkulainen, 2010). What distinguishes this Predator/Prey domain from others is that agents are expected to succeed as both predators *and* prey, rather than just one or the other (much like in Ms. Pac-Man: Chapter 8).

In the Predator task, monsters are predators and the bot is prey (Figure 5.2a). The bot tries to escape by moving through a gap between two monsters. When a predator monster hits the bot, the bot takes damage. All agents move at the same speed, which means the monsters must avoid crowding the bot, since hitting it can knock it so far away that it is impossible to catch. Therefore, evaluation ends prematurely if the bot is no longer surrounded.

The Prey task reverses the dynamics of the Predator task. The bot now deals damage to monsters, who are the prey (Figure 5.2b). This task is fairly simple since monsters can avoid the bot by just running away. The bot's behavior consists of moving forward towards the closest monster. Thus, PP is challenging because a single evolved controller must execute essentially opposite behaviors depending on the task.

PP has three objectives. In the Predator task, the only objective is to maximize damage dealt, which is shared across monsters as in FBR. The Prey task has two objectives: minimize damage

(a) Predator Task (b) Prey Task

Figure 5.2: **Predator/Prey Domain.** (a) The movement path of the bot in the Predator task: It tries to escape through the nearest gap between two monsters. (b) The bot path in the Prey task: It pursues the nearest monster prey in front of it. Both situations look the same to the monsters, but because the environmental dynamics and bot behavior are different, different behavior is needed to succeed.

received, and maximize time alive. As in FBR, each amount is averaged across team members to get the team score.

Both FBR and PP are examples of domains with strictly isolated tasks. Additionally, although the individual tasks in FBR leave room for offensive and defensive behaviors, the initial setup of FBR makes committing to offensive behavior in both tasks optimal. In the next domain, the Battle Domain, evolved monsters have to switch between offensive and defensive behaviors to succeed, because they do not have rams behind which they can hide. Therefore, the Battle Domain provides an example of blended tasks.

5.4 Battle Domain

In the Battle Domain (BD), monsters have to switch between offensive and defensive behavior depending on what the bot is currently doing. The bot repeatedly swings a bat that damages monsters on impact, but the monsters damage the bot when they run into it (Figure 5.3).

The behavior of the bot is simple yet challenging. The bot moves constantly forward, always swinging its bat and turning to pursue the nearest monster that it sees. The bot can only see monsters

Figure 5.3: **Battle Domain.** The bot starts an evaluation surrounded by four monster agents. It approaches the nearest monster while swinging its bat. If the bat hits a monster, the monster is damaged; if it receives enough damage, it dies. If a monster hits the bot, the bot is damaged, and any in-progress bat swing is cancelled (the bat disappears). Monsters must avoid the bat while looking for opportunities to strike the bot, but it is unclear where the boundary between these two tasks is. Therefore, the task is blended.

in front of it, so it ignores near monsters that are behind it. If no monsters are in front of the bot, it turns left until it finds one. Because the bot constantly swings its bat, a frontal attack by a monster is difficult but possible, with precise timing. The uncertainty in timing such an attack is why BD blends offensive and defensive tasks. However, teamwork can also help the monsters overcome this challenge, which means that different members of the team are taking on different roles at the same time.

The tension between attack and defense is caused by multiple contradictory objectives. These objectives are similar to those in FBR and PP. The first is to maximize damage dealt to the bot, which is shared across monsters as in FBR and PP. The other two objectives are to minimize damage received from the bat, and maximize time alive. As in FBR and PP, each of these amounts is averaged across team members to get the team score.

Because the tasks are blended, success in BD depends on knowing how offensive and how defensive to be at each point in time. Remaining exclusively defensive leads to a stalemate: The monsters run away forever, and neither the bot nor the monsters ever damage each other (since they move at the same speed). However, any defensive maneuvering short of pure retreat makes it inevitable that the monster is eventually hit by the bat, unless it acts in a defensively offensive

manner. In other words, going on the attack is sometimes the only way to avoid being hit, but it is hard to figure out where this switchover occurs.

Because there is a team of monsters, the switchover point for one monster can sometimes be determined by how distracted the bot is by another monster. One monster may find that it is ideal to go on the offensive once the bot starts approaching a different monster. As with counterattacking, the timing on this behavior is hard to define with precision, and depends on the relative positions and orientations of agents, but if done correctly it can be very effective against the bot.

The uncertainty in switching between defensive and offensive behavior is why this domain has blended tasks. As a result, it is also challenging, but agents can succeed in this domain if they learn multimodal behavior.

5.5 Conclusion

This chapter described the simulation environment BREVE, and several domains that have been implemented in it to study the evolution of multimodal behavior. The specific domains are: (1) Front/Back Ramming, which involves two isolated tasks requiring offensive behavior, (2) Predator/Prey, which also has two isolated tasks, although one requires offensive behavior and the other requires defensive behavior, and (3) Battle Domain, in which offensive and defensive tasks are blended together in a way that requires the agent to make intelligent decisions about when to switch from one to the other. Experiments in each of these domains are conducted over the next two chapters.

Chapter 6

Evaluation: Network Modules in Isolated Tasks

This chapter describes experiments carried out in the Front/Back Ramming (FBR) and Predator/Prey (PP) domains. These domains are both made of two isolated tasks. The purpose of these experiments is to demonstrate the benefits of evolving neural networks with multiple modules. This chapter starts by explaining the experimental setup used in both domains, then continues with the results.

6.1 Experimental Setup

In order to discover multimodal behavior, constructive neuroevolution is applied, as described in Section 2.2.1. These experiments use a feature selective approach, so networks are initialized with only one synapse per output neuron (Whiteson et al., 2005). As also explained in Section 2.2.1, crossover is not used in conjunction with feature selective networks. Whenever new child networks are produced, there is a 40% chance a single weight will be perturbed, a 20% chance a single link will be added, and a 10% chance a new neuron will be spliced along an existing link.

In order to show the benefits of modular networks, several network architectures are evaluated. `Control` represents networks with one module used in both tasks of each domain. `Multitask` represents networks with one module for each task in a domain. These networks always know which task they are facing, and use the appropriate module accordingly. Two Module Mutation methods, `MM(P)` and `MM(R)`, are evaluated as well. These methods have initial populations containing networks with only one module. New modules can be added by the appropriate Module Mutation, which occurs with a 10% chance for each child network produced. Additionally, `MM(R)` uses a mutation to delete the least used output module 10% of the time. In the fifth method, `Multinetwork`, networks are evolved for each task individually. Then the resulting controllers are combined so that the appropriate network is used in the task for which it was evolved. Each run of a `Multinetwork` population actually consists of a pair of runs, one in each of the two isolated

tasks in the domain. The scores from each individual in the Pareto front of a given single-task run are combined with scores from each individual in the Pareto front of the other task from the corresponding paired run. The result is a population of scores for the full domain, representing the result of a single `Multinetwork` run.

Neural network controllers for the monsters were evolved 20 times for 500 generations for each method in each domain, using NSGA-II with a population size of $\mu = \lambda = 52$. Each controller earned scores by being evaluated in either FBR or PP. For each task within these domains, a network was copied into each of the four members of a team of monsters, as described in Section 5.1.

To deal with noisy evaluations, averaging fitness scores across multiple evaluations is a common approach (van Hoorn et al., 2009; Bryant and Miikkulainen, 2006; Cardamone et al., 2009). Therefore, every network was evaluated three times in each task, and the final score in each objective was obtained by averaging the individual scores.

On each time step of the simulation, the bot acts according to scripted behavior for the task, and the evolving agents act according to their neural networks. The monsters' sensors provide inputs to the network, which then processes them to produce outputs, which then define the behavior of the monster for the given time step.

The inputs to the monsters' neural networks are described in several tables: binary sensors in Table 6.1, orientation sensors in Table 6.2, and ray-trace sensors in Table 6.3. Though each team member is controlled by a copy of the same network, each member senses the environment differently, and can therefore take action in accordance with its particular circumstances. Additionally, each monster's network has its own recurrent state dependent on how the evolved network's recurrent links are structured, and what information they have transmitted from the monster's history of senses and actions. The recurrent states of all monsters are reset whenever the bot respawns. Even though there are many network inputs, recall that a feature-selective approach (Whiteson et al., 2005) allows for some of these inputs to be ignored or incorporated later if necessary.

In contrast to the long list of inputs, the list of outputs (per module for multimodal approaches) is short: one output for the degree of backward vs. forward thrust (negative for backward, positive for forward), and another for left vs. right turn (negative for left, positive for right). However, complex behaviors can be produced from these outputs, as the results show. Interpreting these results requires knowledge of how to assess performance in multiobjective domains, which will be discussed next.

6.2 Assessing Multiobjective Performance

The hypervolume and epsilon indicator metrics defined in Section 2.1.3 are used to assess performance in these domains with isolated tasks.

When analyzing how hypervolume changes across generations, the reference points used for each domain were their corresponding zero vectors, containing the minimum scores for each ob-

Name	Description
Bias	Constant of 1
Monster Dealt Damage	1 if monster dealt damage to bot on previous time step, 0 otherwise.
Monster Received Damage	1 if monster received damage from bot on previous time step, 0 otherwise.
Any Monster Dealt Damage	1 if *any* monster dealt damage to bot on previous time step, 0 otherwise.
Any Monster Received Damage	1 if *any* monster received damage from bot on previous time step, 0 otherwise.
Bot Knock-back	1 if bot is temporarily invulnerable because it is being knocked back, 0 otherwise.
In Front of Bot	1 if magnitude of shortest turn the bot would need to make to face the monster is less than or equal to $\pi/2$ radians, 0 otherwise.
Monster 0 Dealt Damage	1 if monster that starts evaluation north of the bot dealt damage to bot on previous time step, 0 otherwise.
Monster 1 Dealt Damage	1 if monster that starts evaluation east of the bot dealt damage to bot on previous time step, 0 otherwise.
Monster 2 Dealt Damage	1 if monster that starts evaluation south of the bot dealt damage to bot on previous time step, 0 otherwise.
Monster 3 Dealt Damage	1 if monster that starts evaluation west of the bot dealt damage to bot on previous time step, 0 otherwise.

Table 6.1: **Description of Binary Input Sensors for Monsters.** Each row is a monster sensor that reads 0 or 1 based on whether or not some aspect of the environment is in one state or another.

jective: $(0, 0, -50, -50, 0, 0)$ for FBR and $(0, -50, 0)$ for PP, where the zeroes are for the various damage-dealt and time-alive objectives, and each -50 is for one of the damage-received objectives. The normalization used for calculating hypervolumes was on a scale between the minimum points above, and maximum points based on maximum scores achieved in each objective across all experiments in a given domain. The maximum points turned out to be $(310, 210, 0, 0, 600, 600)$ for FBR and $(250, 0, 600)$ for PP. The maximum point for FBR indicates that the best damage scores in Front and Back Ramming were 310 and 210, respectively. Monster teams in each of these tasks also managed to survive the entire 600 time steps sustaining no damage. The maximum point for PP indicates that the best damage score in the Predator task was 250, and monster teams survived the full 600 time steps of the Prey task sustaining no damage. The normalization scheme ranges all the way from minimum to maximum because hypervolume scores are presented from generation 0, when scores are very small, all the way to generation 500, where the maximums occur.

Although hypervolumes are calculated for each generation, epsilon indicator values are only calculated for the final generation of each run because of the added complication of computing a reference set (Section 2.1.3). However, since final performance is all that matters, it is appropriate to focus on the final generation, especially since hypervolume values are calculated at every generation

Name	#	Description
Monster/Bot Heading Diff.	1	Shortest amount in radians that the monster would have to turn to have the same heading as the bot.
Monster Heading/Bot Loc. Diff.	1	Shortest amount in radians that the monster would have to turn to be directly facing the bot.
Monster/Teammate Heading Diff.	4	For each slot x within the team of monsters, this sensor returns the shortest amount the sensing monster would have to turn to have the same heading as teammate x. The difference in heading from a monster to itself is always 0.
Monster Heading/Teammate Loc. Diff.	4	For each slot x within the team of monsters, this sensor returns the shortest amount in radians that the sensing monster would have to turn to be directly facing teammate x. The value 0 is returned by the sensor corresponding to the sensing monster.

Table 6.2: **Description of Orientation Input Sensors for Monsters.** Each row stands for a different sensor or group of sensors, with number indicated by the "#" column. These sensors all deal with the orientation of certain domain entities relative to other entities. Therefore, all of these sensors measure radians in the range $(-\pi, \pi]$. A negative turn value corresponds to a left turn and a positive turn value corresponds to a right turn. Some groups of sensors refer to team member slots. These groups consist of four sensors each, where each sensor corresponds to a given monster, determined by its *starting* position with respect to the bot (north, south, east or west). Given monster x, its sensors that correspond to team slot x will refer to itself. Those same sensors in a different monster y will refer to monster x as well. Note that the values for the "Monster/Teammate Heading Diff." and "Monster Heading/Teammate Loc. Diff." sensors will be different for monsters x and y ($x \neq y$), because the values depend on relative monster positions and headings.

Name	Description
Bot Ray Traces	Each sensor returns a 1 if it is currently intersecting space occupied by the bot, and 0 otherwise.
Teammate Ray Traces	Each sensor returns a 1 if it is currently intersecting space occupied by any teammate, and 0 otherwise.

Table 6.3: **Description of Ray Trace Input Sensors for Monsters.** Each row stands for a group of five sensors based on ray traces. Each group consists of an array of five rays that are 3.5 times the length of an agent, and positioned around the monster relative to its heading at the angles of $-\pi/4$, $-\pi/8$, 0, $\pi/8$ and $\pi/4$ radians. The value reported by each ray depends on whether it is intersecting an agent in the environment. When combined with the sensors in Tables 6.1 and 6.2, these inputs are sufficient for the evolving monsters to develop complex and interesting behavior in the domains of this paper.

and already give insight into how multiobjective performance changes over time.

Objective scores also need to be normalized in order for epsilon indicator scores to be calculated, but since these values are only calculated for the final generation, a different normalization scheme is used. By the final generation, most objective scores are confined to smaller ranges, thus allowing normalization to focus on more relevant areas of objective space. The maximum scores used for normalization are the same, but the minimum scores, specifically the minimums in each objective across all final populations of each method, are sometimes higher. In particular, $(10, 10, -50, -50, 503.25, 264.5)$ is the minimum point for FBR, indicating that the minimum damage dealt in both ramming tasks was 10, but the tasks are different in that even the individual that died the quickest had a time-alive score of 503.25 in Front Ramming, whereas the shortest-lived individual in Back Ramming had a low time-alive score of 264.5. For PP, the minimum point is $(0, -50, 315.333333)$, and the lowest time alive score is now 315.333333. However, damage dealt and received are still minimal.

The details above indicate how multiobjective performance metrics will be applied in FBR and PP. However, there are other useful ways in which to assess performance in domains with multiple, isolated tasks, as will be described next.

6.3 Measuring Performance Across Isolated Tasks

Extra care must be taken to characterize performance properly in domains with multiple isolated tasks. In these domains, monsters that do $each$ task well are desired. However, a Pareto-based approach allows extreme trade-offs where performance is excellent in one task, but terrible in another. Because tasks are isolated, there are no inherent trade-offs between objectives from separate tasks. When such trade-offs are observed in the evolved agents, they are entirely based on differences in

the policy representation and learning method.

One way to detect whether a population performs well in both tasks is to calculate performance metrics with respect to approximation sets for each individual task, and compare these results to those for the full domain. If one method is superior to another in the combined domain, but equal in the component tasks, then its individuals score well in $both$ tasks instead of just one.

The emphasis on good performance across all tasks can be extended into an emphasis on good performance across all objectives. This focus does not mean abandoning the ability of multiobjective optimization to capture diverse trade-offs, but because this chapter is concerned with intelligent monster behavior, it is at least possible to say that a monster is only successful if it surpasses certain minimum expectations, i.e. obtains adequate goal scores in $each$ objective. Once goals are chosen, the number of individuals in a population that surpass all goals can be counted, thus giving an idea of whether the population tends to contain individuals that do well across many objectives as opposed to just a few, i.e. just the objectives for one task.

Of course, picking specific goal values requires expert domain knowledge, but since the purpose here is to assess performance, a range of goal values is used. Since all scores are normalized, any value x in the range $[0, 1]$ can be picked to define goals by translating the chosen x back into the appropriate range for each objective. For example, for $x = 0.5$ in FBR, the goals would be $(155, 105, -25, -25, 300, 300)$, since these values are halfway between the minimum and maximum scores in FBR (Section 6.2). As x increases, the number of successful individuals will drop, but the decline will be slower in populations that do well in all objectives across multiple tasks. A plot of the number of successful individuals in a population vs. x is a "Success Plot." This method of analyzing multiobjective performance is an original contribution of this dissertation.

Note that this performance metric cannot be calculated for the `Multinetwork` approach because each `Multinetwork` run consists of two runs with different sets of objectives. Because of how runs are combined, `Multinetwork` populations effectively have a size of $52 \times 52 = 2704$, and the relative significance of the $number$ of successful individuals in such a population would be difficult to interpret when compared to the other methods. Therefore, no `Multinetwork` results are shown on the success plots.

Armed with these means of performance assessment, the results for each domain can now be discussed.

6.4 Results

The results for FBR and PP are presented in this section in terms of the metrics described above. FBR is described first because its results are more straightforward, followed by PP, for which the results are more surprising.

6.4.1 Front/Back Ramming Results

The results for FBR conform to expectations of how the different methods should perform: `Control` performed the worst, both `MM(P)` and `MM(R)` are better, `Multitask` is better still, with `Multinetwork` emerging as the best. The hypervolume learning curves (Figure 6.1) indicate that this ordering is established early and maintained throughout evolution.

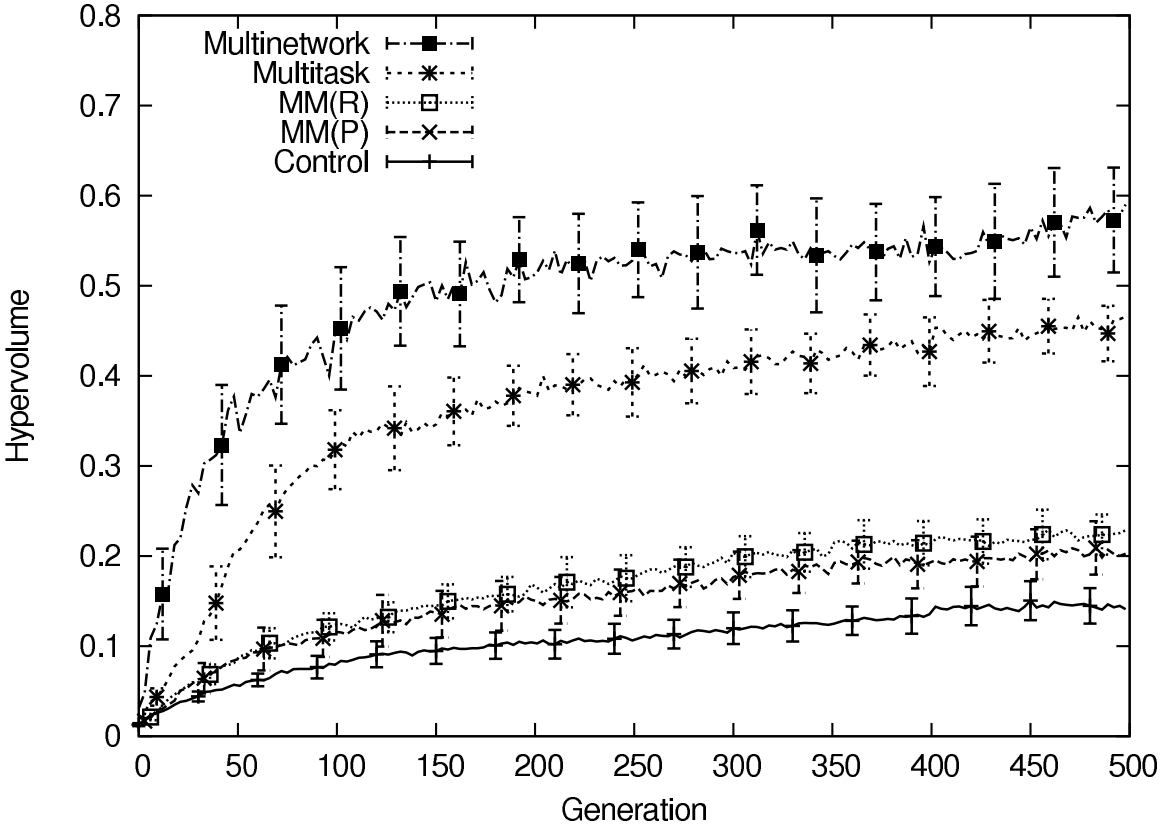
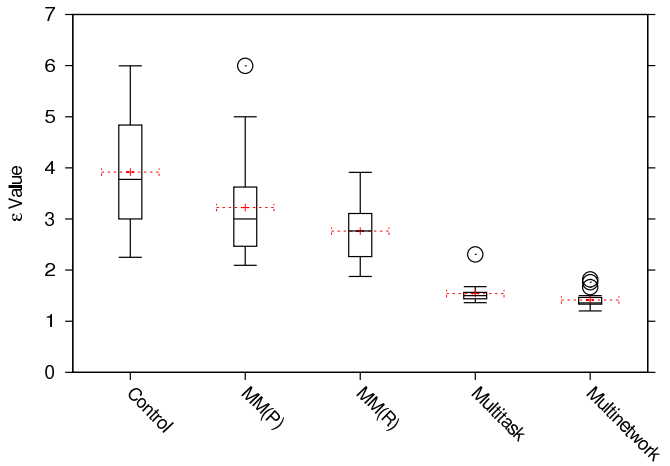
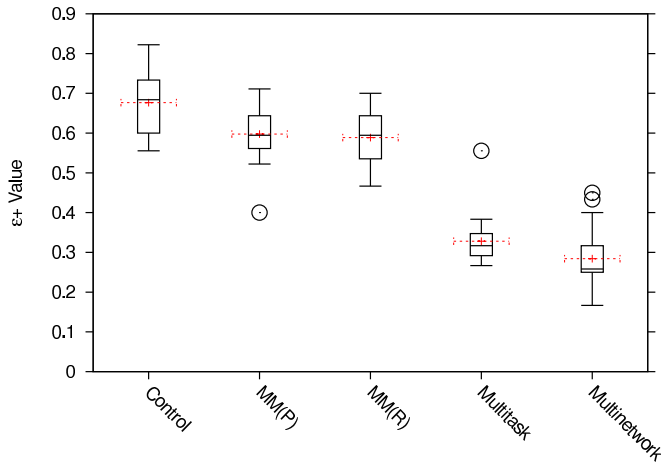


Figure 6.1: **Average Hypervolumes in the Front/Back Ramming Domain.** For each method, average normalized hypervolumes across 20 runs are shown by generation with 95% confidence intervals. The figure indicates that multimodal approaches are superior to the single-module approach, represented by `Control`. Both `MM(P)` and `MM(R)`, which have no knowledge which task they are currently facing, significantly outperform `Control`. In turn, `Multitask` has significantly higher hypervolumes than either Module Mutation method. As expected, the best performance is achieved by the `Multinetwork` approach, which is significantly better than even `Multitask`. This progression indicates that increased ability to tackle this domain as a pair of independent tasks leads to better performance.

When considering the final generation only, epsilon indicator values corroborate these results (Figure 6.2). Furthermore, Mann-Whitney U tests confirm that there are significant differences



(a) I_ϵ^1 Values.



(b) $I_{\epsilon+}^1$ Values.

Figure 6.2: **Epsilon Indicator Values in the Final Generation of Front/Back Ramming.** Given the final Pareto fronts for each of the 20 runs with each method, the (a) I_ϵ^1 values and (b) $I_{\epsilon+}^1$ values are calculated and shown as box-and-whisker plots (depicting the minimum, lower quartile, median, upper quartile and maximum scores with scores more than $1.5IQR$, or inter-quartile range, from the nearest quartile shown as outliers). Additionally, the dashed line intersecting each box is the average score in the metric. The decreasing epsilon values indicate that `Control` results in the worst Pareto fronts, both types of Module Mutation are better than `Control` and roughly equivalent to each other, `Multitask` is the next best, and `Multinetwork` is the best of all, confirming the hypervolume results from Figure 6.1

in the final generation between adjacent methods in order of increasing performance: `Control`, `MM(P)`/`MM(R)`, `Multitask`, and then `Multinetwork`. The two Module Mutation methods are lumped together because there is no significant difference between them (Table 6.4).

Comparison	HV	I_ϵ^1	$I_{\epsilon+}^1$
`Control` vs. `MM(P)`	**88**	**87.5**	127.5
`Control` vs. `MM(R)`	**36**	**81.5**	**76**
`MM(P)` vs. `MM(R)`	148	183.5	150.5
`MM(P)` vs. `Multitask`	**2**	**4.5**	**3**
`MM(R)` vs. `Multitask`	**4**	**7**	**6**
`Multitask` vs. `Multinetwork`	**59**	*111*	**93**

Table 6.4: **Two-tailed Mann-Whitney U Test Values for the Final Generation of Front/Back Ramming.** A difference between two methods is significant with $p < 0.05$ if $U < 127$ (*italic*), and with $p < 0.01$ if $U < 105$ (**bold**). All but the comparisons between `MM(P)` and `MM(R)`, and the $I_{\epsilon+}^1$ comparison between `Control` and `MM(P)` are significantly different, and of these all but the I_ϵ^1 comparison between `Multitask` and `Multinetwork` are different at the $p < 0.01$ level.

If Pareto fronts are recalculated for the constituent tasks, the hypervolumes are similar for `Control`, `MM(P)` and `MM(R)`, but significantly different for `Multitask` and `Multinetwork` (Figure 6.3, Table 6.5). Therefore, the better overall performance in FBR of `Multitask` and `Multinetwork` is due to their exceptionally good performance in both tasks. In general, individuals in the final populations of each method can do at least one task well, but the better methods have individuals that do both tasks well.

This point is seen in the average success counts as well (Figure 6.4), in which the multimodal methods, especially `Multitask`, are better than `Control` because they perform well across all objectives rather than focusing on extreme regions of the trade-off surface.

The results so far describe how the different methods perform compared to each other, but the metrics used to measure performance are somewhat removed from the actual scores achieved by agents in the domain. Presenting such data is difficult because FBR has six objectives, and the trade-offs between objectives make it impossible to identify any one best agent for any run. However, most individuals in any Pareto front for FBR earn the maximum score of 600 in both time-alive objectives, so this objective can be mostly ignored. The remaining four objectives can be split up by task, resulting in the two 2D plots of Figure 6.5, which shows the best trade-offs achieved across all runs of each method. However, this figure does not indicate when one evolved network did well in both tasks or just one.

Success plots, introduced in Section 6.3, can be used to assess what kind of scores are produced by individuals that do well in many objectives. The most successful individual of each method is defined as the one that passes the highest success thresholds in each objective, and therefore in

Comparison	Front	Back
`Control` vs. `MM(P)`	171	184
`Control` vs. `MM(R)`	176	190
`Control` vs. `Multitask`	**42**	134
`Control` vs. `Multinetwork`	**18**	**65**
`MM(P)` vs. `Multitask`	**63**	146.5
`MM(R)` vs. `Multitask`	**46**	130
`MM(P)` vs. `Multinetwork`	**35**	**88**
`MM(R)` vs. `Multinetwork`	**24**	**71**
`Multitask` vs. `Multinetwork`	132	**71**

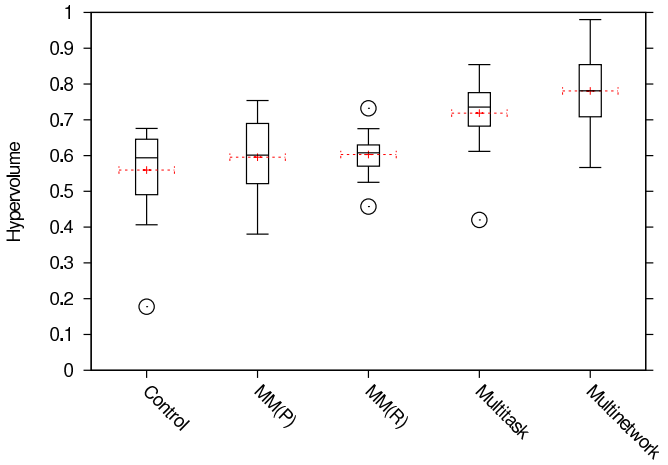
Table 6.5: **Two-tailed Mann-Whitney U Test Values Comparing Hypervolumes for the Isolated Front and Back Ramming Tasks, i.e. Ignoring Objectives From the Other Task.** `Multitask` and `Multinetwork` are significantly different from the other methods in Front Ramming, but not different from each other. However, in Back Ramming, `Multinetwork` is significantly better than all other methods, including `Multitask`. There are no other significant differences between methods in either task.

both tasks. The scores of these individuals are shown in Table 6.6.

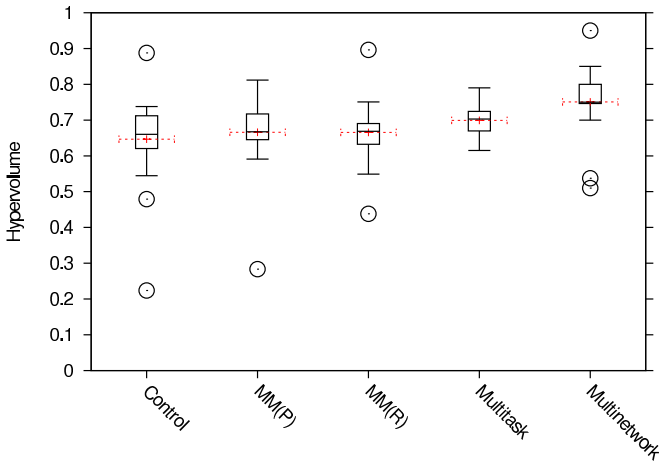
Because feature-selective evolution was used, it is interesting to analyze which inputs were used most often. The selections turned out to vary widely across runs, but the "Monster/Bot Heading Diff." and the "Monster Heading/Bot Loc. Diff." sensors were commonly included. Monster teams also tended to have sensors for the "Monster Heading/Teammate Loc. Diff." and "Teammate Dealt Damage" of at least one teammate, though the exact teammates varied across runs. Use of "Bot Ray Traces" and "Teammate Ray Traces" near the front of the rammers was also common, with one exception: Component networks evolved for `Multinetwork` in the Back Ramming task mostly ignored all ray trace sensors, which makes sense because these sensors are worthless when moving rear-first. However, `Multinetwork` teams did use ray traces in the Front Ramming task. Perhaps the ray trace sensors are actually distracting in the Back Ramming task, which may explain why `Multitask`, which had to use the same set of inputs in both tasks, performed worse than `Multinetwork` in Back Ramming, but was just as good at Front Ramming.

The behaviors of monsters with each method are in line with these results (animations can be seen at `http://nn.cs.utexas.edu/?multitask`). In general, `Control` networks easily learned to perform one of the two tasks well, but rarely both. These networks often perform the same behavior in both tasks, even when the behavior is only successful in one of the two tasks, and detrimental in the other.

In contrast, `Multitask` networks are almost always capable of performing both tasks well, as exhibited by the behavior depicted in Figure 6.6. The specific scores achieved by the team in this figure are $(220, 120, -17.5, 0, 600, 600)$. Such behaviors are easy for `Multitask` to learn since



(a) Hypervolumes for Front Ramming.



(b) Hypervolumes for Back Ramming.

Figure 6.3: **Hypervolumes for the Individual Tasks of Front/Back Ramming.** Scores corresponding to each domain of FBR are isolated from the final Pareto fronts, and used to calculate new Pareto fronts and their corresponding hypervolumes with respect to the individual tasks that make up FBR. (a) In Front Ramming, there is little difference between `Control`, `MM(P)`, and `MM(R)`; (b) In Back Ramming, these methods are also similar to each other, and to `Multitask`. However, both `Multitask` and `Multinetwork` have better hypervolumes in Front Ramming, and `Multinetwork` performs better in Back Ramming as well. The fact that both Module Mutation methods have hypervolumes similar to `Control` in the individual tasks, but better hypervolumes when tasks are combined in FBR, indicates that their good performance comes from individuals performing well in both tasks.

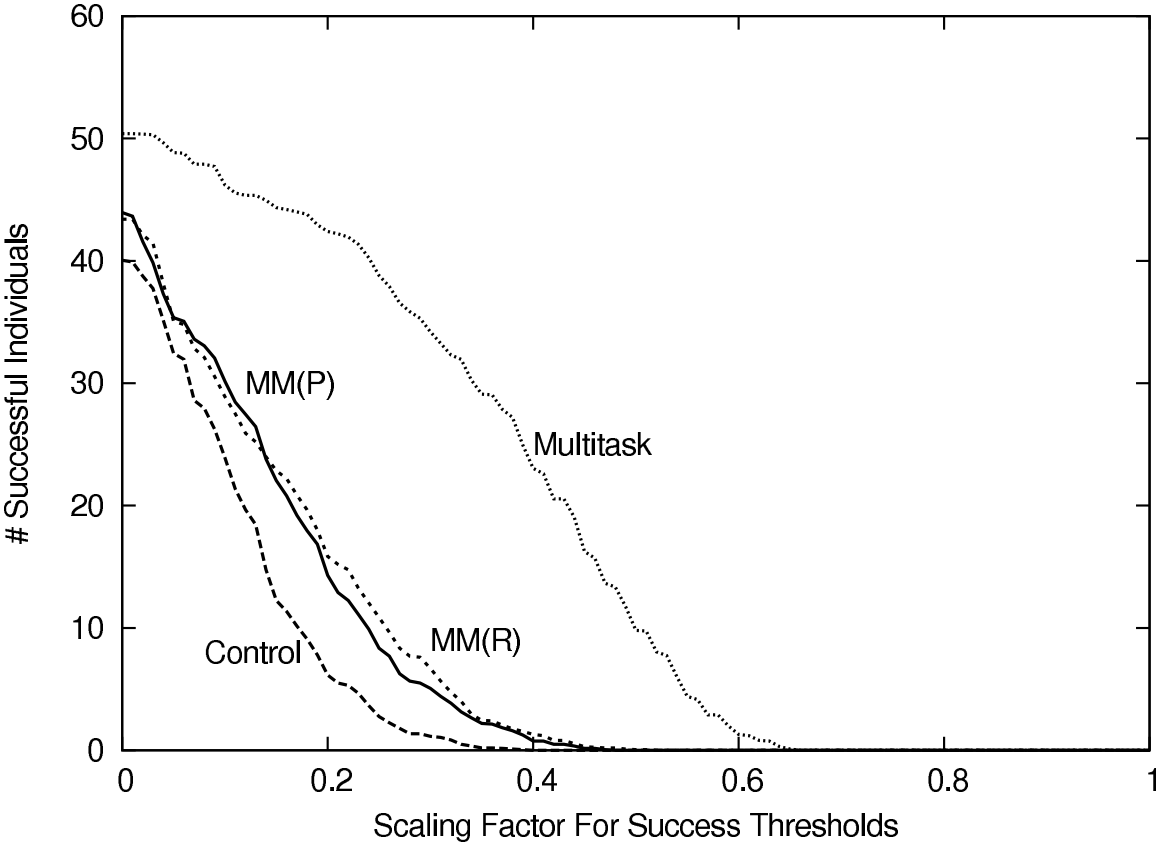


Figure 6.4: **Average Success Counts in Front/Back Ramming.** This plot shows the average number of individuals across the final size 52 populations of each single-network method that are considered successful, in that their objective scores pass a threshold for all objectives, indicating the ability to perform well across them all. The x-axis corresponds to different thresholds, i.e. the value of all normalized objective scores that must be surpassed. More `Multitask` individuals remain successful for larger success thresholds, because they perform well in multiple objectives. Similarly, the `MM(P)` and `MM(R)` curves dominate the `Control` curve. Furthermore, the two Module Mutation curves intersect each other, emphasizing that they are not very different in FBR.

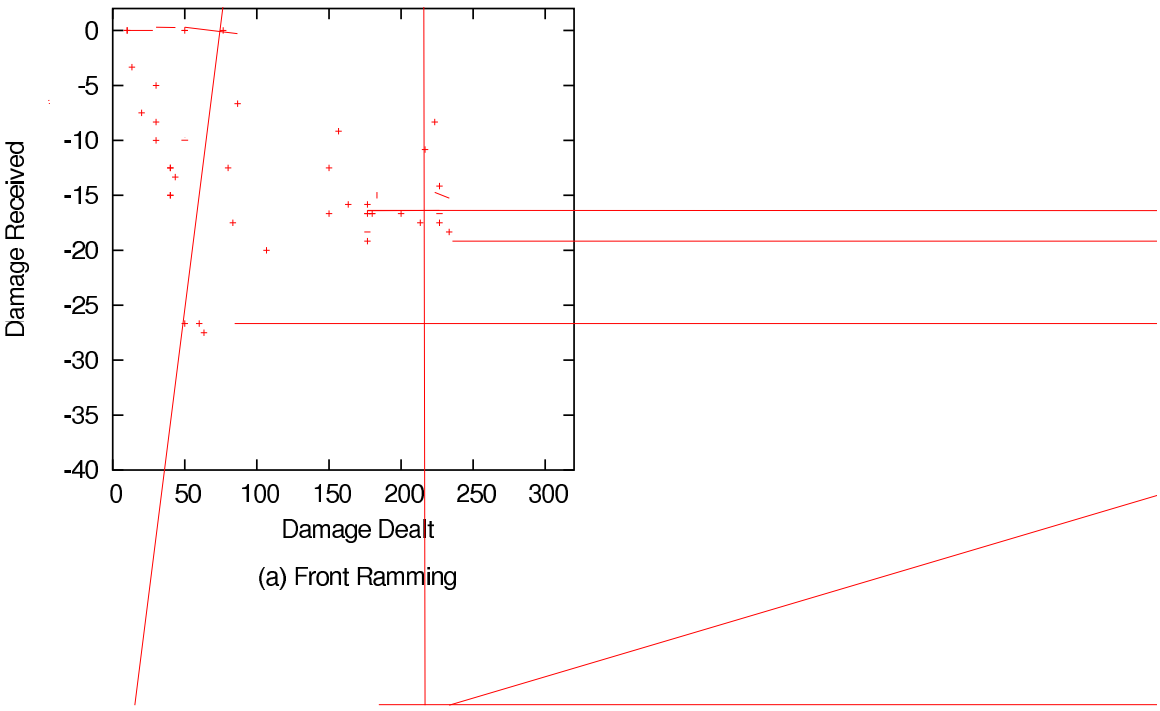


Figure 6.5: **Slices of Super Pareto Fronts From Front/Back Ramming.** (a) Damage received vs. damage dealt in Front Ramming, and (b) damage received vs. damage dealt in Back Ramming are shown for all members of each super Pareto front across 20 runs of each method. These plots illustrate the non-normalized domain performance of each method. Notice that a two-dimensional slice of a six-dimensional Pareto front will not necessarily be a Pareto front in terms of the two objectives under consideration. There are points that are dominated within the limited context of the objectives being plotted. In particular, methods that are unaware of which task they are performing (`Control`, `MM(P)`, and `MM(R)`) often have more points with low scores in one of the tasks. It is only possible for these low scoring points to be in the Pareto front for the full task if the points with low scores in one task correspond to points with high scores in the other task. In contrast, `Multitask` and especially `Multinetwork` approaches only have high scoring points in each task. These methods know which task they are facing, and can tailor their behavior to that task more easily.

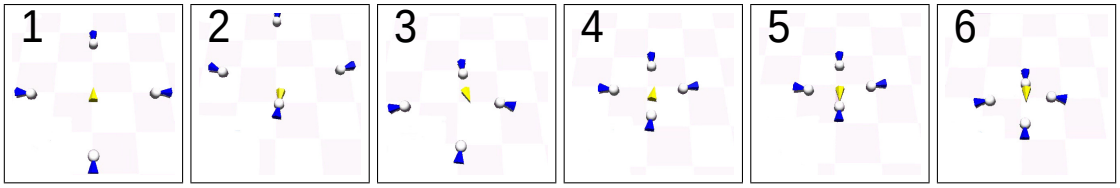
Method	Threshold	Objective	Scores
Control	0.38	0	$(116.67, 133.33, -15, -7.5, 600, 600)$
MM(P)	0.47	0	$(146.67, 113.33, -2.5, -10, 600, 600)$
MM(R)	0.51	0	$(156.67, 123.33, -10, -11.67, 600, 600)$
Multitask	0.62	0	$(193.33, 140, -15.83, -5.83, 600, 600)$
Multinetwork	0.81	1	$(280, 170, -5, 0, 600, 600)$

Table 6.6: **The Most Successful Individual of Each Method in Front/Back Ramming.** The scores for the most successful individual of each method pass the highest success threshold, which directly maps to goal values for each objective based on the minimum and maximum scores in FBR. Even though success plots cannot be generated for Multinetwork runs, its most successful individual can be determined by assessing each Multinetwork individual in terms of the goal thresholds it surpasses. The threshold reached by each set of scores is shown, as well as the objective that determines what this threshold is; an individual's threshold is its lowest threshold across scores in all objectives. The damage dealt in Front Ramming determines the success threshold for all methods except Multinetwork, whose least-successful objective score is damage dealt in Back Ramming. The damage dealt objectives are the most challenging because they have no hard ceilings like the damage-received and time-alive objectives. Notice that the ordering of the success thresholds for the most successful individuals of each method corresponds to the same performance ordering established by all previous metrics.

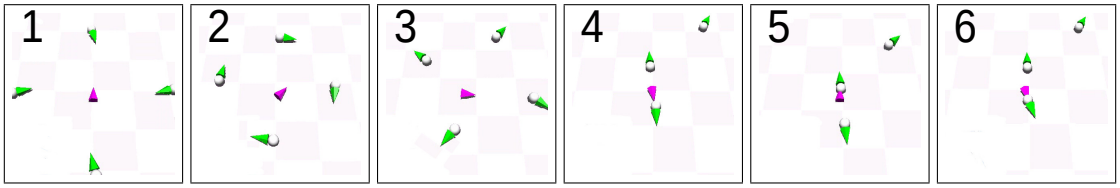
the networks have different policies for each task. In the Front Ramming task monsters rush forward to ram the bot, and in Back Ramming the same monsters immediately turn around at the start of the trial so that they can attack the bot with the rams on their rears. Multinetwork teams behave similarly, but are even more efficient at Back Ramming, presumably because their behavior for that task is optimized in isolation from Front Ramming.

Module Mutation networks, though lacking information available to Multitask and Multinetwork, are significantly different from Control networks in an important way: They are capable of solving both tasks instead of just one. However, since Module Mutation networks need to overcome the challenge of not knowing which task they are facing, their scores tend to be lower than those of Multitask networks.

For example, an MM(R) network with scores of $(140, 170, -12.5, -7.5, 600, 600)$ exhibited the interesting behavior depicted in Figure 6.7, which uses recurrency to remember the current task and manage module usage. The network had nine modules. Their usage profile in the Front Ramming task was 30.26%, 34.82%, 34.51%, 0%, 0.40%, 0%, 0%, 0%, 0%; and their usage in the Back Ramming task was 59.33%, 34.12%, 6.25%, 0.04%, 0.21%, 0%, 0.04%, 0%, 0%. Each percentage represents how much a particular module was used by the monster team during evaluation in a particular task. Therefore, three of the nine modules were not used at all, two were not used in Front Ramming and only used sparingly in Back Ramming, and another module was only used

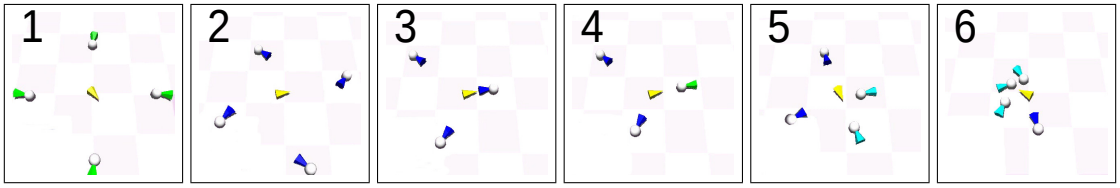


(a) Front Ramming

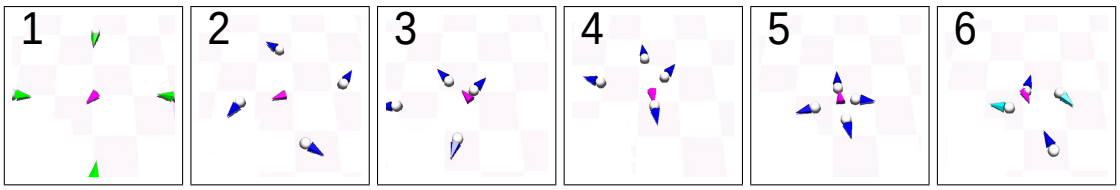


(b) Back Ramming

Figure 6.6: **Illustration of Intelligent Behavior Learned by `Multitask` Networks in Front/Back Ramming.** Animations of these and other behaviors can be seen at `http://nn.cs.utexas.edu/?multitask`. Each row shows snapshots from the evaluation of an agent over time from left to right. (a) Behavior in the Front Ramming task is shown first, and in (b) behavior of the same agent in the Back Ramming task is shown. The monster behavior is distinctive in each task, since different output modules are dedicated to each one. `Multitask` networks immediately take advantage of their knowledge of the current task: In Front Ramming they attack immediately (Frame 2), and in Back Ramming they turn around immediately (Frame 2) and then start attacking. No time is wasted figuring out what task is being faced, as Module Mutation networks must do (Figure 6.7).



(a) Front Ramming



(b) Back Ramming

Figure 6.7: **Illustration of Intelligent Behavior Learned by Module Mutation (`MM(R)`) in Front/Back Ramming.** Notice that in the first two frames of both (a) Front Ramming and (b) Back Ramming, the evolved monsters perform the same maneuver, since they do not know yet which task they are in. They start by turning their backs towards the bot. In the Back Ramming task, this strategy is immediately effective, as illustrated by the monsters confining the bot while knocking it around with their rams (Frames 3–6). In the Front Ramming task, this behavior causes the monsters to be hit (Frame 3), but this hit does two things: (1) the attacking monster is flung backwards with its front ram facing the bot (Frame 4), and (2) monsters sense being hit, and as a result switch network modules so they now attack with their front rams (Frames 5–6). Preference for the new attack module is then maintained by internal recurrent state. This multimodal behavior is a good example of how Module Mutation networks can learn to overcome the challenges of a domain with multiple isolated tasks with a combination of recurrent connections and structural modularity.

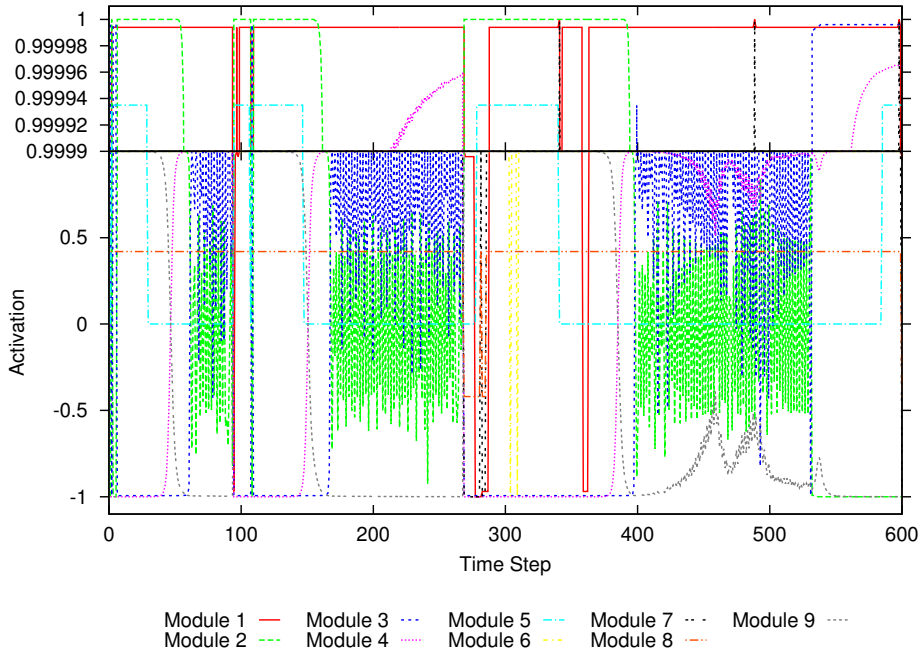


Figure 6.8: **Preference Neuron Activations of `MM(R)` Network in Front Ramming.** These activations correspond to a single member of the team whose behavior is illustrated in Figure 6.7. At any given time step, the module whose preference neuron has the highest activation is chosen. Neuron activations are restricted to the range $[-1, 1]$ from using the `tanh` activation function, but in order to better tell which module has the highest activation on each time step, the range from 0.9999 and up is magnified in the figure. Module 1 was used the most because it almost always maintains high activation. This module was used when the monster was attacking the bot, after it realized which task it faced. Module 2 has activations even higher than Module 1 for three prolonged periods. Each of these periods begins immediately after the bot spawns. This module is the 2nd most used because whenever the bot respawns, the recurrent states of monsters are flushed, so that the monsters lose the knowledge they had previously gained from interacting with the bot; Module 2 gets used after each bot respawn until the monster is reminded that it is in the Front Ramming task. Module 3 is used the 3rd most because it controls the monster for a prolonged period near the end of the evaluation. This unusual usage occurs because the monster team's plan goes awry after the last respawn, which is why the monsters come up short of killing the bot a third time. Module 3 gets used because the monster is trying to escape taking damage; a situation it does not have to deal with during the first two successful bot spawns. The other modules were either not used at all, or used only for single isolated time steps at points when their activation spiked. For example, see how Module 7 spikes around generations 340, 490 and 600. This figure shows how `MM(R)` can evolve a network that makes use of multiple modules to exhibit separate modes of behavior in a domain with isolated tasks.

sparingly in both tasks. The remaining three modules were primarily in charge of controlling the team. In particular, monsters made use of Module 1 more often in Back Ramming than in Front Ramming, and less use of Module 3 in Back Ramming than in Front Ramming. Module 2 is used equally often in both tasks.

For a single monster in the team generated from this network, Figure 6.8 shows how the activation of each preference neuron fluctuates during a single Front Ramming evaluation. Some modules mostly maintain constant activation while others exhibit wild thrashing behavior, in essence merging two modules. Others behave like digital signal waves, and some gradually rise and fall like sine waves. The modules that are most often chosen by this network tend to control the monster for prolonged periods, making it easy for a human observer to associate particular modules with particular behaviors. Other networks (not shown) sometimes utilize the other types of modules instead. `MM(R)` thus uses its modules in a variety of ways to help establish complex behavior.

Behaviors similar to those exhibited by `MM(R)` were also learned by `MM(P)`, but `MM(P)` networks often had many unused modules, because they could not delete modules. For example, an `MM(P)` network with scores of $(117.78, 152.22, -7.22, -8.61, 600, 600)$ had 22 modules, but only five were used. The usage profile of these five modules in the Front Ramming task was 54.60%, 3.22%, 0%, 0%, 42.19%; and their usage in Back Ramming was 40.19%, 1.66%, 0.47%, 0.34%, 57.34%. So out of 22 modules, only five were used, and in Front Ramming only three of those modules were used. Even in Back Ramming, these two extra modules are used sparingly. In each task, monster teams are controlled a majority of the time by the same two modules. However, these two modules combined with minor contributions from the remaining three to define an effective multimodal strategy for the monsters.

Though in terms of performance there is no significant difference between `MM(P)` and `MM(R)`, `MM(R)` networks tend to associate particular modules more clearly with particular behaviors. That is, `MM(R)` behaviors are more transparent, whereas `MM(P)` networks frequently thrash between modules, or exhibit multiple behaviors in a single module. Most likely the reason is that `MM(P)` modules are more interconnected. Since each module leads into the next, a given module might actually behave much like the module that precedes it. Most hidden-layer connections in `MM(P)` networks lead into the oldest output modules, even when there are several newer output modules in the network as well.

Though results in FBR make sense given the resources and information available to each method, less balanced domains can lead to different results, as is demonstrated next with PP.

6.4.2 Predator/Prey Results

In contrast to FBR, the results in PP are unexpected in that neither `Multitask` nor `MM(P)` performs better than `Control`, and `MM(R)` and `Multinetwork` greatly outperform all of these methods, achieving roughly equal performance. The hypervolume learning curves (Figure 6.9) show `MM(R)` and `Multinetwork` quickly improving and remaining better than all the other methods.

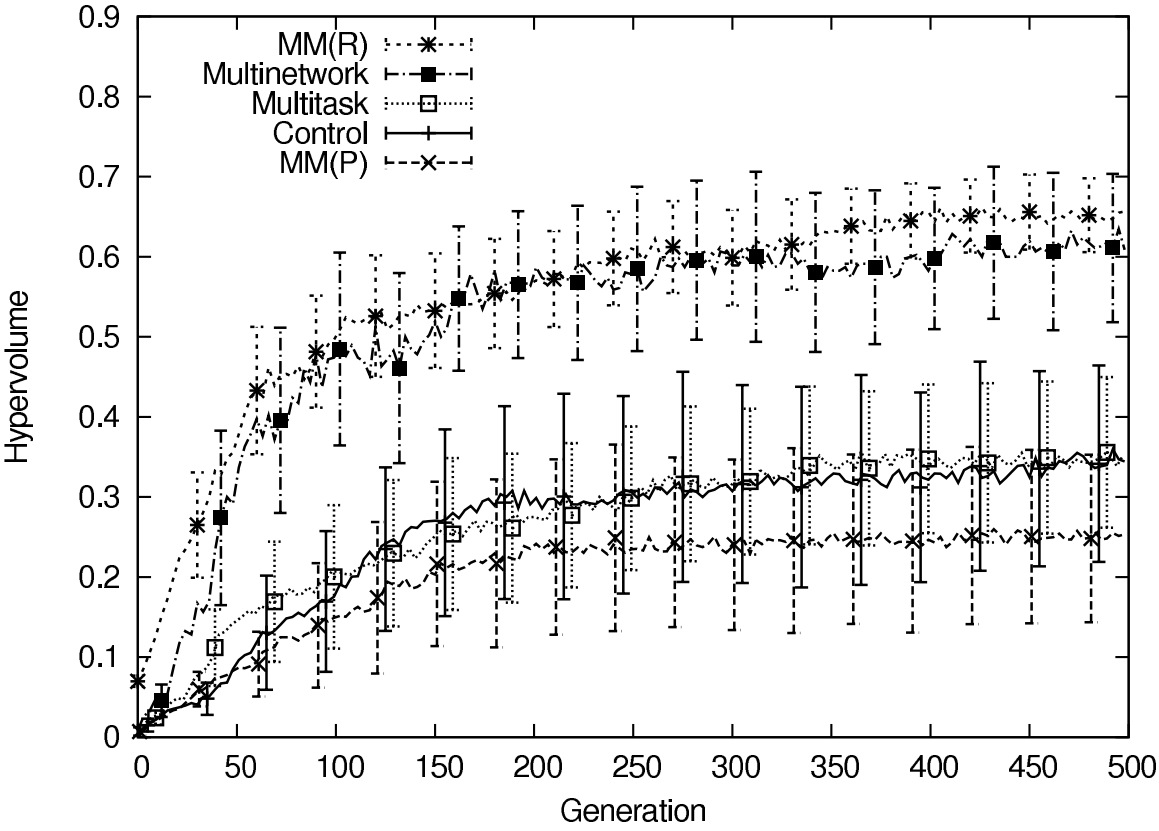


Figure 6.9: **Average Hypervolumes in the Predator/Prey Domain.** For each method, average hypervolumes across 20 runs are shown by generation with 95% confidence intervals. In contrast to results in FBR, `MM(R)` outperforms `Control`, `MM(P)`, and `Multitask` significantly. None of these three methods with lower performance are significantly different from each other, though `MM(P)` is slightly below `Control` and `Multitask`, which are on roughly the same level. `Multinetwork` also significantly outperforms these three low-performing methods, and by the end of evolution has an average hypervolume that is slightly lower, but not significantly different from, that of `MM(R)`. This domain demonstrates a surprising failure of the `Multitask` approach, which should easily develop distinct behaviors for each task, and the impressive success of `MM(R)`, which performs as well as if it could completely isolate both tasks, as `Multinetwork` does. This result thus demonstrates the potential power of discovering modules automatically.

The epsilon indicator values in the final generation confirm these results (Figure 6.10), and the Mann-Whitney U tests (Table 6.7) confirm that the relevant differences are significant.

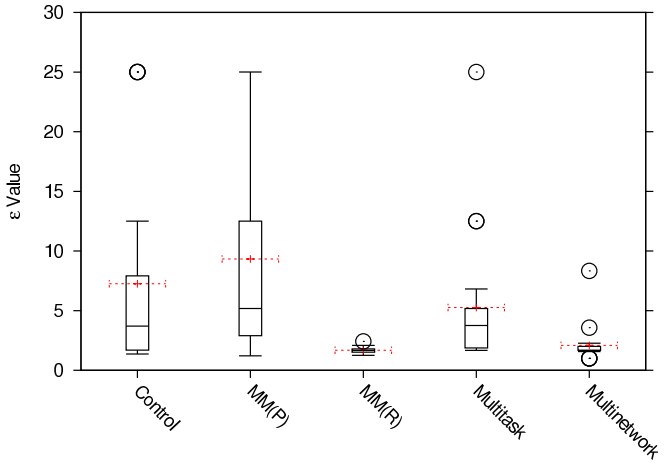
Comparison	HV	I_ϵ^1	$I_{\epsilon+}^1$
`Control` vs. `Multitask`	189	195	195
`Control` vs. `MM(P)`	163.5	159	159
`Control` vs. `MM(R)`	**78.5**	**76**	**76**
`Control` vs. `Multinetwork`	**87**	**89**	**89**
`MM(P)` vs. `Multitask`	135	138.5	138.5
`MM(P)` vs. `MM(R)`	**37.5**	**37.5**	**37.5**
`MM(P)` vs. `Multinetwork`	**52**	**52**	**52**
`Multitask` vs. `MM(R)`	**43**	**40.5**	**40.5**
`Multitask` vs. `Multinetwork`	**66**	**66**	**66**
`MM(R)` vs. `Multinetwork`	180.5	176	176

Table 6.7: **Two-tailed Mann-Whitney U Test Values for the Final Generation of Predator/Prey.** `MM(R)` and `Multinetwork` are significantly better than all other methods in all metrics, but not significantly different from each other. Neither `MM(P)` nor `Multitask` are significantly different from `Control`. The two epsilon indicators have identical U values for all comparisons because the damage-dealt score in the Predator task always determines the epsilon value needed to dominate the reference set. Hypervolume is different, though still consistent, because varying scores in the Prey task result in a slightly different ranking of hypervolume scores than of epsilon scores.

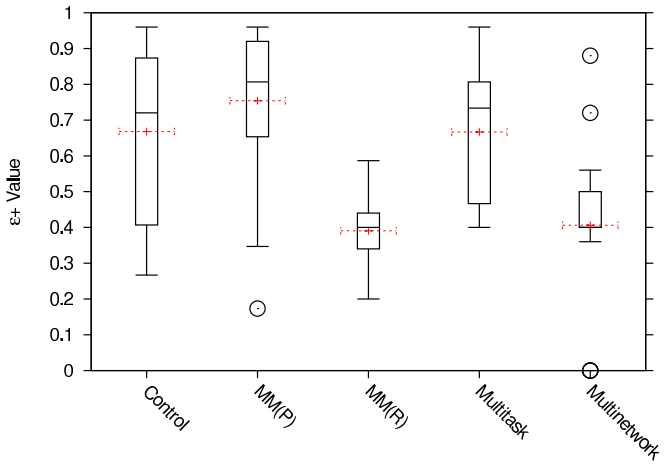
When the Pareto fronts from the final generation of each method are split up by task, it turns out that every run of each method results in at least one individual with perfect scores in the Prey task. This is not surprising since monsters simply need to run away from the bot to avoid all damage and stay alive the whole time, thus having perfect scores in these objectives. Consequently, the Pareto fronts for the Prey task are not different across methods.

In contrast, the Pareto fronts for the Predator task *are* different. Of course, a front for the single-objective Predator task is simply the highest damage dealt in that run. Since different objectives are not being compared, there is no need to normalize. To compare methods, it is sufficient to look at the distribution of best damage-dealt scores (Figure 6.11): The Mann-Whitney U test values for comparing these scores are identical to those for I_ϵ^1 and $I_{\epsilon+}^1$ in the full PP game (Table 6.7). These results show that `MM(R)` and `Multinetwork` are significantly better than all other methods, none of which are significantly different from each other.

Therefore, the main determinant of overall performance in PP is performance in the Predator task. It also primarily determines the form of average success count curves in PP (Figure 6.12). However, pressure to do well in both the Predator *and* Prey tasks is what ultimately makes this domain challenging, since achieving high performance in the Predator task is easier without the additional pressure to avoid damage in the Prey task. This fact is demonstrated by the high perfor-



(a) I_ϵ^1 Values.



(b) $I_{\epsilon+}^1$ Values.

Figure 6.10: **Epsilon Indicator Values in the Final Generation of Predator/Prey.** Given the final Pareto fronts for each of the 20 runs with each method, the (a) I_ϵ^1 values, and (b) $I_{\epsilon+}^1$ values are calculated, and presented in the same manner as in Figure 6.2. `MM(R)` and `Multinetwork` are superior to all other methods. It is surprising that `Multitask` performs so poorly since, like `Multinetwork`, it always has knowledge of the current task, and has a specific policy for each task. It is also impressive that `MM(R)` achieves such high performance, since it does not have this knowledge. There is little difference between `Control`, `Multitask`, and `MM(P)`. In terms of I_ϵ^1 values, `Control` and `Multitask` seem better than `MM(P)`, but Table 6.7 indicates that these differences are not significant. The slight difference between `MM(R)` and `Multinetwork` is not significant either.

mance of Multinetwork in the Predator task.

Another indicator of how easy the Prey task is to solve is the selection of inputs in Prey task networks of the Multinetwork approach. None of the inputs go to saturation, even though the population is solving the task. In other words, there are no particular inputs that are vital to solving the Prey task. Any input that gives a signal of fairly consistent sign can be hooked up to the forward/backward output to make monsters run from the bot. The "Bias" input seems like a natural candidate for this job, but Multinetwork solutions in the Prey task often use other inputs instead.

In contrast, Multinetwork populations in the Predator task and populations from the other methods tend to favor certain inputs strongly, indicating that individuals that started using those inputs gained a large advantage over members that did not. As in FBR, the "Monster/Bot Heading Diff." and "Monster Heading/Bot Loc. Diff." sensors are often used. Each individual population also favors some set of ray-trace and team-slot sensors, but the specifics vary greatly across runs.

The insights gleaned from the empirical data are further supported by observing the evolved behaviors of the monsters (animations at http://nn.cs.utexas.edu/?multitask). Control networks tend to be good in only one of the two tasks, but because the Prey task is so easy, there are also Control networks that succeed in both tasks. The Predator task is more challenging. Sometimes monsters that take damage and die in the Prey task make it into the Pareto front because they deal a large amount of damage in the Predator task.

What is surprising is that Multitask networks do not do better in the Predator task. These networks *always* master the Prey task because they start running from the Predator as soon as evaluation starts; all individuals in all 20 Pareto fronts for Multitask networks in PP get perfect scores in the Prey task. It is easy for Multitask networks to have one policy that makes the monsters run away. However, it is unclear why Multitask networks do not always do well in the Predator task as well. In fact, the best Multitask scores in the Predator task are slightly lower than the best Control scores (Figure 6.11).

A possible explanation is that giving equal attention to each task, as the Multitask architecture requires, is unnecessary and even detrimental in this domain, because the relative challenge of the two tasks is so different. Good Prey behavior thus becomes over-optimized at the expense of good Predator behavior. This problem does not arise with Multinetwork because the networks learning each individual task do not face the extra challenge of learning the other task as well. The task division works well if the tasks are completely separated so that they are not competing with each other.

This trade-off in evolutionary search might also explain why MM(R) does so well: With Module Mutation, evolution is free to choose how many modules to make, and how often to use each of them. While the "obvious" task division may hinder evolution, MM(R) can overcome this problem by finding its own task division. However, this "division" often favors a single module that

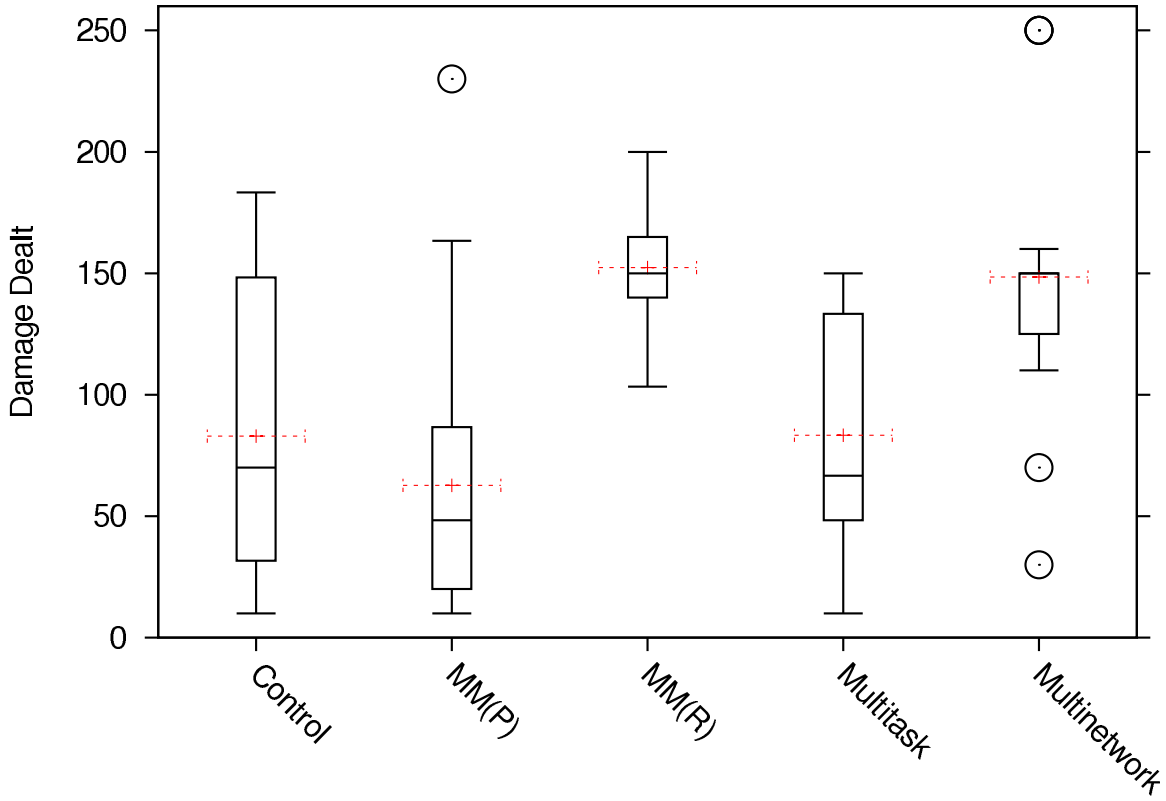


Figure 6.11: **Best Damage-Dealt Scores in the Predator Task of Predator/Prey.** Isolating the Pareto fronts for the Predator task reduces it to a single-objective task. Because most individuals in the Pareto fronts for the full task had perfect damage-received and time-alive scores, these damage-dealt scores are primarily responsible for the differences in hypervolumes in the full task. The main reason that `MM(R)` is better than the other methods is that it always succeeds; it has no low scores at all. Most of the other methods have scores varying over a wider range, and thus much lower averages and medians. The only exception is `Multinetwork`, which, despite some low outliers, has very high median and average performance. It is particularly surprising that the lowest scores in the other methods can be so low. The pressure to perform well in the Prey task is a big distraction to these other methods, and sometimes leads the evolving populations in a one-way direction away from good performance in the Predator task.

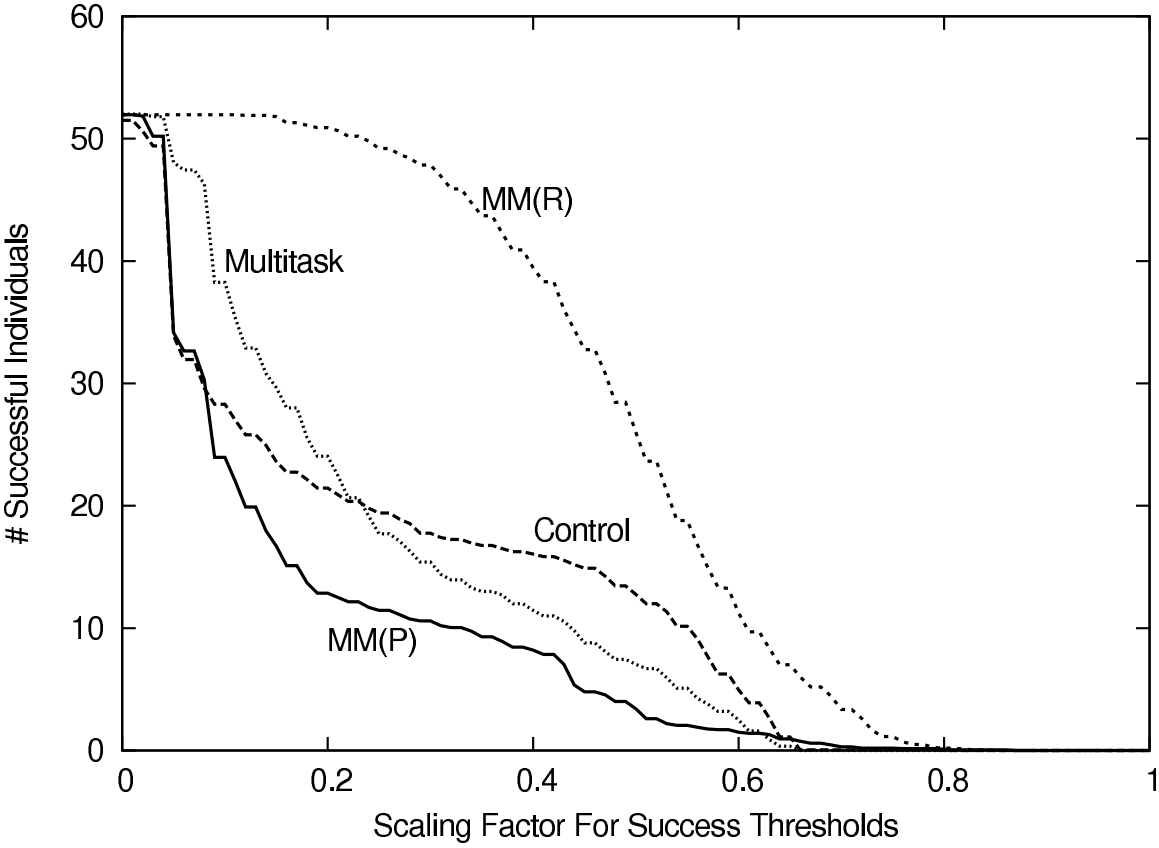


Figure 6.12: **Average Success Counts in Predator/Prey.** The single-network method that has the most high-performing individuals for the highest success thresholds is `MM(R)`, whose curve dominates all others. At low success thresholds, the next best method is `Multitask`, but at higher success thresholds it is overcome by `Control` because `Multitask` networks have relatively low damage scores in the Predator task.

is extensively used in both the Predator and Prey tasks. For example, an MM(R) network scoring $(146.67, 0, 600)$ had 19 modules, but only four were used. Their usage in the Prey task was 87.17%, 0%, 3.33%, 9.50%; and their usage in the Predator task was 83.40%, 0.04%, 0.83%, 15.72%. The first module is thus primarily responsible for controlling the team in both tasks. Furthermore, the fourth module, which is the 2nd most used in both tasks, is used erratically, meaning that it controls the agent for one or two time steps in a row every once in a while before control switches back to the primary module. Even though the function of this seldom used module is unclear, the overall behavior of the agents is multimodal, and very successful.

But why is MM(R) behavior so good, while MM(P) behavior is so erratic? MM(P) networks can be mediocre in both tasks, or spectacular in both tasks. When MM(P) succeeds, it also tends to use few of its modules. For example, an MM(P) network scoring $(155.67, 0, 600)$ had 10 modules, but only used three of them. In fact, only *one* module was used in the Prey task, which makes sense because this task is comparatively easy. However, this same module was the most used module in the Predator task as well, where the usage profile of these three modules was 88.73%, 6.99%, 4.28%. It seems that because MM(P)'s output modules are so interconnected and similar, it is difficult for networks to specialize modules in any way. The chain of interconnected modules is fragile, because improvements to earlier modules can harm later modules, and thus decrease overall fitness. Using fewer modules to control behavior makes MM(P) networks less fragile, but means that individual modules may have to develop multiple modes of behavior, which is also challenging. There are many chances for failure with both of these approaches, which is why MM(P) success depends on luck more so than MM(R). However, the fact that the seldom used modules of this successful MM(P) network are only used in one of the two tasks is evidence of intelligent module usage based on the current task.

The superiority of MM(R) in the PP task is surprising, because it provides an example where evolution uses modules in a way that trumps human knowledge, as represented by the Multitask approach. This result is in stark contrast with the result in FBR, where Multitask is superior to both Module Mutation methods. The next section addresses the differences in PP and FBR that led to these different results.

6.5 Discussion

Interestingly, although Multitask Learning and Module Mutation each work well in at least one of the domains of this chapter, the Multinetwork approach worked well in both. However, these domains are stepping stones towards more difficult domains where tasks are not isolated (such as Ms. Pac-Man: Chapters 9 and 10), and the Multinetwork approach may not be as easy to apply. In order to best exploit these methods in more complex domains, some idea of which methods are most likely to be successful is needed.

First, Multinetwork and Multitask are restricted by needing to know the current

task, whereas MM(R) is not. Since MM(R) does well in PP and better than Control in FBR, it is a good choice for domains in which the task division is not known. However, when task divisions are clear, programmers can simply tell agents what the current task is. Even when the division is not clear, programmers can sometimes use domain knowledge to guess how to split the domain into tasks. Multitask performed well using the task division for FBR, but even in this case it was outperformed by Multinetwork, which also did well in PP.

In fact, the results demonstrate a surprising example of how Multitask Learning can fail (as discussed in Section 3.2.1); the ability to share knowledge about tasks in the hidden neurons of the network was unhelpful in PP, and not helpful enough in FBR to overcome the performance of Multinetwork. Although Multitask can provide a helpful bias for learning, the strength of the human-specified task division has a strong influence on its chances of success.

The limited success of Multitask Learning only applied in FBR, where the task division *properly* split the challenges of the domain. As was shown by PP, where tasks were not equally difficult, Multitask's separate dedicated modules were actually detrimental to evolution, even though Multinetwork performed very well. However, it is surprising that MM(R) performed just as well as Multinetwork, and even more surprising that it achieved success by mostly using only one output module. Thus the main advantage of MM(R) is that it can discover a task division that is effective, albeit counterintuitive to human designers. By extension, Module Mutation may also work well in domains where the tasks are interleaved or blended.

Because the Module Mutation methods never knew which task they were facing, their multimodal behavior was likely established with the help of recurrent connections. For instance, the behavior demonstrated by MM(R) in Figure 6.7 involves a module switch after being hit by the bot, but the hit itself is sensed for only a single time step. Therefore, this knowledge must have been maintained by recurrent connections. This behavior demonstrates how recurrent connections in a modular network can produce multimodal behavior.

This example network used recurrency to maintain memory of brief environmental cues that revealed what the task was. This ability is promising for domains where there is no clear task division; recurrent links could accumulate evidence indicating what the current task is.

Also, the PP domain produced agents via Module Mutation that favored one module, but exhibited multimodal behavior that was successful in two tasks. This behavior seems to have been informed by awareness of the direction in which the bot was moving, which could only be tracked using recurrent connections. This behavior within PP shows how multimodal behavior can be achieved with the help of recurrent connections even when evolution eschews structural modularity by primarily using only one module.

Another way to make a network aware of the current task is to simply give it sensors that directly indicate what the task is. Such sensors are easy to define if the tasks are isolated or interleaved, but when a domain has blended tasks, some uncertainty about the current task is likely to remain. When combined with preference neuron networks, this approach also has the potential

to discover its own novel task divisions. The `Multitask` and `Multinetwork` approaches are required to use a particular human-specified task division, but if this human-specified information is moved into the sensors, then knowledge of the human-specified task division would still be available, but its use would not be mandatory. The ability to learn the task division would prevent the kind of failure that `Multitask` experienced in the PP domain. The impact of such sensors on the evolution of multimodal behavior will be studied later in Ms. Pac-Man (Chapters 9 and 10).

These experiments also reveal some ways in which Module Mutation can be improved. Although `MM(P)` is better than `Control` in FBR, it performed poorly in PP. New `MM(P)` modules are only imperfect copies of their source modules, and it is impossible to modify the source module without also modifying the new module. These shortcomings of `MM(P)` may have contributed to its failure in PP. These are the reasons that Module Mutation Duplicate was developed (MM(D); Section 3.2.3), and will be used in place of `MM(P)` in Chapters 9 and 10.

Although different methods excelled in each of the two domains of this chapter, methods that have multiple modules were always the best. Thus, multimodal approaches provide a clear benefit in domains consisting of multiple isolated tasks.

6.6 Conclusion

Two domains with isolated tasks, Front/Back Ramming (FBR) and Predator/Prey (PP), were used to evaluate two methods of evolving modular networks: Multitask Learning and Module Mutation. These approaches were compared against a control involving networks with a single module, and a method of combining separately evolved networks called Multinetwork.

In FBR, where the task division is both obvious *and* balanced, Multitask Learning is effective, although not as good as the Multinetwork approach. Module Mutation methods come in third, but still ahead of networks with just one module. In PP a form of Module Mutation, named MM(R), tied with the Multinetwork approach as the most effective method. MM(R) succeeded by efficiently searching the space of policies to find one module that worked well with other modules, that were used far less.

Multinetwork, Multitask Learning, and Module Mutation thus allow evolved agents to have multiple policies to fit different situations. Such an ability is useful in developing multimodal behaviors for domains with isolated tasks, and should be useful for more challenging domains as well. In subsequent chapters, these techniques will be developed further, and evaluated in challenging variants of Ms. Pac-Man featuring both interleaved and blended tasks (Chapters 9 and 10). However, the next chapter deals with a different approach to discovering multimodal behavior: Fitness-based shaping using TUG.

Chapter 7

Evaluation: Fitness Shaping in Blended Tasks

Multiobjective evolution helps discover multimodal behavior because different objectives are often associated with different modes of behavior. NSGA-II discovers trade-offs between all objectives, and the best solutions will have competence in multiple objectives because they have different modes of behavior associated with each objective.

However, the link between objectives and behavioral modes may not always be completely clear, especially in a domain with blended tasks. The extra challenge posed by learning how to respond in each particular situation may cause evolutionary search to get stuck in local optima. This chapter presents experiments using Targeting Unachieved Goals (TUG; Section 3.3), which is designed to focus search in such a way that it escapes the local optima and increases scores in the objectives that need the most attention. This chapter first describes the experimental setup used to evaluate TUG in the Battle Domain (BD), which has blended tasks of attack and defense, and then covers the results.

7.1 Experimental Setup

This section describes the experiments conducted in BD to evaluate the performance of TUG. First the manner in which evolved networks interact with the environment of BD is described, then the different approaches that are evaluated, and finally, the parameters that influence neuroevolution.

7.1.1 Agent Control

Because the evolved monsters in these experiments are simulated in BREVE environments similar to FBR and PP, these monsters interact with the environment in very similar ways. Monster behavior is defined by two policy outputs, just as in FBR and PP: One controls the forward/backward impulse,

and the second controls the left/right turning. Input sensors are also very similar. Monsters in BD have access to all of the sensors used by FBR and PP monsters defined in Tables 6.1, 6.2, and 6.3.

However, there is one extra feature of BD that is important enough to justify the use of additional sensors: the bot's bat. The bat is a dangerous object controlled by the bot, so the monsters need to sense it in order to learn intelligent multimodal behavior. Therefore, monsters in BD have seven additional sensors. There are two new binary sensors: "Close to Bat" and "Very Close to Bat." A monster is close if it is within 9.5 distance units, and very close if it is within half this distance (4.75 distance units). These values were determined through trial and error in preliminary experiments. For reference, the length of the monster and bot agents is 2 units, and the length of the bat itself is 5 units.

The remaining five new sensors are ray traces similar to those described in Table 6.3 for FBR and PP. The ray traces are affixed to the monsters in the same way as the other ray traces, but they sense the bot's bat. These ray traces are particularly useful because the signals they give will change as the bat swings through them, allowing the monsters to learn precise maneuvers for avoiding the bat and launching counter-attacks.

With these additional sensors, neural networks can be evolved to control the monsters in BD. The details of how this is done are described next.

7.1.2 Selection Methods

The experiments in this chapter are designed to show the benefits of TUG (Section 3.3) in a domain with blended tasks (Section 4.4). TUG is an alternative approach to selecting which members of a population are considered best in each generation. Regular NSGA-II always uses all objectives to determine which individuals are best in terms of Pareto dominance. TUG also uses the Pareto dominance criteria, but does not always use all objectives. The objectives in use at any given time depend on the current goal values and the performance of the population from generation to generation. The current goal values depend on the initial goal values and how they change over time.

Three methods are tested in these experiments: NSGA-II by itself (`Control`), and NSGA-II combined with TUG using two different sets of initial goal values. Of these, `TUG Low` uses starting goal values that are the absolute minimum possible in each of the three objectives of BD (Section 5.4), and `TUG High` uses higher goal values that are set at levels that represent a reasonable level of performance in this domain. The specific goal values for each objective are:

1. *Maximize Damage Dealt* $= 50$*:* The bot has 50 health points, so this goal requires the monsters to kill the bot at least once per trial. The bot respawns after death, giving the monsters a chance to inflict more damage.

2. *Minimize Damage Received* $= -20$*:* Bat strikes deal 10 damage points each, so each monster should take no more than two hits on average. However, because this value is averaged across

team members, it is possible to achieve this goal even if one team member dies (50 damage), since the average across the four team members could still be above -20.

3. *Maximize Time Alive* = 540*:* On average across team members, monsters must survive throughout 90% of the trial. It is still possible to achieve this goal even if some monsters die in fewer than 540 iterations (recall the total number of iterations per trial is 600).

Each method was evaluated in 30 runs. The parent and child population sizes were $\mu = \lambda = 52$, and individual runs lasted 500 generations. Every network was evaluated three times, and its objective scores were averaged across evaluations in order to get more reliable scores with noisy evaluations.

Recall that TUG has a parameter that influences how long performance above each current goal must persist to be considered achieved (Section 3.3). Specifically, this parameter determines how quickly the recency-weighted average for each objective catches up to the current performance level. The value of this step-size parameter was $\alpha = 0.15$. Both TUG methods also made use of variable goal increases: Whenever all three objectives were achieved, each recency-weighted average was reset, and the goal value for each objective was moved closer to the current maximum score in that objective. The step-size parameter by which goal values increased was $\eta = 0.15$.

The remaining parameters affecting these experiments relate to how neuroevolution is used.

7.1.3 Neuroevolution Parameters

All neural networks in these experiments consisted of a single module. TUG will be combined with modular network architectures in Chapter 10.

The parameters affecting neuroevolution were the same as in the experiments in FBR and PP (Chapter 6): Constructive neuroevolution was used with feature selection (Whiteson et al., 2005). Crossover was not used, but mutations were applied when a child population was produced: There was a 40% chance that a single weight would be perturbed, a 20% chance that a single (potentially recurrent) link would be added, and a 10% chance that a new neuron would be spliced along an existing link.

In each trial, the network being evaluated was copied into each monster to create a homogeneous team as described in Section 5.1. On every time step, each monster acted in accordance with its neural network brain, as in the FBR and PP experiments. The evolution of these agents in BD lead to the results described next.

7.2 Results

First the manner in which results are evaluated is discussed, followed by the actual quantitative results, and analysis of the evolved behaviors.

7.2.1 Assessing Multiobjective Performance

Although the Battle Domain is challenging, performance assessment in this domain is easier than in FBR and PP. First, there are only three objectives in comparison with FBR's six. Second, although the different objectives correspond to different blended tasks, they are not explicitly split across isolated tasks.

As a result, looking at the raw scores and Pareto fronts is more informative. However, this information is still supplemented by calculating the hypervolume metric (Section 2.1.3) for the final population of each run. Hypervolume calculations are based on normalized objective scores scaled between the minimum and maximum scores in the final generation for each objective across all runs.

The minimum point for BD was $(0, -49.166667, 331.916667)$. The 0 indicates that no damage was dealt to the bot. The -49.166667 indicates that across three evaluations, all four monsters died, except in one evaluation where one monster barely escaped death with 10 health points remaining. The 331.916667 is the lowest average survival time of the four monsters in a team across three evaluations. The maximum point was $(250, 0, 600)$, indicating that the best team of monsters killed the bot five times, and the best surviving teams managed to go through all evaluations without receiving any damage.

This information is used to analyze the results in BD in the next section.

7.2.2 Performance

Both forms of TUG achieve higher performance than NSGA-II alone. When two-tailed Mann-Whitney U tests are used to compare TUG to the `Control`, both `TUG High` ($U = 113, p < 0.001$) and `TUG Low` ($U = 98, p < 0.001$) have significantly higher final hypervolumes (Figure 7.1). The difference between `TUG Low` and `TUG High` is not significant, indicating that TUG can succeed even without expert knowledge in the form of initial goal values.

The methods can also be compared in terms of the final Pareto fronts. In this case it is most informative to combine all 30 Pareto fronts from each method into a single set per method (Figure 7.2). For these experiments, it is better to look at the combined Pareto fronts of all runs instead of their super Pareto front because it is then clear how poorly the worst runs of each condition performed. However, even the worst TUG runs are very good: Both `TUG Low` and `TUG High` often produce results that dominate many individuals in the Pareto fronts of plain NSGA-II runs.

Although TUG's Pareto fronts are generally strong, TUG runs do sometimes have low-scoring individuals in their populations, due to the deactivating and reactivating of objectives. Observation of learning curves from individual TUG runs (Figure 7.3) indicates that TUG runs go through cycles in which the number of high-performing individuals is maximized at the point where all goals are achieved, and then lowered soon afterwards when all goals are increased and reactivated. Therefore, it is best to terminate TUG at a point where all goals have just been achieved

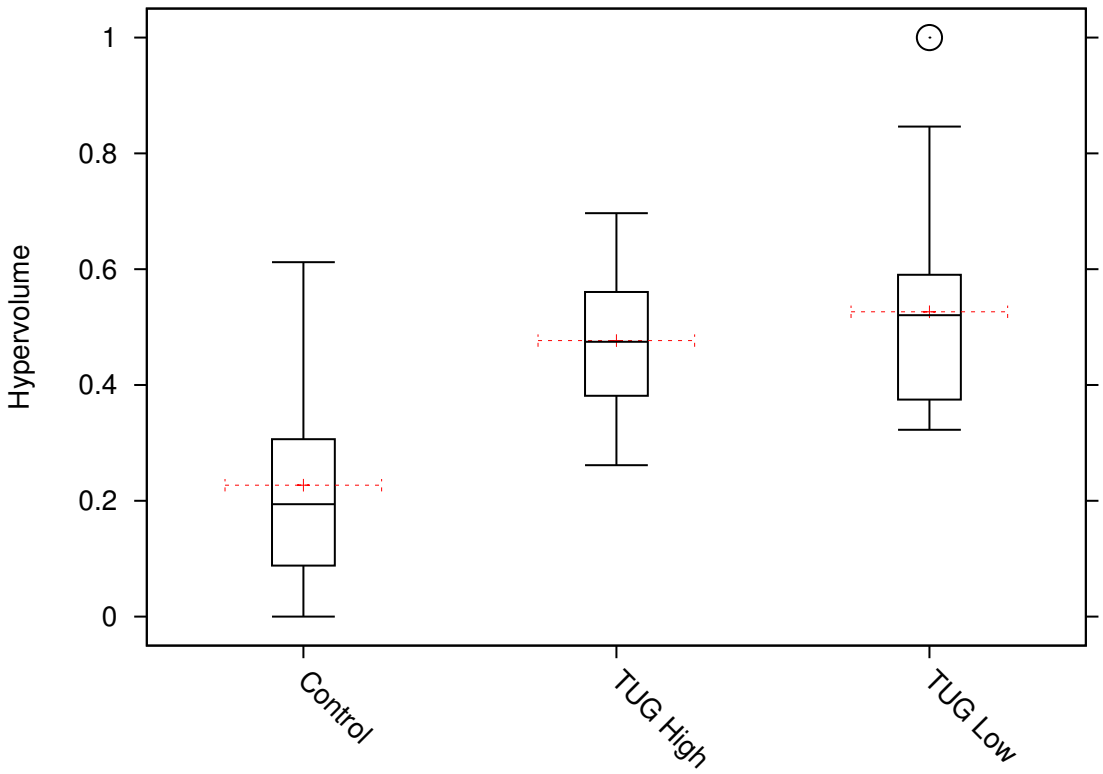


Figure 7.1: **Hypervolumes in Final Generation of Battle Domain.** Given the final Pareto fronts for each of the 30 runs with each method, the hypervolumes are calculated and shown as box-and-whisker plots (depicting the minimum, lower quartile, median, upper quartile and maximum scores, with scores more than $1.5IQR$, or inter-quartile range, from the nearest quartile shown as outliers). Additionally, the dashed line intersecting each box is the average hypervolume. The hypervolumes for `TUG High` and `TUG Low` are much higher than those for `Control`. The best `TUG Low` run actually has a hypervolume of 1.0, because all of the maximum scores used for the hypervolume scaling came from this run.

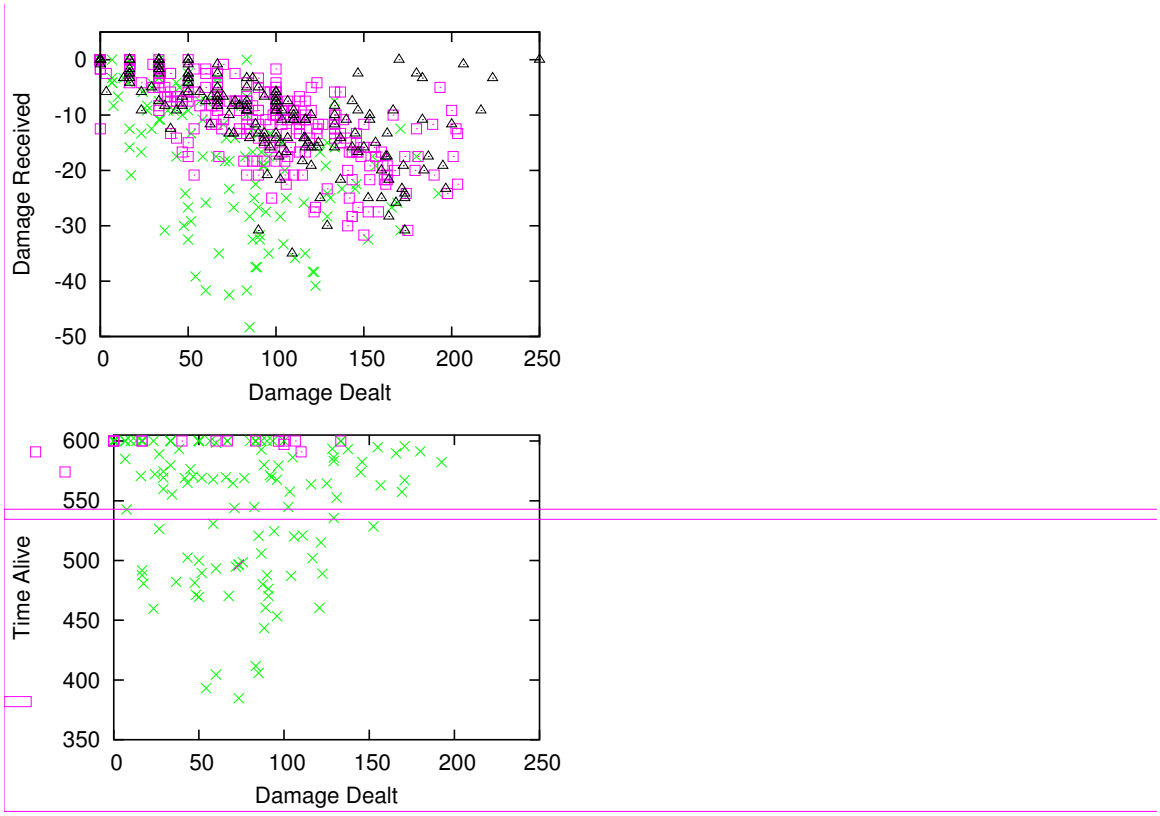


Figure 7.2: **Unions of Pareto Fronts in Battle Domain.** In each of the trials of each method, a Pareto front is produced. Though TUG can turn objectives off, these fronts are calculated with respect to all three objectives of the Battle Domain. The unions of such Pareto fronts for each method are shown in the figures above. Each possible pairing of two objectives is shown in one of the 2D plots, and a full 3D plot is shown at bottom right. Many of the points in the Pareto fronts of `Control` runs are dominated by `TUG High` and `TUG Low` points. The absolute best points occur in `TUG Low` runs, though both TUG-based approaches successfully create fronts dominating `Control` fronts.

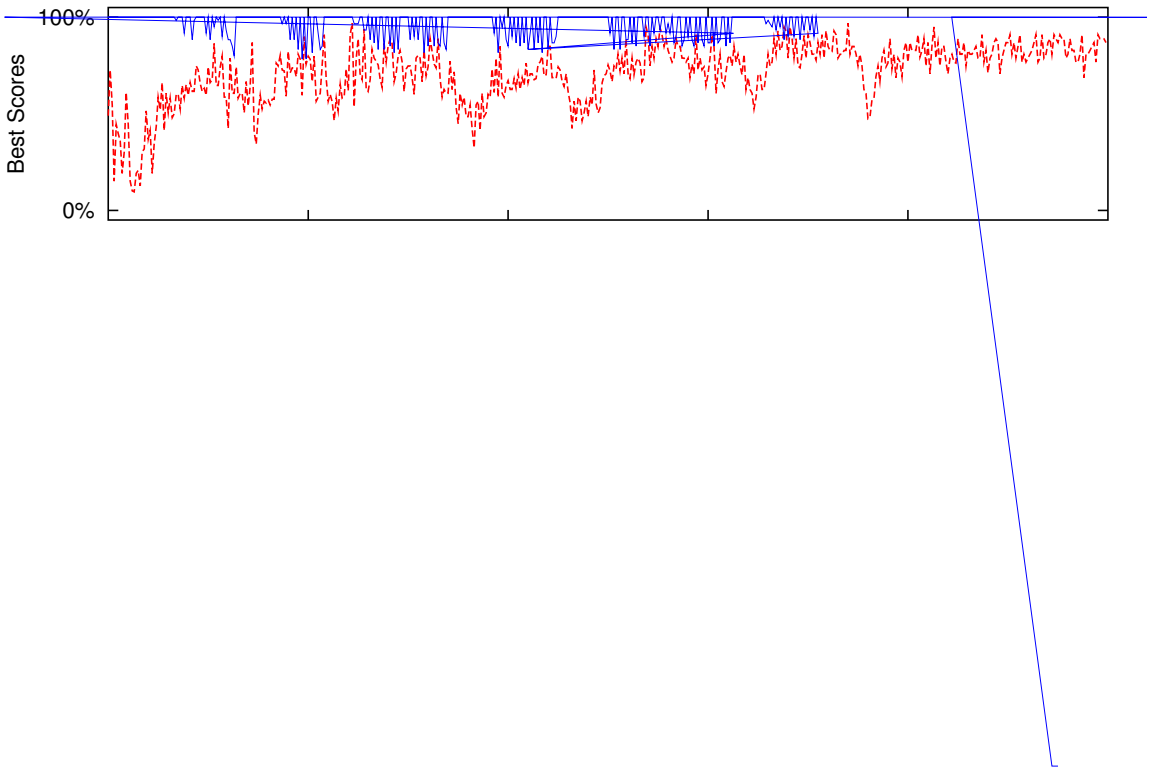


Figure 7.3: **Example Objective Behavior of a `TUG High` Run.** The best, average, and worst scores in each objective by generation for an individual run using `TUG High` is shown. Objective scores are normalized to a common range and measured as percentages of the maximum attained values. Vertical lines on the plot signify generations at which TUG achieved all goals, and thus had its goal values increased. Remember that all objectives are reactivated after goals are increased. The plot for Time Alive is always on top, since it is the easiest objective. Damage Received is just beneath Time Alive, and its plot fluctuates in a similar manner because Time Alive is also affected by receiving damage. Damage Dealt is clearly the hardest objective, since its plot is always beneath the plots of the other objectives. Damage Dealt drops after each vertical line because the reactivation of the other two objectives makes it hard to deal damage, yet as the goals for Damage Received and Time Alive are achieved, selection focuses more on Damage Dealt until all goals are achieved. Therefore, the population scores are always at their best on generations when all goals are achieved.

rather than after a fixed number of generations.

To gain further insight into these results, representative behaviors generated by each method will be analyzed next.

7.2.3 Behavior

As expected from the results above, the behaviors evolved with TUG tend to be better than those evolved with plain NSGA-II. Movies of characteristic behaviors can be viewed at `http://nn.cs.utexas.edu/?fitness-shaping`

The most effective behavior occurred in a `TUG Low` run (Figure 7.4). Across the three trials that the team faced, it avoided all damage and thus stayed alive the whole time. However, this team is exceptional in that it also dealt an average of 250 damage, which amounts to killing the bot five times in each evaluation. Monsters in this team approach the bot from an unusual angle in order to confuse it into focusing on the wrong monster. Then the monsters turn towards the bot and are able to consistently strike it on its left side, which is safe because the bat starts swinging from the right. Each successful hit cancels the current bat swing, enabling monsters to quickly defeat the bot.

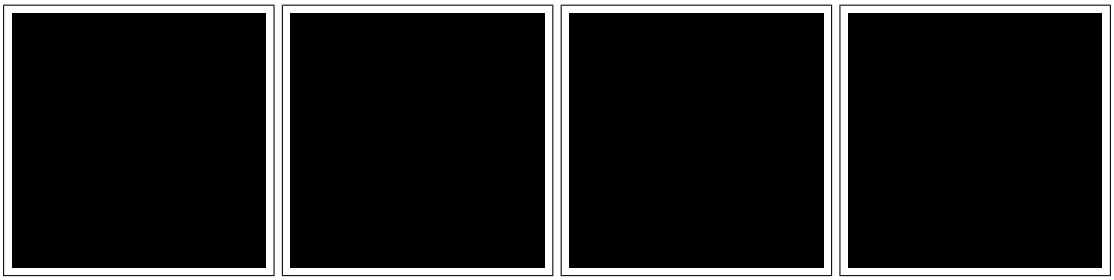


Figure 7.4: **Illustrations of Best `TUG Low` Behavior in Battle Domain.** Behavior progresses in time from left to right. First the bot moves towards the upper left, turning toward the monster on top (Frame 1). The monster on the left turns away (Frame 2) and then towards (Frame 3) the bot while approaching, so that it is able to strike (Frame 4), which cancels the bat swing. This skillful behavior results in high damage dealt with no damage received.

Another successful `TUG Low` behavior demonstrates the blended nature of the tasks in this domain directly (Figure 7.5). This team's trick is to rush at the bot, pause for just the right amount of time as its bat swings past, then rush in to attack the bot repeatedly until it dies. The monsters learn precise timing for switching from offensive (initial rush) to defensive (pause while bat swings by) and then back to offensive behavior (final attack). Similar behaviors emerge in some `Control` runs, but the timing is less precise, which leads to more damage received by these inferior networks.

Another behavior that is good at both avoiding and dealing damage is based on a clever turning maneuver at the start of each trial (Figure 7.6). The monster moves towards the bot while turning left, and after it barely dodges a bat swing the monster starts backing into the bot. The

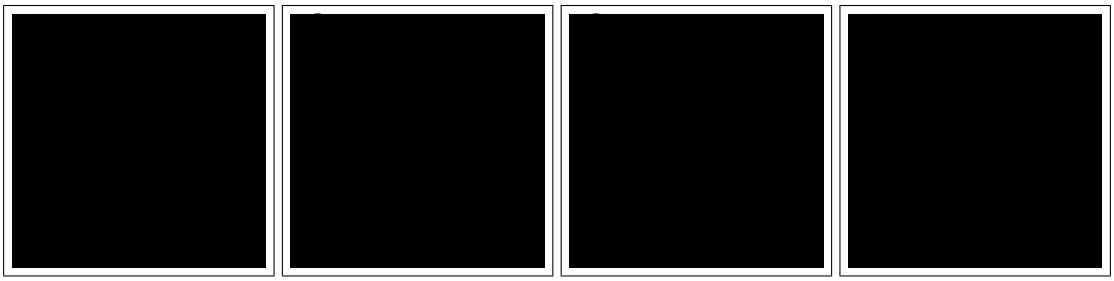


Figure 7.5: **Illustrations of `TUG Low`'s Rush-Pause-Rush Behavior in Battle Domain.** The monster on the bottom approaches the bot (Frame 1), backs up to wait for the bat to swing past (Frame 2), then rushes in just at the end of a bat swing (Frame 3) to be able to attack the bot repeatedly (Frame 4). This behavior requires expert timing, and demonstrates how the agents must learn where the threshold between the blended tasks of offense and defense are.

monster then continues to strike while turning, and thereby manages to avoid the bat. This behavior is slightly less efficient than the ones above due to the turning motion. Since the maneuver takes more time to execute, monsters cannot deal as much damage within the same amount of time. This behavior occurred in both `TUG Low` and `TUG High` runs, but not in the `Control` runs.

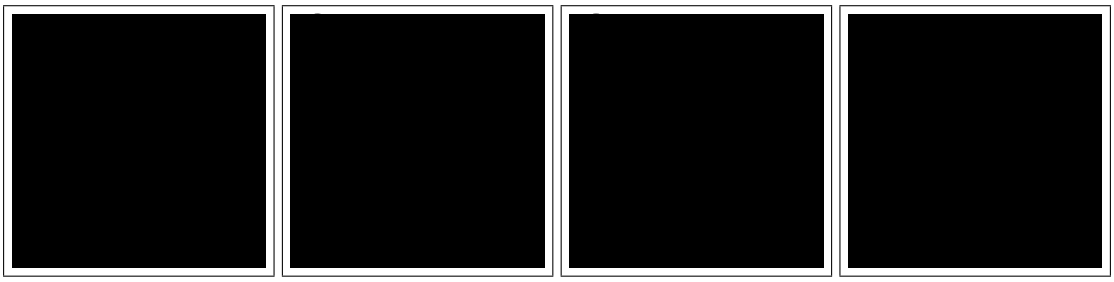


Figure 7.6: **Illustrations of `TUG High`'s Turn-Reverse Behavior in Battle Domain.** As the bot approaches the monster on the right (Frame 1), the monster moves toward the bot while turning to its left (Frame 2). As a result, the bot's bat narrowly misses the monster (Frame 3), allowing it to then go on the attack by reversing into the bot (Frame 4). This behavior blends an offensive advance with a defensive turn so that the monster can sneak past the bat.

Another set of behaviors popular in TUG runs, but not in `Control` runs, employed baiting motions by one of the monsters (Figure 7.7). A monster backs away from the bot while turning such that it has a greater risk of being hit by the bat. Yet it also slows down the progress of the bot so that teammates can sneak up from behind to attack. Though this behavior requires teamwork and is visually compelling, it is not as efficient as other behaviors because extra time is required to move the bot into a vulnerable position via baiting. There is also a greater risk that one of the monsters will be hit. Still, this strategy is another interesting way to deal with blended tasks: The defensive baiting action of one monster actually helps other monsters launch offensive strikes from behind.

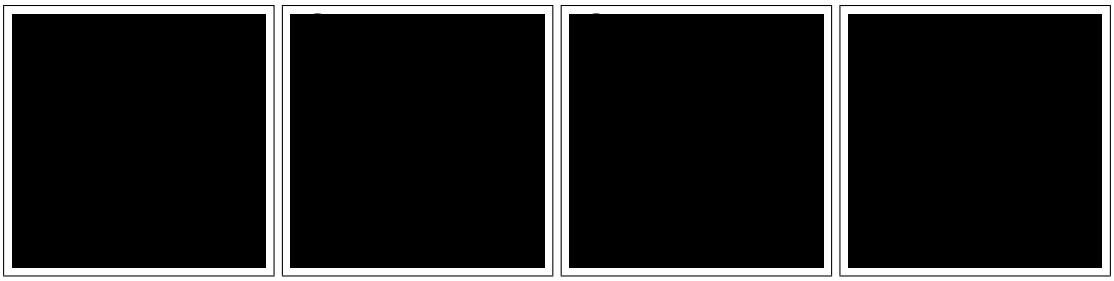


Figure 7.7: **Illustrations of `TUG High`'s Baiting Behavior in Battle Domain.** The monster in the upper-right is retreating while the monster in the upper-left sneaks up from the side (Frame 1). The bot is focusing on the monster in the upper-right, and manages to land a hit with its bat (Frame 2), but this focus allows the other monster to get close (Frame 3) and start attacking the bot (Frame 4). This behavior is an example of tasks being split up across teammates, since one monster is on the defensive while another sneaks in to attack.

The TUG runs also evolved an effective coordinated counter-clockwise attack behavior (Figure 7.8). Because the bot swings its bat from right to left, it is safer to attack it on its left side. Starting from the monster on which the bot currently focuses, and moving counter-clockwise around the bot, the next monster will always be in a position to attack the bot on its left side. Therefore, the monsters will be able to repeatedly blindside the bot. Once again, the team of monsters trades off defensive and offensive roles between teammates. This behavior is generally effective, but tends to result in large damage received because it is hard to get it coordinated precisely.

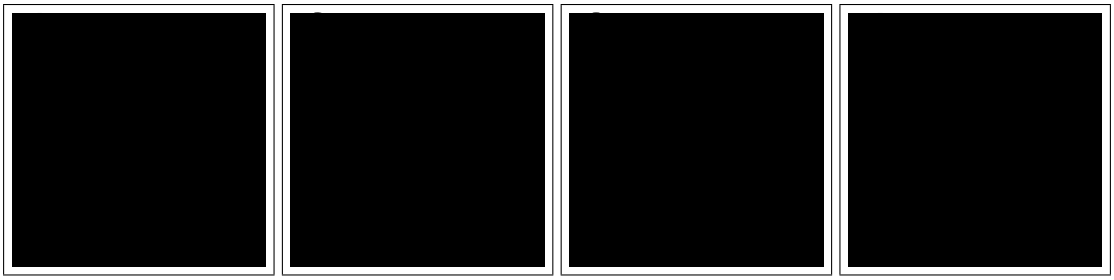


Figure 7.8: **Illustrations of `TUG Low`'s Counter-Clockwise Striking Behavior in Battle Domain.** The monsters surround the bot and rush in between bat swings (Frame 1). First the monster on top attacks (Frame 2), which leaves the bot vulnerable to a follow-up attack from the left (Frame 3), which sets up another attack from below (Frame 4). Each monster passes on the responsibility of filling an offensive role to a neighboring monster in rapid succession.

`Control` runs were characterized primarily by reckless behaviors that would sacrifice life in order to deal damage, and cowardly behaviors that would run away to avoid damage. Often different members of the same population exhibit these opposing behaviors, which makes sense given that they represent different trade-offs between objectives. However, some `Control` trials did achieve

decent performance, usually by approximating the coordinated attack behavior described above.

The variety of ways that TUG learns to deal with blended tasks is interesting. Some runs learn teamwork, in which different monsters trade off offensive and defensive roles, so that each task is occurring at the same time for different agents. Other runs depend on the skill of a single monster that must precisely time its attack, which it usually accomplishes by merging offensive and defensive actions, e.g. by rushing in then pausing, or approaching the bot from an unusual angle that looks defensive.

In contrast, `Control` runs often learn bad behavior, or at best learn poor approximations of good TUG behavior. Evolutionary search pushed these runs in the right general direction, but the networks were unable to refine their behaviors and become truly skilled. It seems they were stuck in the kinds of local optima that TUG is designed to bypass. Thus TUG's high performance in BD demonstrates its potential in domains with blended tasks.

7.3 Conclusion

This chapter applied TUG to a domain with blended tasks: the Battle Domain. BD blurs the line between when monsters should be offensive and when they should be defensive, and this distinction proves hard to learn using vanilla neuroevolution plus NSGA-II. However, using TUG results in great success. Because it is easy to optimize the Damage Received and Time Alive objectives by being overly defensive, the Damage Dealt objective does not always receive enough attention. TUG fixes this imbalance by turning off the easier objectives when they are not needed.

The neural networks in this chapter all consisted of a single module. TUG will be combined with modular neural networks in a more challenging real-world domain in Chapter 10, but first this domain needs to be described. The domain in question is the classic arcade game of Ms. Pac-Man.

Chapter 8

Domain Description: Ms. Pac-Man

Pac-Man and its sequel Ms. Pac-Man are among the most popular video games of all time. This popularity extends into the computational intelligence research community, as evidenced by numerous papers and two different annual competitions. Pac-Man is interesting because a simple set of rules gives rise to a game in which complex strategies are needed to succeed.

At its core, Pac-Man can be thought of as a predator-prey scenario — with a twist. For the most part, the player-controlled Pac-Man is the prey of computer-controlled ghosts, but if Pac-Man consumes a power pill, this state of affairs is reversed: The ghosts temporarily become the prey of Pac-Man. The sudden switch in game dynamics requires a sudden switch in play strategy on the part of the player. In other words, multimodal behavior is required to succeed in Pac-Man.

Despite the inherent multimodal nature of Pac-Man, most learning approaches to Pac-Man have focused on learning monolithic policies that control the Pac-Man agent regardless of whether or not predator or prey ghosts are present. This chapter details that previous work, describes the particular Pac-Man simulator used in this work, and elaborates on the need for multimodal behavior in Pac-Man.

8.1 Previous Pac-Man Research

Pac-Man-style games have been studied for a long time, but until recently, individual researchers tended to create their own simulators for their studies (Koza, 1992; Rosca and Ballard, 1996; Gallagher and Ledwich, 2007; Wirth and Gallagher, 2008; Martín et al., 2010). This diversity of simulators was problematic both because they made fair comparisons impossible, and because in some cases the custom simulators were much simpler or less challenging than the original game.

For example, Koza (1992) used Genetic Programming (GP) to learn Pac-Man behavior in a custom simulator, but his variant of the game was easy in comparison with the arcade version (Lucas, 2005; Svensson and Johansson, 2012; Recio et al., 2012). Other researchers used Koza's simulator rules since his results provided the first basis for comparison. For example, Rosca and Ballard

(1996) improved upon Koza's GP approach using a *Hierarchical* GP approach that encapsulated portions of genomes for use in other genotypes. Hierarchical behavior is essentially synonymous with multimodal behavior, so this particular approach will be discussed later in Chapter 11. In each follow-up study using Koza's rules, new simulators were built from scratch, as late as 2007 when Szita and Lőrincz used a cross-entropy method to learn rule-based policies for Pac-Man. In contrast, Gallagher and Ledwich (2007) focused on an even simpler version of the game featuring only one ghost, but they were attempting to solve the harder task of evolving a neural network controller whose inputs were the raw input grid immediately surrounding the agent.

Even the original game of Pac-Man has a shortcoming that makes it a poor choice for computational intelligence research: All ghost behaviors are deterministic. Therefore, it is possible to maximize one's score by following memorized paths, without having any sort of situational or strategic intelligence regarding how to respond to the actions of the ghosts. The determinism of the original Pac-Man is the reason why nearly all current research in Pac-Man focuses on Ms. Pac-Man instead.

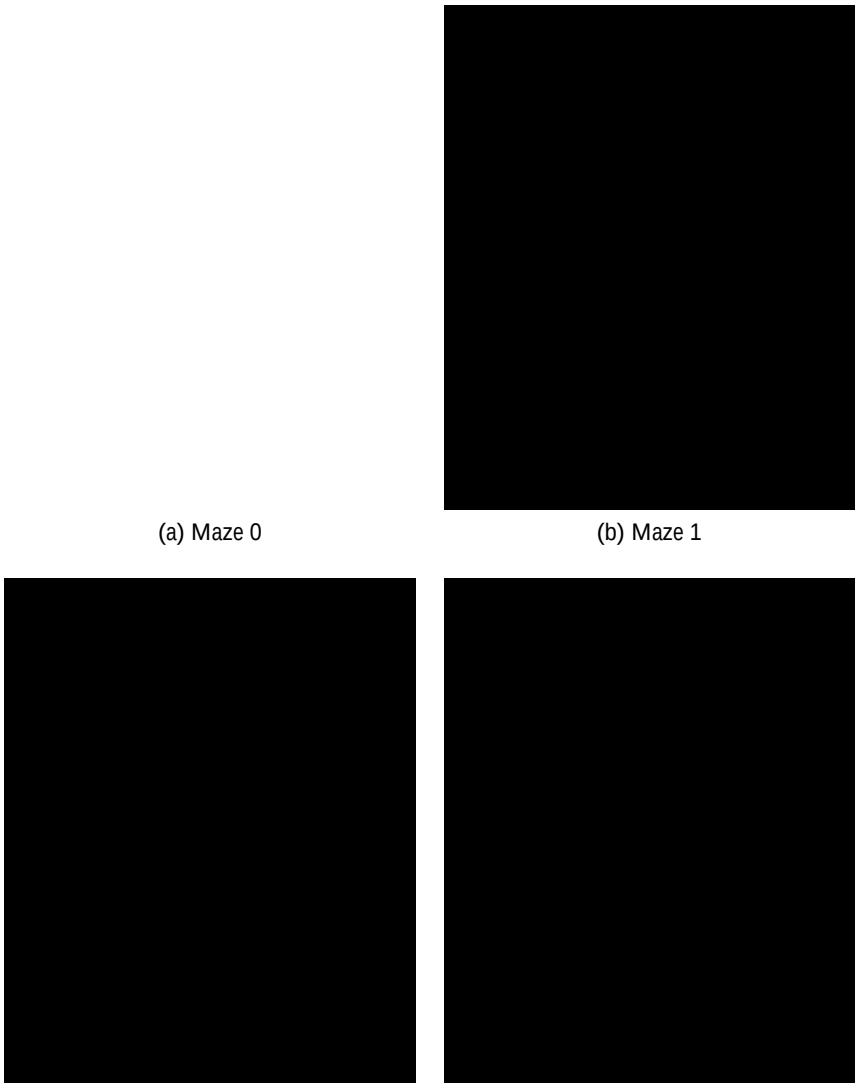
Ms. Pac-Man is the non-deterministic sequel to Pac-Man. Microsoft's Revenge of Arcade port of this game was used in the annual Ms. Pac-Man screen-capture competition[1] at both the Computational Intelligence in Games and the Congress on Evolutionary Computation conferences from 2007 to 2011. The non-determinism makes evaluations noisy, which in turn makes learning in this domain hard. Other than the non-determinism, the most obvious difference in game-play is that Ms. Pac-Man has four different mazes in comparison to Pac-Man's one (Figure 8.1). Because of these two differences, success in Ms. Pac-Man depends more on generalization than memorization.

In the Ms. Pac-Man screen-capture competition, Thawonmas and others constructed a rule-based system by hand (Thawonmas and Matsumoto, 2009), and later used Evolution Strategies to optimize parameters in this system (Thawonmas and Ashida, 2010). Further, Handa and Isozaki (2008) made use of evolved fuzzy logic systems, while Wirth and Gallagher (2008) created an influence map model to play the game. In 2009, Robles and Lucas (2009) adapted traditional game-tree search to work in Ms. Pac-Man, and in the most recent competition, Ikehata and Ito (2011) used Monte-Carlo Tree Search in their winning entry. Though they did not win, Tong et al. (2011) also made use of Monte-Carlo Tree Search that year. The competition has not been run since 2011, but research continues in the screen-capture version of the game: Foderaro et al. (2012) painstakingly modeled the idiosyncratic details of the ghosts' behaviors[2] and decomposed the corridors and junctions of the mazes into cells in order to learn a decision-tree-based policy that outperformed Monte-Carlo Tree Search. However, it should be noted that the detailed, human-supplied ghost model is likely what led to this success.

A common conclusion throughout these papers is that the quality of any learning method is greatly affected by the quality of the screen-capture procedure used to assess the current game

[1] `http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html`

[2] Based on `http://home.comcast.net/~jpittman2/pacman/pacmandossier.html`



(a) Maze 0 (b) Maze 1

(c) Maze 2 (d) Maze 3

Figure 8.1: **Ms. Pac-Man Mazes.** The game of Ms. Pac-Man consists of four mazes. The starting state of each maze is shown in this figure. Ms. Pac-Man starts in the lower center of each maze, and the ghosts start in the lair, which is slightly above the center. At the start of each maze, all pills are present and available to be eaten by Ms. Pac-Man. Power pills are located near the four corners of each maze. A maze is beaten once all pills and power pills in a maze have been eaten. Though the general goals in each maze are the same, the unique structure of each maze presents a unique challenge. Despite the different structures in each maze, Ms. Pac-Man must learn behavior that generalizes across all four in order to succeed.

state. In order to separate issues of computer vision from issues of machine learning, Lucas (2005) developed a Ms. Pac-Man simulator that has gradually become standard for research on Ms. Pac-Man.

This simulator has changed a great deal since it was first introduced. Initially, it was designed to evolve after-state evaluating neural networks (Lucas, 2005), but it gradually improved as Lucas and others used it for other research projects, such as showing how evolved multi-layer perceptrons (MLPs) outperformed temporal difference learning using both interpolated tables and MLPs (Burrow and Lucas, 2009), and showing how game-tree search could be applied to Ms. Pac-Man (Robles and Lucas, 2009).

The current version of the simulator is stable, and was used as part of the Ms. Pac-Man vs. Ghosts competitions[3] in 2011 and 2012. The primary appeal of this simulator is that it allows controllers for both Ms. Pac-Man and the ghosts to be programmed. However, it also comes with a standard `Legacy` team that is an approximation of the ghost team in the commercial game. Several studies have used the `Legacy` team as a basis for comparison. Alhejali and Lucas (2010, 2011) used two different Genetic Programming approaches to evolve Ms. Pac-Man policies, and also used Genetic Programming to improve the random simulations conducted by Monte-Carlo Tree Search (2013). Brandstetter and Ahmadi (2012) also used Genetic Programming, but emphasized use of primitive actions (up, down, left, right) with simple direction-oriented sensors, which is also the approach taken in this dissertation (Section 9.2). Other work using Monte-Carlo Tree Search includes that of Samothrakis et al. (2011) and Pepels and Winands (2012). The results by Samothrakis et al. are particularly impressive, though they made the unusual choice to only evaluate their agent in the first maze. Pepels and Winands's results are also good, but are obtained against a different team than the standard `Legacy` team used by most other researchers. Ant Colony Optimization is yet another technique that has been applied to this simulator (Recio et al., 2012). The results in this dissertation are compared against these previous studies in Section 10.7.

Although these common platforms are useful, other platforms are used as well. Bom et al. (2013) used a custom simulator to train Ms. Pac-Man using Q-Learning on neural networks. Subramanian et al. (2011) used a Reinforcement Learning approach that automatically learned temporally extended actions called options (Sutton et al., 1999) based on game recordings of human subjects. This particular work made use of a Pac-Man simulator developed primarily for teaching undergraduate AI (DeNero and Klein, 2010). Though these studies are interesting, it is difficult to compare these results with those obtained using the Ms. Pac-Man vs. Ghosts simulator.

Since the Ms. Pac-Man vs. Ghosts simulator is the most commonly used, it will be used as a platform to learn multimodal behavior in this dissertation. Details of how it works, and how it differs from the commercial Ms. Pac-Man game, are given in the next section.

[3] `http://www.pacman-vs-ghosts.net/`

8.2 Ms. Pac-Man Simulator

In Ms. Pac-Man, a human player controls the Ms. Pac-Man agent. In each of the four different mazes there are several pills and exactly four power pills. To progress from one level to the next, all pills and power pills in a level must be eaten by Ms. Pac-Man. Each pill eaten earns the player 10 points, and each power pill earns 50 points. In the original game, players are required to replay mazes several times and revisit old mazes, but the simulator simplifies level progression by simply going through each maze in sequence and looping back to the beginning after the last maze is completed. However, for the sake of reducing learning time, all experiments in this dissertation end evaluations when the fourth maze is cleared. Each maze has a different number of regular pills, but the total across all four mazes is 932.

In each maze there are four hostile ghosts that start the level inside a lair near the center of the maze. They come out one by one, and each one pursues Ms. Pac-Man according to its own algorithm. If a ghost comes in contact with Ms. Pac-Man, the player loses a life and all agent positions are reset to how they were at the start of the level, though previously eaten pills are not reset. Therefore, it is up to Ms. Pac-Man to avoid four aggressive predators to survive. However, if Ms. Pac-Man eats one of the aforementioned power pills, then for a limited duration the game dynamics are reversed. All ghosts are colored dark blue to indicate that they are vulnerable to being eaten by Ms. Pac-Man. The first ghost eaten is worth 200 points, the second 400 points, the third 800 points, and the fourth is worth 1600 points. However, if another power pill is eaten while the ghosts are vulnerable, the score multiplier is reset so that the next ghost eaten is worth just 200 points. Therefore, the maximum score is achieved by eating all four ghosts after eating each power pill of each level. This goal is hard to accomplish, and it becomes more difficult to accomplish with each subsequent level, because the time that ghosts stay vulnerable decreases as the level increases.

Counting all pills, power pills, and edible ghosts, the highest score that can be achieved across four levels is 58,120. Ms. Pac-Man normally starts the game with three lives, and gains a fourth when 10,000 points are reached. However, some of the experiments in this dissertation only allow Ms. Pac-Man to have a single life, both to reduce evaluation time and to see how the increased risk affects the evolution of multimodal behavior.

There are many ways in which the simulator differs from the original Ms. Pac-Man. In the original game, the speed with which the ghosts move is affected by various factors, such as the number of pills remaining in the level, and whether or not the ghost is currently traversing one of the tunnels on the edge of the maze that wraps around to the other side. Ms. Pac-Man's speed is also variable in the original game, slowing down when pills are being eaten, and speeding up when turning corners. The simulator simplifies movement by having all agents move at the same speed under nearly all circumstances. The one exception is that edible ghosts move at half speed, which is both in line with the original game, and necessary for Ms. Pac-Man to have a chance at catching them.

Another significant change is the behavior of the aforementioned `Legacy` team of ghosts.

This team approximates the behavior of ghosts in the original game, but does so only very roughly. The original ghosts were thoroughly analyzed and modelled by Foderaro et al. (2012). With every movement a ghost makes, it heads along the shortest path towards a specific point defined relative to Ms. Pac-Man and/or other features of the environment. However, the way that this point is defined is different for each ghost, so that each ghost approaches Ms. Pac-Man in a slightly different manner. One ghost heads directly to Ms. Pac-Man, one heads to a point directly in front of Ms. Pac-Man, one heads to a point on the opposite side of Ms. Pac-Man relative to the position of another ghost, and one alternates between heading directly to Ms. Pac-Man and heading to the bottom left corner of the maze.

The simulator's `Legacy` team achieves a similar diversity of behaviors by using different path metrics for each ghost instead of having each head towards a different point. Specifically, the red ghost (Blinky) pursues Ms. Pac-Man directly along the shortest available path, the blue ghost (Inky) pursues Ms. Pac-Man along the path with the shortest Manhattan distance, the pink ghost (Pinky) pursues Ms. Pac-Man along the path with the shortest Euclidean distance, and the orange ghost (Sue) makes uniformly random movement choices. Sue's movements are one source of non-determinism in the game. The other source applies to all ghosts. Normally, ghosts cannot reverse their current direction of motion; they only make movement choices at junctions. However, on every time step there is a 0.15% chance that all ghosts will randomly reverse their current direction of movement. Such reversals also occur whenever a power pill is eaten. However, a random reversal is a rare and unpredictable event that can either help or harm Ms. Pac-Man.

One final quirk of the simulator is that if a power pill is eaten when ghosts are in the lair, then those ghosts are not made edible, and can emerge as threats. This small difference introduces an additional challenge: Ms. Pac-Man must be careful to avoid eating power pills when ghosts are in the lair, since this action means fewer points are able to be obtained from eating ghosts.

The simulator's variation on Ms. Pac-Man is quite challenging, and has already proven itself worthwhile as a benchmark, as noted in Section 8.1. The simulator also has an advantage over the actual Ms. Pac-Man game in that the code is completely accessible, and no screen capture is needed. These are the reasons that this particular Ms. Pac-Man simulator was used to carry out the experiments described in Chapters 9 and 10. These experiments focus on multimodal behavior, as described in the next section.

8.3 Multimodal Behavior in Ms. Pac-Man

The most obvious reason that Ms. Pac-Man requires multimodal behavior to succeed is that she must respond differently to edible and threat ghosts. Although this distinction alone would be enough to require multimodal behavior, the game is actually more complicated than that. Usually, all four ghosts are either threats or edible, but there are a few cases in which ghosts of both types are present at the same time. Edible ghosts that are eaten return to the lair for a short time before reemerging

as threats, which can potentially happen before the edible time has expired for the other ghosts. Ghosts that are in the lair when a power pill is eaten also emerge as threats before the edible time has expired.

Assuming that the appropriate task division for this game distinguishes between situations when ghosts are edible or threatening means that this game has blended tasks. A learned policy must therefore not only have behaviors for these distinct situations, but also establish behavior for the ambiguous situations in between.

In order to understand what aspects of the domain make the evolution of multimodal behavior challenging, experiments in a less-complicated version of Ms. Pac-Man are discussed first (Chapter 9), followed by experiments in the full game (Chapter 10). This less-complicated version of Ms. Pac-Man is called Imprison Ms. Pac-Man, because ghosts that end up in the lair are confined there as long as there are any edible ghosts outside of the lair. This slight modification to the game makes the tasks of dealing with edible and threat ghosts interleaved rather than blended.

More specifically, this domain is interleaved because it has a low thrashing rate between tasks: 0.01, as mentioned in Section 4.3. The calculation of this rate can be explained now that the rules of the game have been presented. Because there are four power pills in each maze, the maximum number of potential task switches per maze is eight: one switch whenever a power pill is eaten, and one when the edible time expires. Fewer switches are possible if power pills are eaten while ghosts are still edible, or if Ms. Pac-Man dies before eating all power pills, but the maximum number of switches per maze is eight. Because the edible time is (in the first maze) 200 time steps, Ms. Pac-Man will have to spend at least 800 time steps in the edible task to maximize the number of task switches. Eight task switches divided by 800 time steps results in a thrashing rate of 0.01. This estimate is conservative, however, because clearing each maze in just over 800 time steps is highly unlikely. Even a skilled agent is likely to spend at least 3,000 time steps in each maze, and could spend even more, which means the actual thrashing rate is much lower.

Because of the domain's interleaved nature, learning multimodal behavior should mostly depend on recognizing this clear task division, and developing modules to correspond to each task. For added pressure to develop effective behavior, Ms. Pac-Man only has a single life in evaluations in this domain.

The next domain in which multimodal behavior will be evolved is the full Ms. Pac-Man game. In this version of the game, eaten ghosts leave the lair once their lair time expires, making it possible for both threat and edible ghosts to be active at the same time. In the first set of experiments in this domain, Ms. Pac-Man only has a single life, so this version is called the One Life (OL) variant. Because evaluations are noisy, and a single mistake can be fatal, it is more difficult to evolve good behavior here than in the original commercial game.

Experiments are also conducted in a Multiple Lives (ML) variant in which Ms. Pac-Man starts with three lives and gains a fourth at 10,000 points. Evaluations are still noisy, but getting an occasional extra chance after a mistake allows evolving controllers to show off what they are

good at without being excessively punished for a bit of bad luck. As a result, evaluations are more consistent, and good behavior is easier to evolve than in OL Ms. Pac-Man.

It is worth noting that in all versions of the game, even though a division into edible ghost and threat ghost behaviors seems obvious, the game is complicated enough that other task divisions may also have merit. Dealing with threat ghosts can actually be considered a collection of blended tasks in its own right, since Ms. Pac-Man must avoid threats, collect pills, and decide when best to eat power pills so that eating the edible ghosts will be easy. This last behavior, which is a form of luring, will prove extremely important in the experiments below: The best performing policies actually dedicate a network module almost completely to luring, which is a surprising and powerful result.

8.4 Conclusion

This chapter described the domain of Ms. Pac-Man, and has made it clear why it is an ideal domain for the study of multimodal behavior: (1) Ms. Pac-Man must treat the ghosts in distinct ways depending on whether they are edible or not, (2) even when all ghosts are threats, potentially distinct behaviors such as pill eating and luring may be needed, (3) there is a large body of previous work to compare against because Ms. Pac-Man is a real domain that people care about, and (4) the game can be modified to have different levels of task division, allowing for a better understanding of how the nature of the task division affects the ability to learn multimodal behavior. These issues will be studied in the experiments in the next two chapters.

Chapter 9

Evaluation: Interleaved Tasks in Imprison Ms. Pac-Man

The purpose of the experiments in this chapter is to demonstrate how multimodal behavior evolves in a domain with interleaved tasks. Learning in such an environment should be easier than learning in a domain with blended tasks (Chapter 10), but harder than learning to solve a problem with isolated tasks (Chapter 6). Interleaved tasks create an additional challenge because an agent's actions in one task have consequences that influence performance in the next task. This chapter will also demonstrate how the choice of sensors can encourage learning of a human-specified task division by evolving agents. The chapter proceeds by describing the policy representation, sensors, and objectives used, and then provides details on what types of networks are evolved, followed by results.

9.1 Direction-Evaluating Policy

A learned policy can control Ms. Pac-Man in several different ways (see Section 8.1), regardless of what method is used to represent the function approximator that defines the policy.

A standard approach to Reinforcement Learning problems is to learn a value function that evaluates game states. By applying this function to states immediately following available actions, i.e. afterstates, a learning agent can simply choose the action that leads to the state that the value function found most appealing for the agent. This approach has been used in Ms. Pac-Man before (Lucas, 2005; Burrow and Lucas, 2009), but the outcomes of these experiments have been overshadowed by later research.

Another approach to controlling Ms. Pac-Man is to provide the agent with several state features and to require it to output either up, down, left, or right as its choice of action. Only one study in the literature used this approach (Gallagher and Ledwich, 2007). It is difficult to learn with such a policy because behaviors that depend simply on the relative positions of other entities do not

easily generalize across different directions. Even if the sensors are defined in relative terms, this approach still has the problem of having to duplicate sensors to correspond to multiple available directions.

Most Genetic Programming approaches to Pac-Man (Koza, 1992; Alhejali and Lucas, 2010, 2011) avoid this complication by allowing the evolved policy to select from high-level actions, such as "Goto Neatest Pill" and "Goto Safety" (a non-trivial scripted routine that decides on the safest route by which to avoid threat ghosts). Though these approaches have had some degree of success, they bias the kinds of behaviors that can be discovered, potentially ruling out better, less-intuitive behaviors.

Brandstetter and Ahmadi (2012) avoided using such high-level actions. Their approach is similar to evaluating afterstates for movement in each available direction, but is different in that direction-oriented sensors evaluate each available direction instead of evaluating the afterstate. For example, instead of computing the distance between Ms. Pac-Man and the nearest pill after moving left one step, there is a sensor that computes the distance from Ms. Pac-Man to the first pill that can be reached by moving left and never reversing direction. This approach is both surprisingly effective and works at a low enough level to impose little bias on learning.

This direction-oriented sensor approach is the general framework adopted in this dissertation as well. The specific sensors used are chosen to best illustrate interesting points about how sensor choices affect the learning of multimodal behavior, as will be described next.

9.2 Sensor Configurations

Of the methods previously used in Ms. Pac-Man (Section 8.1), the approaches that learned from features of the game state nearly always made a distinction between edible and threat ghosts. Any sensor defined to specifically provide information about threat ghosts, such as the distance to the nearest one, was accompanied by a similar sensor that only gave information about edible ghosts. This design choice assures that the sensors are not conflicting in the sense described in Section 3.1.1. The sensors were therefore split between the tasks of dealing with threatening and edible ghosts.

One of the purposes of this chapter is to show that the methods for developing multimodal behavior described in Chapter 3 are general, and work with different types of sensors, including conflict sensors. Therefore, agents will be evolved using both a `Conflict` sensors configuration and a `Split` sensors configuration. However, some sensors are common to both sensor configurations. These common sensors can be further divided into those that are not direction oriented, and those that are. Recall that the evolved neural networks are evaluated for each direction in which Ms. Pac-Man can potentially move. The sensors that are not direction oriented will provide the same reading for each direction checked on any given time step. These sensors are listed in Table 9.1. The directed sensors depend on the specific direction being evaluated, and are listed in Table 9.2.

Of the undirected sensors, those that measure a proportion are obviously useful, and have

Sensor Name	Description
Bias	Constant 1
Proportion Pills	Number of regular pills left in maze
Proportion Power Pills	Number of power pills left in maze
Proportion Edible Ghosts	Number of edible ghosts
Proportion Edible Time	Remaining ghost edible time
Ghosts Edible?	1 if ghosts are edible, 0 otherwise
All Threat Ghosts Present?	1 if four threats are outside the lair, 0 otherwise
Close to Power Pill?	1 if Ms. Pac-Man is within 10 steps of a power pill, 0 otherwise

Table 9.1: **Common Undirected Sensors in Ms. Pac-Man.** These sensors are shared by both the `Conflict` and `Split` sensor configurations. All sensors that measure a proportion are scaled to the range $[0, 1]$. These sensors are not direction dependent, so the same values will be returned for each potential movement direction evaluated on any given time step. Since these sensors are undirected, they can only meaningfully influence direction preference when combined with direction-oriented sensors (Table 9.2).

Sensor Name	Description
Nearest Pill Distance	Distance to nearest regular pill in given direction
Nearest Power Pill Distance	Distance to nearest power pill in given direction
Nearest Junction Distance	Distance to nearest maze junction in given direction
Max Pills in 30 Steps	Number of pills on the path in the given direction that has the most pills
Max Junctions in 30 Steps	Number of junctions on the path in the given direction that has the most junctions
Options From Next Junction	Number of junctions reachable from the next nearest junction that Ms. Pac-Man is closer to than a threat ghost

Table 9.2: **Common Directed Sensors in Ms. Pac-Man.** These sensors are shared by both the `Conflict` and `Split` sensor configurations. The maximum distance that can be sensed is 200. Higher distances, and distances to objects that are no longer present in the current maze, are simply reduced to 200. All such distance sensor values are divided by 200 so that they are confined to the range $[0, 1]$. The remaining sensors are similarly scaled to the range $[0, 1]$ in accordance with their maximum possible values. These sensors are all direction oriented, meaning that they will compute different values for each direction checked. Distance measurements are made along routes that go in the given direction without reversing, as are object counts. When combined with the undirected sensors in Table 9.1, Ms. Pac-Man can sense everything of importance except for the ghosts, which are handled differently by the `Conflict` and `Split` sensor configurations.

been used in several previous works. The "Ghosts Edible?" sensor only makes sense in the Imprison Ms. Pac-Man task because it reports that either all ghosts outside the lair are edible or none of them are. The sensor for "All Threat Ghosts Present?" helps maximize the ghost eating score since Ms. Pac-Man only has a chance at eating all four ghosts if they are outside the lair when a power pill is eaten. The "Close to Power Pill?" sensor is taken from Alhejali and Lucas (2010, 2011), and is useful because it warns Ms. Pac-Man that a power pill is about to be eaten, which helps optimize the timing of this strategically important event.

The common directed sensors were previously used by Brandstetter and Ahmadi (2012). The one exception is the "Options From Next Junction" sensor, which is useful in helping Ms. Pac-Man decide which directions are safe. Brandstetter and Ahmadi had a weaker version of this sensor that merely detected whether an upcoming junction was blocked by a threat ghost. The "Options From Next Junction" sensor also checks the junctions that are reachable from the next junction. This sensor takes the movement speeds of all agents into account, but does not perform any forward simulation. It is also worth noting that this sensor is technically a split sensor, because it is specifically aware of threat ghosts. This sensor makes it easier to learn multimodal behavior by having a behavioral mode for avoiding death. However, for multimodal behavior in this domain to be truly successful, Ms. Pac-Man must have a behavior that encourages the eating of edible ghosts, which are not detectable by any of the common sensors.

Sensor Name	Description
1st Closest Ghost Distance	Distance to closest ghost in given direction
2nd Closest Ghost Distance	Distance to 2nd closest ghost in given direction
3rd Closest Ghost Distance	Distance to 3rd closest ghost in given direction
4th Closest Ghost Distance	Distance to 4th closest ghost in given direction
1st Closest Ghost Incoming?	1 if closest ghost is moving towards Ms. Pac-Man, 0 otherwise
2nd Closest Ghost Incoming?	1 if 2nd closest ghost is moving towards Ms. Pac-Man, 0 otherwise
3rd Closest Ghost Incoming?	1 if 3rd closest ghost is moving towards Ms. Pac-Man, 0 otherwise
4th Closest Ghost Incoming?	1 if 4th closest ghost is moving towards Ms. Pac-Man, 0 otherwise
1st Closest Ghost Trapped?	1 if path to closest ghost does not include any junctions, 0 otherwise
2nd Closest Ghost Trapped?	1 if path to 2nd closest ghost does not include any junctions, 0 otherwise
3rd Closest Ghost Trapped?	1 if path to 3rd closest ghost does not include any junctions, 0 otherwise
4th Closest Ghost Trapped?	1 if path to 4th closest ghost does not include any junctions, 0 otherwise

Table 9.3: **Conflict Sensors in Ms. Pac-Man.** Again, the distance sensors are limited to a maximum of 200, and are scaled to $[0, 1]$. All of these sensors are directed, and none of them distinguish between edible and threat ghosts. To make informed decisions about how to act around ghosts, these sensor readings must somehow be combined with readings from the "Ghosts Edible?" sensor (Table 9.1).

The sensors specific to the `Conflict` configuration are listed in Table 9.3. Some previous

learning approaches settle for just having an awareness of the closest ghost (e.g. Brandstetter and Ahmadi, 2012), but when all ghosts are sensed it generally makes sense to sort them according to distance (Alhejali and Lucas, 2010, 2011), which is why this approach is used in this dissertation. An alternate approach is to sense specific ghosts regardless of distance. This approach should allow the sensors to account for the unique behaviors of each ghost, but preliminary experiments indicate that this approach is not very successful. Regardless of how they configured the ghost sensors, all previous studies used sensors unique to threat and edible ghosts, which is not done with the `Conflict` configuration in this dissertation. Ghosts are only separated into threat and edible ghosts in the `Split` sensor configuration.

The sensors unique to the `Split` configuration are derived by taking those unique to the `Conflict` configuration and splitting each one into two sensors: For each sensor in the `Conflict` configuration, there is one sensor in the `Split` configuration that works the same way with respect to threat ghosts only, and another sensor that works in the same way with respect to edible ghosts only. For example, the conflict sensor "3rd Closest Ghost Distance" is replaced in the `Split` configuration with "3rd Closest Threat Ghost Distance" and "3rd Closest Edible Ghost Distance" sensors. Despite having twice as many ghost sensors, this split approach to defining Ms. Pac-Man sensors is the norm in Ms. Pac-Man research. The `Conflict` configuration encodes the same information as the `Split` configuration using fewer sensors, but as the results will show, only modular networks are able to make effective use of the information encoded in the `Conflict` sensors.

Having thoroughly explained how the evolved Ms. Pac-Man controllers will sense their environment, it is now time to explain what the agents will try to achieve and how they will be evaluated.

9.3 Objectives and Performance

As illustrated in Chapters 6 and 7, evolving with multiple objectives encourages the population to explore the various behaviors along the tradeoff surface between objectives. However, in Ms. Pac-Man the research community has always treated the game as a single-objective problem, where the goal is to maximize the game score.

Even if all that matters in the end is the score, it is important to note that pill eating and ghost eating contribute to this score in different ways. Therefore, even though results in this dissertation will be evaluated according to the highest scoring individual in each evolved population, populations will be evolved using NSGA-II to maximize pill and ghost eating scores separately.

The `Pill Score` is simply the number of pills eaten. It is only meant to measure Ms. Pac-Man's ability to clear levels. Even though in terms of game score power pills are worth more than regular pills, and are particularly important because they enable Ms. Pac-Man to eat ghosts, this fitness function treats them simply as regular pills. Because there are 932 pills across the four

mazes and four power pills per maze, this fitness function has a maximum of 948.

The `Ghost Score` is trickier to define. As mentioned in Section 8.2, for each power pill eaten, the point value of each subsequently eaten ghost doubles. Therefore, rather than simply count the number of ghosts eaten, this objective should give higher rewards for the ghosts that are worth more points. Since the points received for each ghost doubles, so do the points associated with the `Ghost Score` objective: The first ghost is worth one point, the next two, the third four, and the fourth eight. Therefore, for each power pill eaten, it is possible to earn 15 `Ghost Score` points, which scales up to 60 points per maze, and 240 points across all four mazes.

Because evaluation in Ms. Pac-Man is noisy, each evolved neural network is evaluated 10 times. The final `Pill Score` and `Ghost Score` assigned to each evolved network are calculated by averaging these objectives across the 10 evaluations. Because 10 evaluations take a long time to carry out, especially as evolved agents get better at the game and have longer evaluations, a time limit is imposed for each maze: After 8,000 time steps without switching levels, Ms. Pac-Man is killed. This restriction discourages behaviors that stay alive a long time without making progress, such as moving in circles while the ghosts chase from behind. A limit of 8,000 is high enough that at the end of evolution, no champions are running out of time. For comparison, the Ms. Pac-Man vs. Ghosts competition only allowed agents 3,000 or 4,000 time steps per level (depending on the year), but also had a more lenient policy of advancing Ms. Pac-Man to the next level when time ran out rather than ending the evaluation.

Disregarding the manner in which power pills are treated like regular pills, the game score is simply a weighted combination of the `Pill Score` and `Ghost Score` defined above. Learning simply based on game score would throw away valuable information about how these objectives interact; using both objectives along with NSGA-II will allow the evolved neural networks to explore different areas of the tradeoff surface in order to find skilled, multimodal behavior.

9.4 Evolving Networks

The experiments described in this section are designed to show the benefits of neural network architectures with multiple modules in a domain requiring multimodal behavior. Therefore, populations with networks of the following types will be evolved: Networks with `One Module` (Control), `Two Modules`, and `Three Modules`. Networks with more than one module use preference neurons to decide which module to use on each time step. If either two or three modules happens to be the ideal number of modules for this domain, then learning to use these fixed modules should be easier than using Module Mutation to create new modules before learning how to use them.

Populations of networks that start with one module, but can add more via Module Mutation, are also evaluated. The Module Mutation variants tested are Module Mutation Random (`MM(R)`) and Module Mutation Duplicate (`MM(D)`). `MM(D)` is used in place of Module Mutation Previous (evaluated in Chapter 6) because it improves on MM(P) in ways described at the end of

Section 3.2.3. Also, the version of MM(R) used in this experiment does not include a module deletion mutation (evaluated in Chapter 6), since it was found that there is actually potential benefit from modules that are used only a small percentage of the time (Section 9.5).

Since the interleaved task structure of Imprison Ms. Pac-Man means it is always clear what the current task is, the following Multitask population will also be evolved: Each network has two modules, and uses one when the ghosts are edible, and the other at all other times.

All of the above methods will be evolved first using the Split sensor configuration, because it is the common way to define Ms. Pac-Man sensors. These experiments will show how the bias imposed by such a division of sensors results in multimodal behavior, even in One Module networks. In addition, populations will be evolved using Conflict sensors, to show how multiple network modules help learn multimodal behavior, even when sensor information is encoded more compactly.

To assure statistical significance, populations of each type are evolved 20 times for 200 generations each. The population size is always $\mu = \lambda = 50$. The following mutation parameters are common across all runs: Each network link has a 5% chance of Gaussian perturbation ($\mu = 0, \sigma = 1$), each network has a 40% chance of having a new random link added between existing neurons, and each network has a 20% chance of a new neuron being spliced along a randomly chosen link. For runs that use Module Mutation, each network has a 10% chance of the particular Module Mutation operation being applied.

These mutation rates are higher than in the earlier experiments in BREVE (Chapters 6 and 7), because unlike the networks in the BREVE experiments, these networks start with full connectivity, meaning there is a link from each input to each output. Another difference from BREVE experiments is the use of topological network crossover (Section 2.2.1), which is applied 50% of the time when new children are being created. Preliminary experiments indicated that fully connected starting populations and crossover both helped prevent populations from getting stuck in bad local optima of the search space. However, preference neurons do not start fully connected: Each one only starts with a single incoming link, as in the feature-selective approach, so that evolution can easily find different ways of choosing which module to use.

9.5 Results

This section first discusses the results from runs with Split sensors, then those with Conflict sensors. To distinguish between results of each type, subscripts will be used. For example, $\texttt{One Module}_{\texttt{Split}}$ refers to One Module runs using Split sensors, and $\texttt{MM(D)}_{\texttt{Con}}$ refers to MM(D) results using Conflict sensors. Videos of the behaviors described below are available online at http://nn.cs.utexas.edu/?imprison.

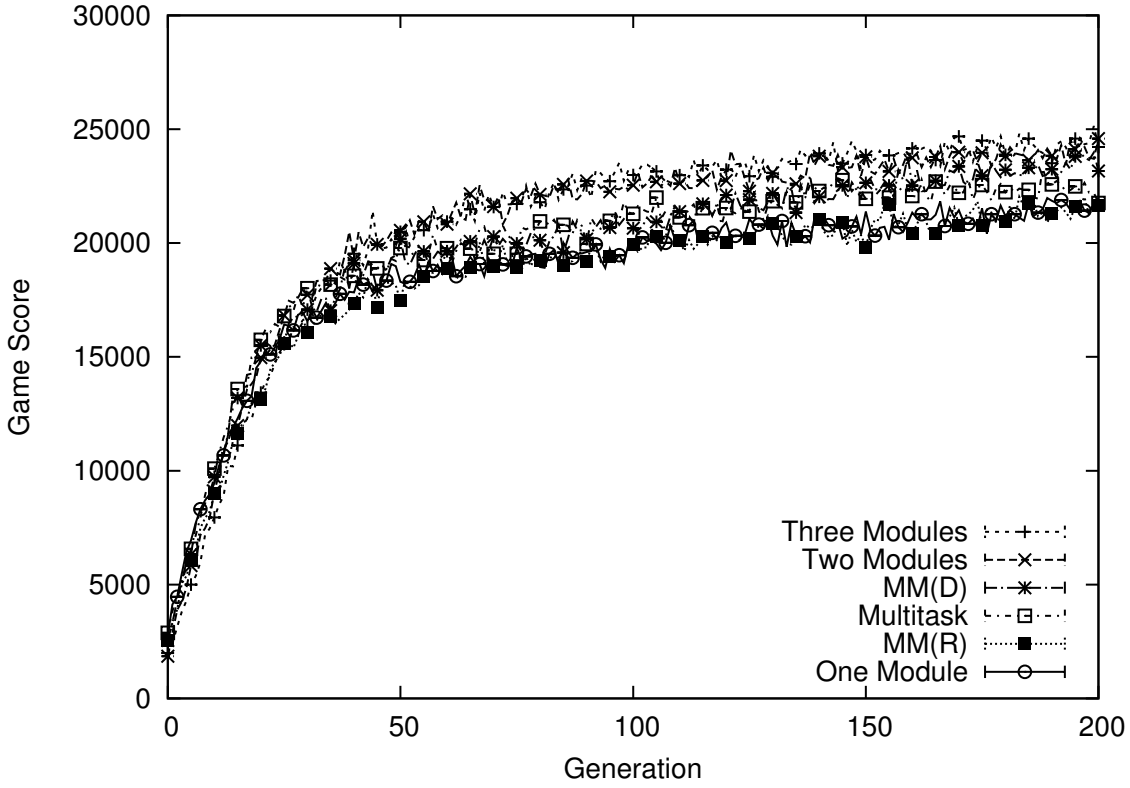
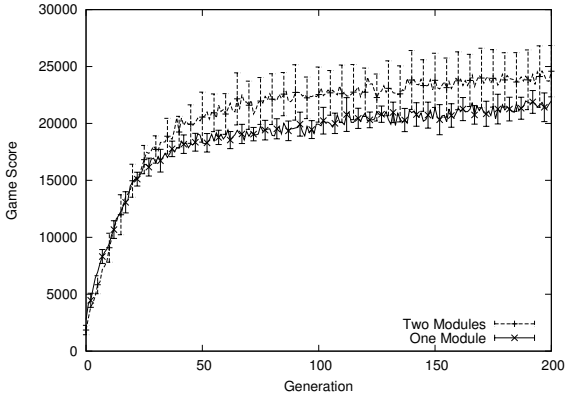


Figure 9.1: **Average Champion Game Score in Imprison Ms. Pac-Man with `Split` Sensors.** For each method using split sensors, the average champion game score across 20 runs is shown. All methods reach approximately the same level of performance. In particular, $\texttt{One Module}_{\texttt{Split}}$ reaches nearly the exact same level of performance as $\texttt{MM(R)}_{\texttt{Split}}$, and is only slightly below the other modular approaches. The small differences between methods are not statistically significant.

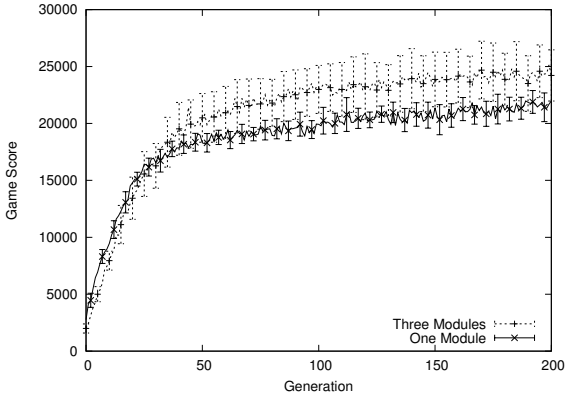
9.5.1 Split Sensors Results

There is no consistent significant difference between the different methods that make use of the `Split` sensor configuration. Figure 9.1 shows the average scores of the champions from each method across all runs. Figure 9.2 compares each multimodal approach against $\texttt{One Module}_{\texttt{Split}}$ individually with 95% confidence intervals. The confidence intervals of one method frequently contain the average of the other method, demonstrating that such differences are not significant.

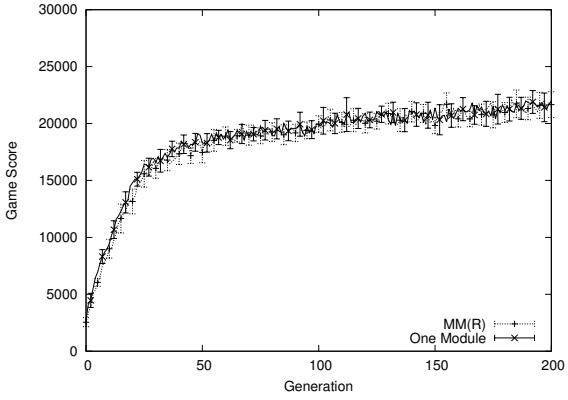
All `Split` sensor runs reach roughly the same level of performance. However, they differ in how consistently they reach that level. $\texttt{One Module}_{\texttt{Split}}$ is consistently good, but still lower than the best scores of individual runs using the other approaches. $\texttt{MM(R)}_{\texttt{Split}}$ and $\texttt{Multitask}_{\texttt{Split}}$ also *consistently* reach the same level of performance as $\texttt{One Module}_{\texttt{Split}}$, which is why all of these methods have narrow confidence intervals. They are using a task division that seems appropri-



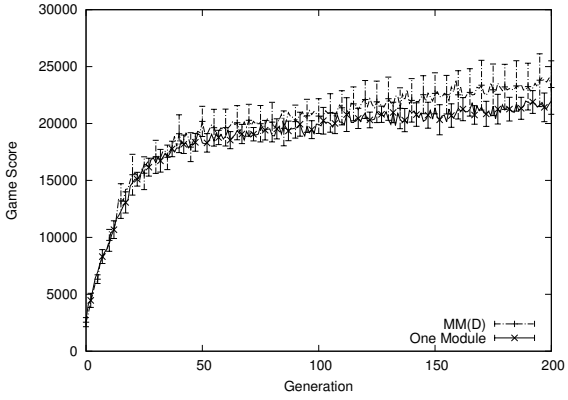
(a) One Module$_{\text{Split}}$ vs. Two Modules$_{\text{Split}}$



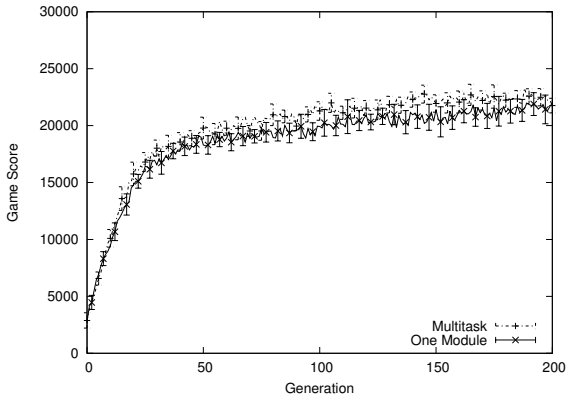
(b) One Module$_{\text{Split}}$ vs. Three Modules$_{\text{Split}}$



(c) One Module$_{\text{Split}}$ vs. MM(R)$_{\text{Split}}$



(d) One Module$_{\text{Split}}$ vs. MM(D)$_{\text{Split}}$



(e) One Module$_{\text{Split}}$ vs. Multitask$_{\text{Split}}$

Figure 9.2: (Caption on following page)

Figure 9.2: **Comparing Modular Approaches Against One Module$_{Split}$ via Average Champion Game Score in Imprison Ms. Pac-Man with Split Sensors.** Results from Figure 9.1 are shown with 95% confidence intervals comparing each modular approach against One Module$_{Split}$ individually. (a) Two Modules$_{Split}$ attains a higher average score than One Module$_{Split}$, but also has larger confidence intervals that often overlap the narrow confidence intervals of One Module$_{Split}$. (b) Three Modules$_{Split}$ scores behave similarly to Two Modules$_{Split}$. (c) MM(R)$_{Split}$ performance is the same as One Module$_{Split}$ in terms of average score and size of confidence intervals. (d) MM(D)$_{Split}$ is similar to Two Modules$_{Split}$ and Three Modules$_{Split}$. (e) Multitask$_{Split}$ performance is similar to One Module$_{Split}$ and MM(R)$_{Split}$. The higher averages and large confidence intervals of some modular approaches indicate that some of the runs score high, whereas the small confidence intervals of One Module$_{Split}$ indicate consistent, but lower, performance. Ultimately, these slight variations do not result in a significant difference between methods.

ate to the game: Multitask$_{Split}$ is forced to split the game into separate threat and edible tasks, One Module$_{Split}$ learns the same division because it uses Split sensors with a single module, and all but one MM(R)$_{Split}$ champion actually ends up using only one module, which limits their available behaviors to those that One Module$_{Split}$ can discover. However, the other methods occasionally learn a better task division.

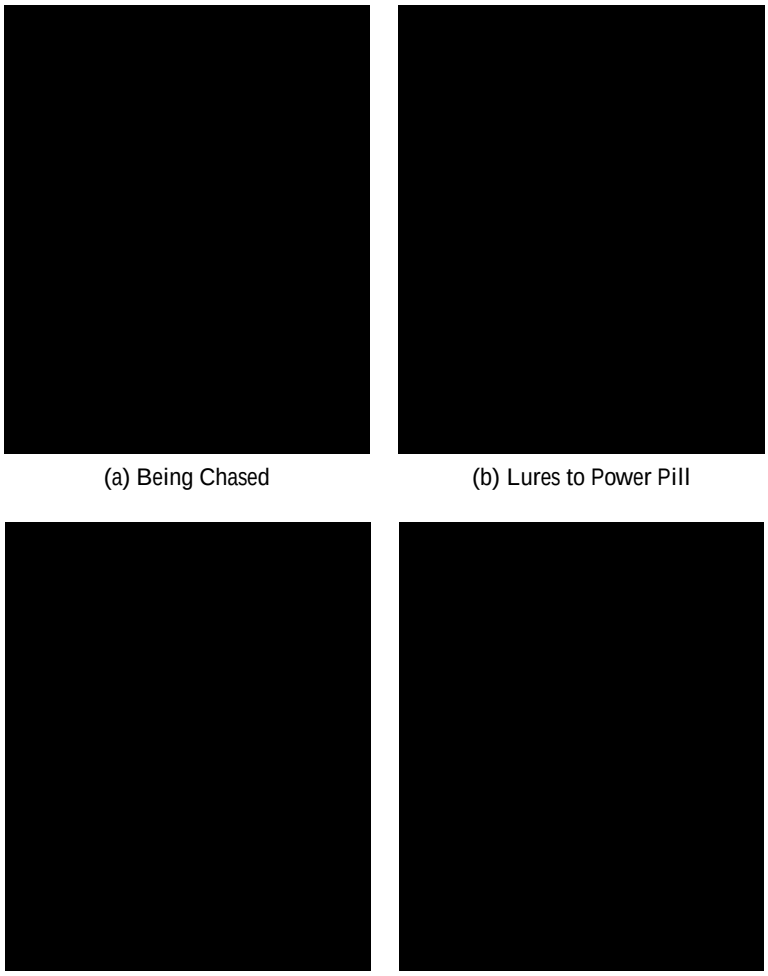
The reason that Two Modules$_{Split}$, Three Modules$_{Split}$, and MM(D)$_{Split}$ have slightly higher average scores and wider confidence intervals is that some runs of these modular approaches perform much better. The reason is that the champions produced by these runs learn a novel task division that is not directly encouraged by the Split sensors. This result can be seen using helpful visualizations while observing the behavior of each champion from each run, as demonstrated in Figure 9.3.

Observation of the high-scoring champions reveals that most share the following novel task division: There is one important but rarely used module that is primarily activated in order to force Ms. Pac-Man to eat a power pill when most of the threat ghosts are close (Figure 9.4). A second module is used in all other situations. In other words, the first module is responsible for luring the ghosts near the power pill. Since the second module handles all other behaviors, it is responsible for avoiding threat ghosts while eating pills, and for eating ghosts when they are edible. These two behavioral modes can be easily handled by a single module because the Split sensors divide the game into edible and threat tasks.

Multitask$_{Split}$ networks cannot learn a luring module because they are confined to a specific task division that, although good, is not the best available. The human-imposed module selection scheme, combined with the learning bias of the Split sensors, makes it nearly impossible to learn a policy that divides the task in any way other than along the distinction of whether or not ghosts are edible. One Module$_{Split}$ networks have the same problem, but for a different reason: The Split sensors encourage a particular task division, and no additional module is available that

(a) Threat Module (b) Edible Module

Figure 9.3: **Module Usage Visualization Example With $\texttt{Multitask}_{\texttt{Split}}$ Network.** Visualization capabilities of the simulator were used to track how modules were used by the evolved agents. These capabilities are best demonstrated using $\texttt{Multitask}_{\texttt{Split}}$ networks, which have a known task division. (a) The threat module is used at the start of an evaluation because ghosts start in threat mode. The green trail leading to Ms. Pac-Man indicates that she was using the threat module in each of those locations on the way to her current location. As time passes, cells marked in this way slowly fade, indicating that more time has passed since the threat module was used in locations with only faint coloring. (b) Later in the same evaluation, Ms. Pac-Man eats a power pill, which causes the $\texttt{Multitask}_{\texttt{Split}}$ network to switch to its edible module. The cyan path starts where the power pill was eaten, and leads to Ms. Pac-Man's current location. As with the threat module, the intensity and location of each colored cell indicate when and where Ms. Pac-Man was using a particular module. This $\texttt{Multitask}_{\texttt{Split}}$ example confirms that the visualizations can be used to understand module usage patterns. These visualizations are the basis for claims made in this dissertation about learned module usage patterns in Ms. Pac-Man.



(a) Being Chased (b) Lures to Power Pill

(c) Eats Lured Ghosts (d) Finishes Remaining Ghost

Figure 9.4: **Luring Behavior of a** Two $\texttt{Module}_{\texttt{Split}}$ **Network.** This sequence of screen shots shows how the luring module is used by the modular networks that develop it. (a) Ms. Pac-Man is being chased by three ghosts, but is not yet using the luring module. (b) Ms. Pac-Man starts using the luring module at the junction, and goes directly to the power pill when the ghosts are close enough. The cells in which the luring module was used are marked green. (c) Ms. Pac-Man loops around the corner and quickly eats the three ghosts that were close by, and now pursues the last ghost by using the wrap around tunnel. The fading green spaces show where the luring module was used; note that it was not used at any point after the power pill was eaten. (d) Ms. Pac-Man has gone through the wrap-around tunnel and is on the verge of eating the final ghost. Therefore, Ms. Pac-Man will have successfully eaten the maximum number of ghosts that can be eaten per power pill. Luring the ghosts to the power pill before eating it thus makes eating edible ghosts easier, which helps Ms. Pac-Man maximize the `Ghost Score`.

can focus on luring.

The bias imposed by the Split sensors is so strong that it also prevents MM(R)$_{Split}$ runs from developing multiple modules. Despite sometimes learning a luring module, several MM(D)$_{Split}$, Two Modules$_{Split}$, and Three Modules$_{Split}$ champions also use only a single module. The resulting behavior is not bad — Split sensors allow different behaviors for threat and edible ghosts to emerge easily — but better behavior is possible with a luring module.

Pursuing ghosts when they are edible is not good enough to get a high Ghost Score, because the edible time runs out very quickly. The ghosts cannot be caught in time if they are far away when the power pill is eaten. The luring module assures more ghosts are eaten by choosing the right time to eat power pills. This behavior works because the other module always avoids power pills, meaning that the only way to eat them is when the luring module allows it.

The luring module also fires occasionally even when there are no power pills nearby. These occurrences coincide with situations when Ms. Pac-Man is close to several of the ghosts, and near a junction. These situations are similar to situations where luring can occur: If Ms. Pac-Man is nearly surrounded by ghosts when near a junction, this module will activate, and either tell Ms. Pac-Man to move towards a nearby power pill, or tell her which junction branch to pick so as best to avoid threatening ghosts.

The specifics of how each approach makes use of its available modules are shown below: For each evolved champion, Figure 9.5 plots the percentage of the time that the most used module was chosen vs. the average game score, Figure 9.6 plots usage of the 2nd most used module vs. game score, and Figure 9.7 plots usage of the 3rd most used module vs. game score. There is a group of high-scoring champions whose most used module is chosen over 95% of the time. These champions are in the luring cluster, which consists of networks that dedicate one module entirely to luring.

There are also three Three Modules$_{Split}$ champions with scores over 23,000 that only use their primary module about 75% of the time. These champions also lure, but the module they use for it is also active while the ghosts are edible. Most modular champions that lure well use one module specifically for luring and another module for dealing with both edible and threat ghosts otherwise. However, these particular Three Modules$_{Split}$ champions use one module to deal with threats when luring and edible ghosts at all times, and another module that deals with threat ghosts, but only when not luring. Because Split sensors make it easy to behave differently in situations with threat and edible ghosts, it does not matter whether luring has its own module, or is lumped in with chasing edible ghosts. In fact, it is somewhat surprising that only these three Three Modules$_{Split}$ champions exhibit this particular module usage pattern, since Two Modules$_{Split}$ runs and Module Mutation runs could conceivably learn the same task division. Perhaps this usage pattern would appear in these other methods if more than 20 runs were conducted.

There are also many champions with middling scores (15,000–20,000) whose most used module is chosen between 55% and 85% of the time. Many of these champions dedicate one module

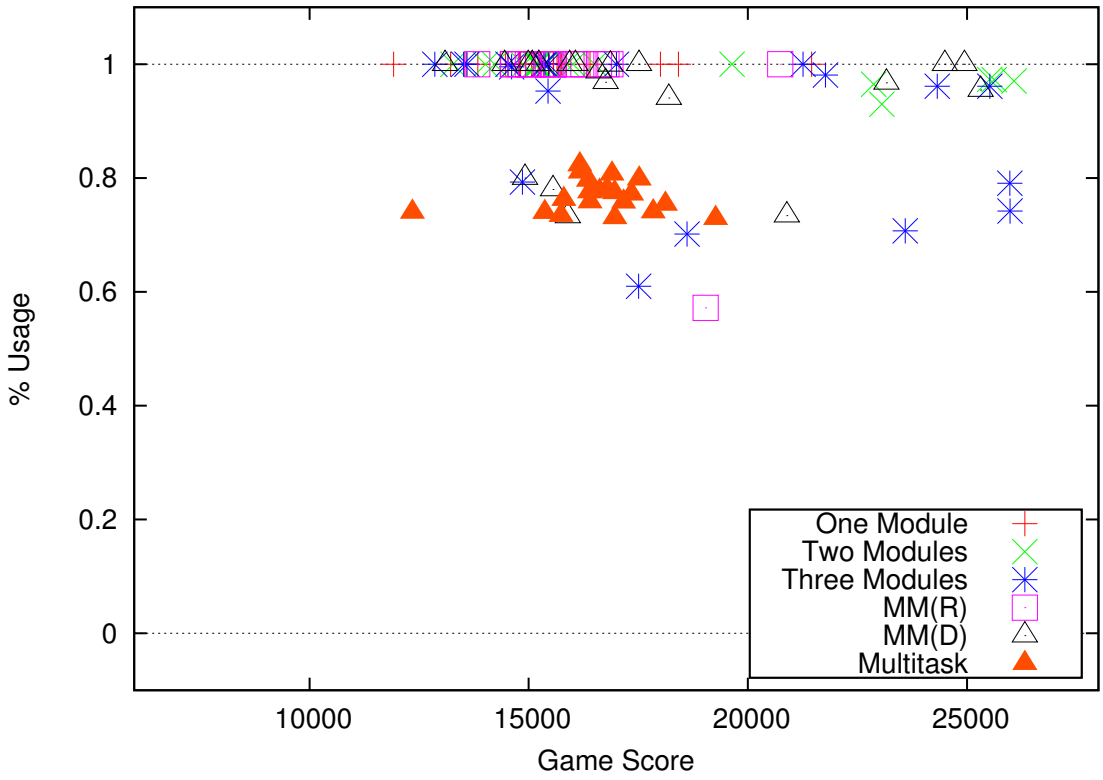


Figure 9.5: **Average Champion Game Score vs. Most Used Module Usage in Imprison Ms. Pac-Man with `Split` Sensors.** The champion of each of the 20 runs per method is evaluated 1,000 times (to minimize the effects of noise in evaluation), and the resulting average game scores are plotted against the percentage of time steps where the primary module was chosen. Recall that evolved networks process inputs for each available direction in which Ms. Pac-Man can move; the "chosen" module for a time step is the module used by the network when fed inputs for the direction in which Ms. Pac-Man ultimately chose to move on that time step. $\texttt{One Module}_{\texttt{Split}}$ champions always use their single module 100% of the time, but so do most of the modular methods. The task division imposed by the `Split` sensors makes multiple modules unnecessary to learn the obvious threat/edible split. However, some modular approaches do use multiple modules, and the best of these reach very high scores by learning luring behavior. In contrast, $\texttt{Multitask}_{\texttt{Split}}$ uses multiple modules because it is forced to, but performs no better than runs using only one module, because the human-specified task division does not allow for a luring module.

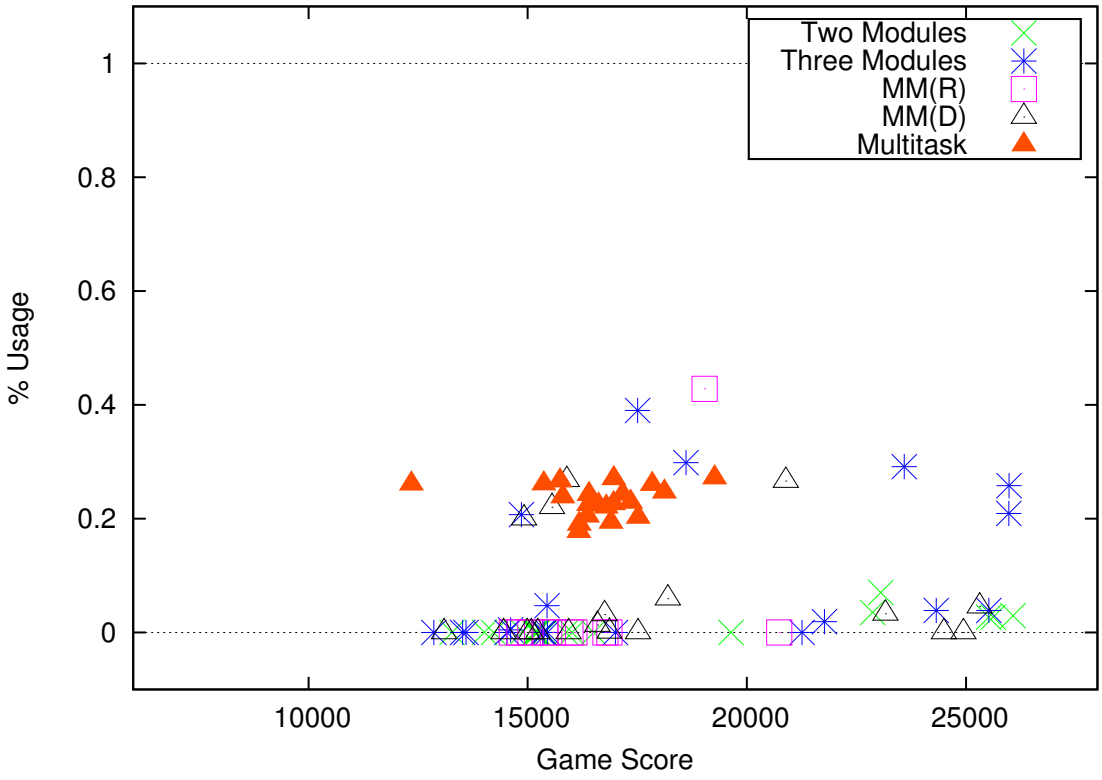


Figure 9.6: **Average Champion Game Score vs. 2nd Most Used Module Usage in Imprison Ms. Pac-Man with `Split` Sensors.** The same game scores for champions described in Figure 9.5 are shown here, but are plotted against usage of the 2nd most used module for each champion that has at least two modules. For all champions, the usage scores here are nearly a perfect mirror of the usage of the most used modules, which indicates that most champions use no more than two modules, and those that use a third barely use it at all. Champions evolved with `Split` sensors do not experience much pressure to develop multiple modules because the sensors already provide a useful task division. However, it is still possible for the modular approaches to go beyond this obvious task division and make use of multiple modules to get even higher scores, as is done by champions scoring above 22,000.

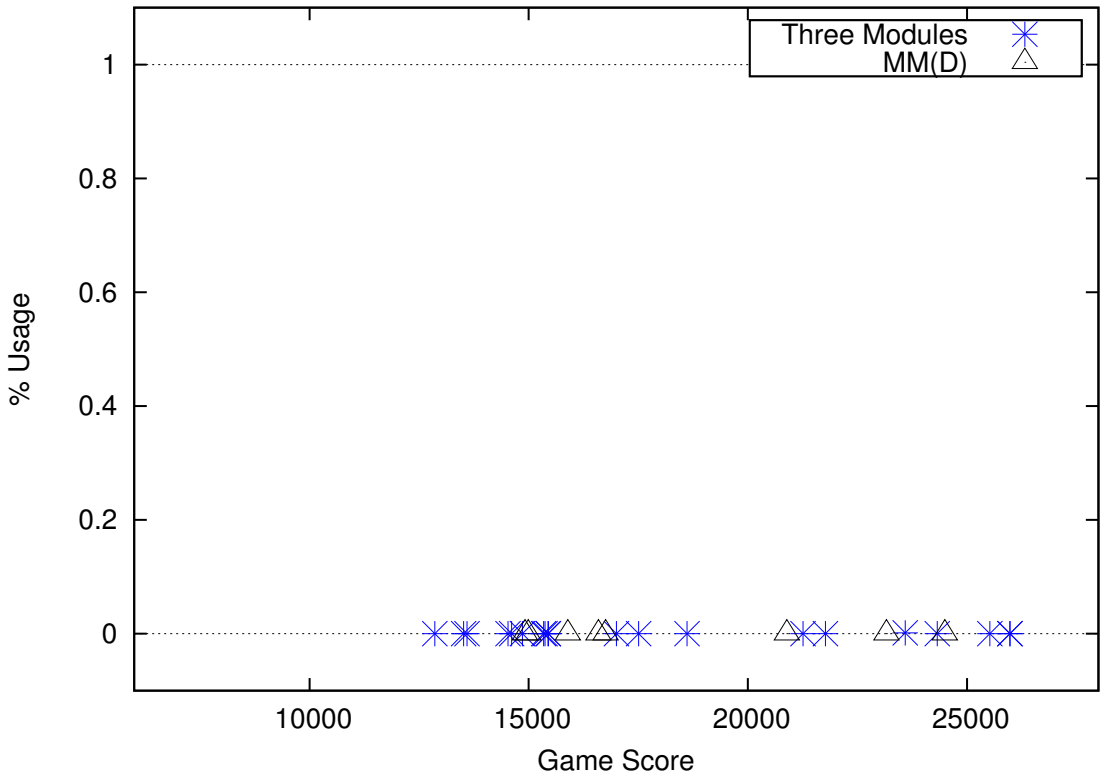


Figure 9.7: **Average Champion Game Score vs. 3rd Most Used Module Usage in Imprison Ms. Pac-Man with `Split` Sensors.** The same game scores for champions described in Figure 9.5 are shown here, but are plotted against usage of the 3rd most used module for each champion that has at least three modules. For all champions, the usage of the 3rd most used module is either zero, or so close to zero that the difference is not apparent (the highest percentage is less than 0.16%). Therefore, a third module is either not useful, or is too hard to learn with `Split` sensors in the Imprison game.

to threat ghosts, and another module to edible ghosts, despite the fact that Split sensors make this module division unnecessary. This is the threat/edible cluster (Figure 9.5). All $\texttt{Multitask}_{\texttt{Split}}$ runs are confined to this cluster because the human-specified task division is a threat/edible division. However, many modular champions in this score range are not in this cluster, because they only use one module 100% of the time. The pre-existing split at the level of sensors makes extra modules mostly superfluous, so only one module is used. The $\texttt{One Module}_{\texttt{Split}}$ runs are also in this score range, and are of course obligated to use one module 100% of the time.

Ultimately, the commonly used split approach to sensor design does not result in a significant advantage for modular networks, but some modular champions stand out because they learn a very useful task division, based on luring, that requires an additional module in order to evolve. The next section explores an uncommon, but more general, approach to sensor design that uses conflict sensors.

9.5.2 Conflict Sensors Results

When Conflict sensors are used to learn Imprison Ms. Pac-Man, $\texttt{One Module}_{\texttt{Con}}$ networks perform the worst ($p < 0.05$). The modular approaches — $\texttt{Two Modules}_{\texttt{Con}}$, $\texttt{Three Modules}_{\texttt{Con}}$, $\texttt{MM(D)}_{\texttt{Con}}$, $\texttt{MM(R)}_{\texttt{Con}}$, and $\texttt{Multitask}_{\texttt{Con}}$ — all perform significantly better. Figure 9.8 shows the average scores of the champions from each method across all runs. Comparisons with 95% confidence intervals between $\texttt{One Module}_{\texttt{Con}}$ and each individual modular method are shown in Figure 9.9.

Although the modular approaches all outperform $\texttt{One Module}_{\texttt{Con}}$, none of them are significantly different from each other. Both $\texttt{Two Modules}_{\texttt{Con}}$ and $\texttt{Three Modules}_{\texttt{Con}}$ are nearly on top of each other, which makes sense, since it turns out that all but one $\texttt{Three Modules}_{\texttt{Con}}$ champions are only using two of the three available modules. By mostly ignoring one module, the $\texttt{Three Modules}_{\texttt{Con}}$ approach produces roughly the same distribution of policies, and therefore scores, as the $\texttt{Two Modules}_{\texttt{Con}}$ approach.

Most Module Mutation champions are also settling on using just two modules, despite having the capacity to learn more. Some Module Mutation champions only have two modules, and others have more modules, but only use two. Only three $\texttt{MM(D)}_{\texttt{Con}}$ champions and two $\texttt{MM(R)}_{\texttt{Con}}$ champions actually use a third module, but observation of their behaviors and module usage patterns reveals confusing task divisions: Modules do not seem to be associated with particular behaviors or situations in any meaningful way. However, despite confusing module usage, the actual behavior of these anomalous networks is generally good; it is usually similar to that of other modular champions. Even though the overall pattern of module usage is confusing, there may be one out of the three modules that has an obvious role, such as dealing with edible ghosts. The confusing modules generally split behavior handled by a single module across two modules in a way that seems indiscriminate (Figure 9.10).

One $\texttt{MM(D)}_{\texttt{Con}}$ champion uses only one module, and its performance is on par with the

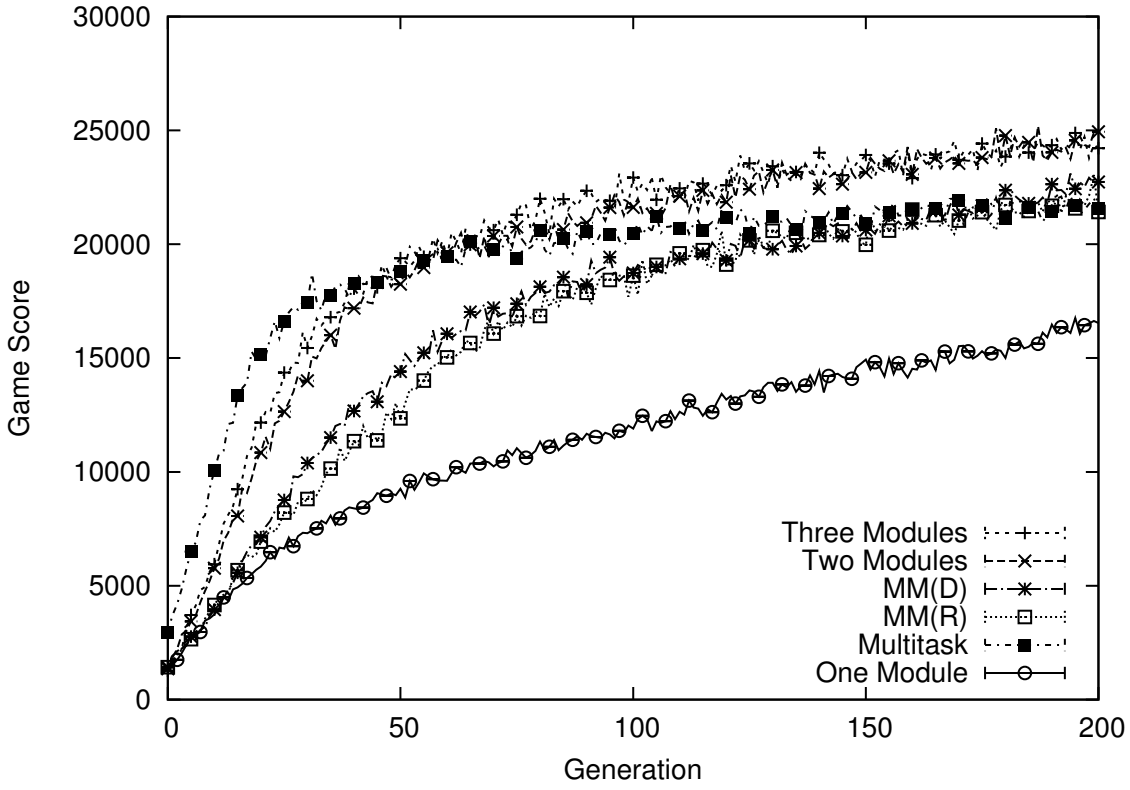
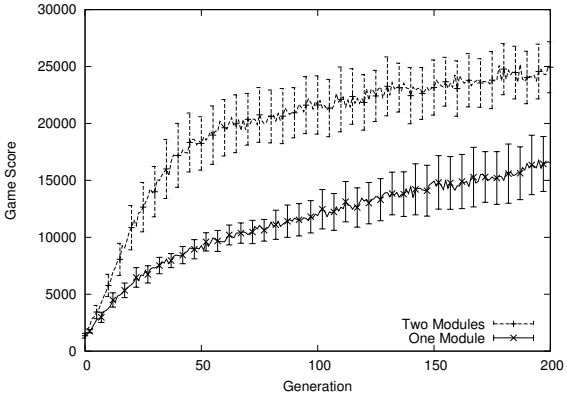
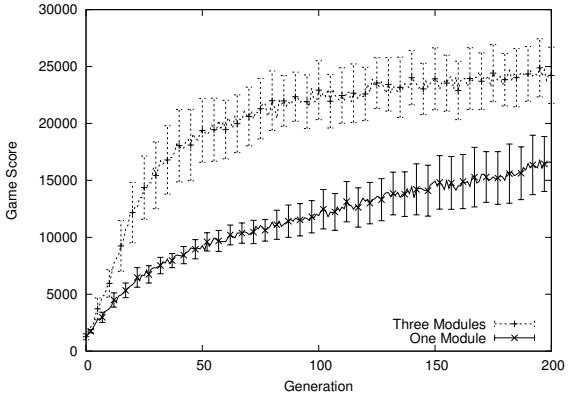


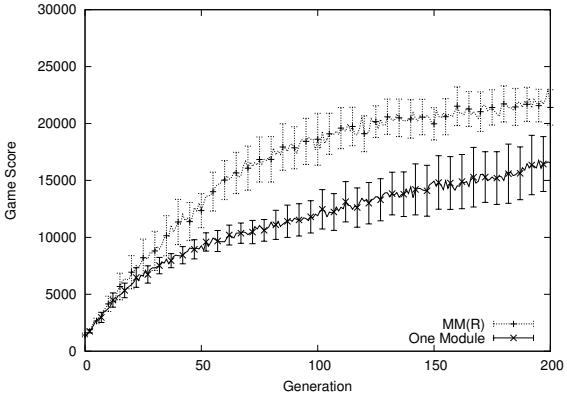
Figure 9.8: **Average Champion Game Score in Imprison Ms. Pac-Man with `Conflict` Sensors.** For each method using `Conflict` sensors, the average champion game score across 20 runs is shown. All methods that can use multiple network modules outperform `One Module`$_{\texttt{Con}}$ by a large margin. `Two Modules`$_{\texttt{Con}}$, `Three Modules`$_{\texttt{Con}}$, and `Multitask`$_{\texttt{Con}}$, which start with a fixed number of modules, all quickly earn high scores. The Module Mutation methods, `MM(R)`$_{\texttt{Con}}$ and `MM(D)`$_{\texttt{Con}}$, take slightly longer to attain high scores because they must evolve the extra modules, but the level of performance they reach is comparable to the other modular methods. Multiple network modules thus help earn higher scores in Imprison Ms. Pac-Man.



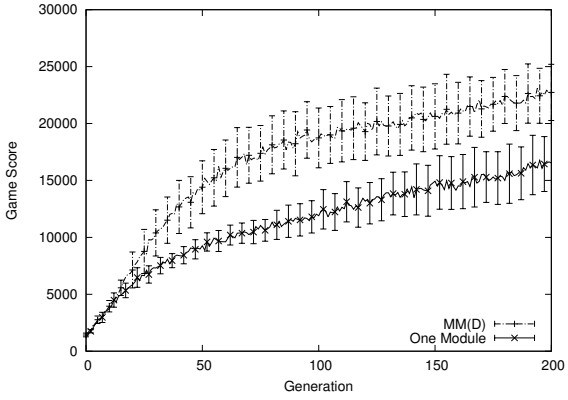
(a) One Module$_{Con}$ vs. Two Modules$_{Con}$



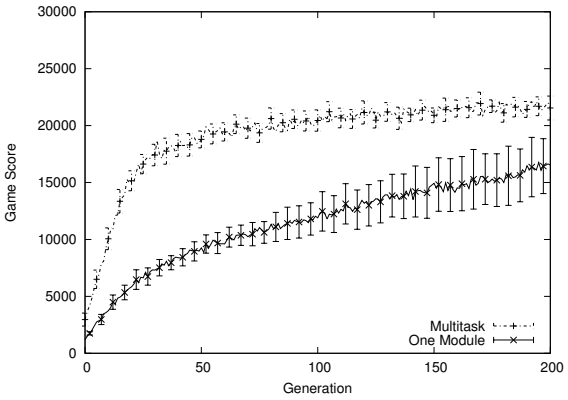
(b) One Module$_{Con}$ vs. Three Modules$_{Con}$



(c) One Module$_{Con}$ vs. MM(R)$_{Con}$



(d) One Module$_{Con}$ vs. MM(D)$_{Con}$



(e) One Module$_{Con}$ vs. Multitask$_{Con}$

Figure 9.9: (Caption on following page)

Figure 9.9: **Comparing Modular Approaches Against One Module$_{Con}$ via Average Champion Game Score in Imprison Ms. Pac-Man with Conflict Sensors.** Results from Figure 9.8 are shown with 95% confidence intervals comparing each modular approach against One Module$_{Con}$ individually. (a) Two Modules$_{Con}$ quickly establishes significantly better scores than One Module$_{Con}$. (b) Three Modules$_{Con}$ also quickly establishes significantly better scores than One Module$_{Con}$ at the same level as Two Modules$_{Con}$. (c) MM(R)$_{Con}$ takes slightly longer to break away from One Module$_{Con}$, but the level of performance reached at the end is significantly better. (d) MM(D)$_{Con}$ is similar to MM(R)$_{Con}$ in that learning is slower, but still reaches a level significantly better than One Module$_{Con}$. (e) Multitask$_{Con}$ performance increases the quickest, but it also flattens out the soonest. However, the level at which it flattens out is still significantly better than what One Module$_{Con}$ ever reaches. Though there are slight differences between the modular approaches, all of them are better than One Module$_{Con}$.

worst of the One Module$_{Con}$ runs. There is also one MM(R)$_{Con}$ champion that uses only one module, though it has a middling score.

The effort required to experiment with different numbers of modules is likely why the Module Mutation runs are less consistent. These runs do not always home in on great task divisions. As a result, Module Mutation game scores tend to rise more slowly across generations, but they still eventually reach the same general level as the other modular approaches.

Despite the few exceptions, the general convergence across methods with Conflict sensors indicates that using two modules is well-suited to the Imprison game, or at least that policies using two modules are highly attractive local optima. However, despite agreement across methods on the use of two modules, different champions do not split the game up in the same way. A good portion of runs from all methods learn the division represented by Multitask$_{Con}$: the threat/edible split.

As in the Split sensor results, the fact that all Multitask$_{Con}$ networks use the same task division explains why its confidence intervals are so narrow. However, note that although the Multitask$_{Con}$ average rises quickly, it also flattens out quickly. Other modular methods have higher averages, and wider error margins, because the better runs are discovering a luring module, as the best Split sensor results did.

The relation between usage of the most used modules and game score is depicted in Figure 9.11. Game score vs. usage of the 2nd most used modules is shown in Figure 9.12, and game score vs. usage of the 3rd most used modules is shown in Figure 9.13. There are three clusters: One Module$_{Con}$ runs with low scores (low single cluster), medium-scoring modular runs whose most used module is used between 50% and 85% of the time (threat/edible cluster), and high scoring runs that use their most used module over 95% of the time (luring cluster).

The threat/edible and luring clusters are so named because members of these clusters have modules dedicated to these behaviors, which was also the case with the similarly named clus-

(a) Module 1 (b) Module 2 (c) Module 3

Figure 9.10: **Indiscriminate Module Usage by MM(D)$_{Con}$ Network.** The behavior of this network is good (average score of about 20,120), despite having a module usage pattern that is confusing, and does not clearly associate certain modules with particular behaviors. Within the brief space of six time steps, the network alternates between the following three modules: (a) The green cells mark all spaces where Ms. Pac-Man used Module 1 on the way to her current location. The gaps along the path are positions where other modules were used. The current time step is 159. (b) On the next time step, Ms. Pac-Man switches over to Module 2 (red cells), and uses it through time step 163. This module seems to fill in all of the gaps where Module 1 was not used, but this is not the case. (c) On the next time step (time 164) Module 3 is being used (magenta cells). Notice that the magenta cells seem to be the same cells where the previous module was used, but this illusion is simply a limitation of the visualization: This MM(D)$_{Con}$ network often switches back and forth between Modules 2 and 3 so quickly that the areas colored for different, but adjacent, locations overlap. This is an example of thrashing behavior, which was also seen in the Front/Back Ramming results of Section 6.4.1. For this particular MM(D)$_{Con}$ network, this module usage pattern probably emerged because the Module Mutation operation made a perfect duplicate of an already skilled module, thus making it irrelevant which one was used at any given time. This is an example of one behavioral mode being shared across two modules. Such a result does not efficiently use the neural resources of the network, but the final behavior is still good, which is what matters the most.

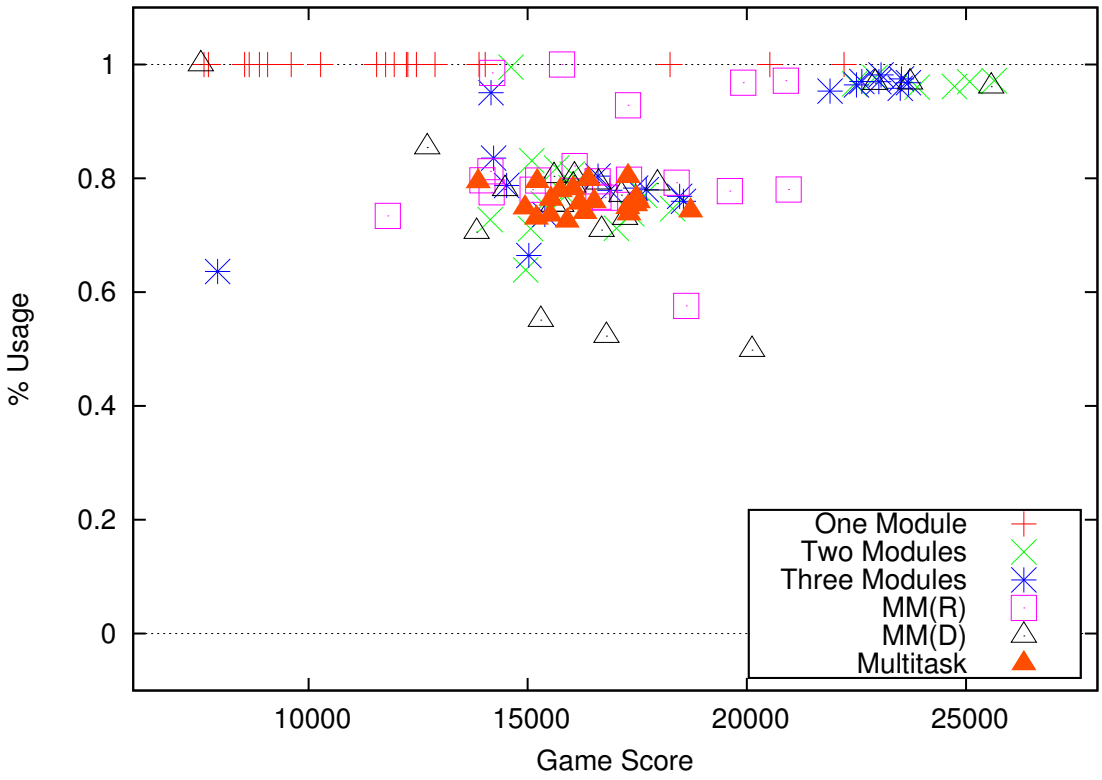


Figure 9.11: **Average Champion Game Score vs. Most Used Module Usage in Imprison Ms. Pac-Man with `Conflict` Sensors.** The champion of each of 20 runs per method is evaluated 1,000 times (to minimize the effects of noise in evaluation), and the resulting average game scores are plotted against the percentage of time steps where the most used module was chosen (the "chosen" module is defined the same way as in Figure 9.1). $\texttt{One Module}_{\texttt{Con}}$ champions always use their single module 100% of the time, and tend to have the lowest scores. Many modular champions with middling scores choose their most used module between 50% and 85% of the time. These individuals are in the threat/edible cluster. However, the best champions choose their most-used module over 95% of the time, indicating usage of a luring module.

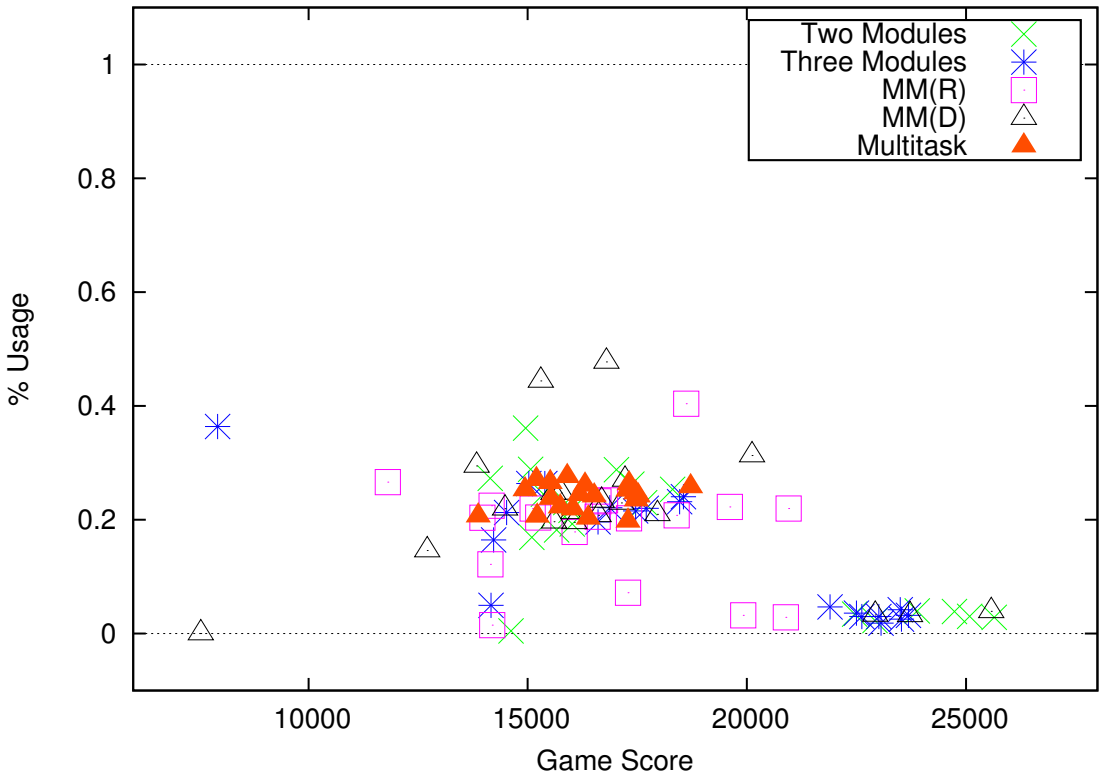


Figure 9.12: **Average Champion Game Score vs. 2nd Most Used Module Usage in Imprison Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 9.11 are shown here, but are plotted against usage of the 2nd most used module for each champion that has at least two modules. Except for a few rare exceptions, most of the usage scores here are a perfect mirror of the usage of the most used modules, which indicates that only two modules are used by most champions. Such a result is of course mandatory for all $\texttt{Two Modules}_{\texttt{Con}}$ and $\texttt{Multitask}_{\texttt{Con}}$ networks, because they only have two modules, but it is surprising that champions that could use more modules learn not to do so. The disuse of more than two modules is confirmed in Figure 9.13.

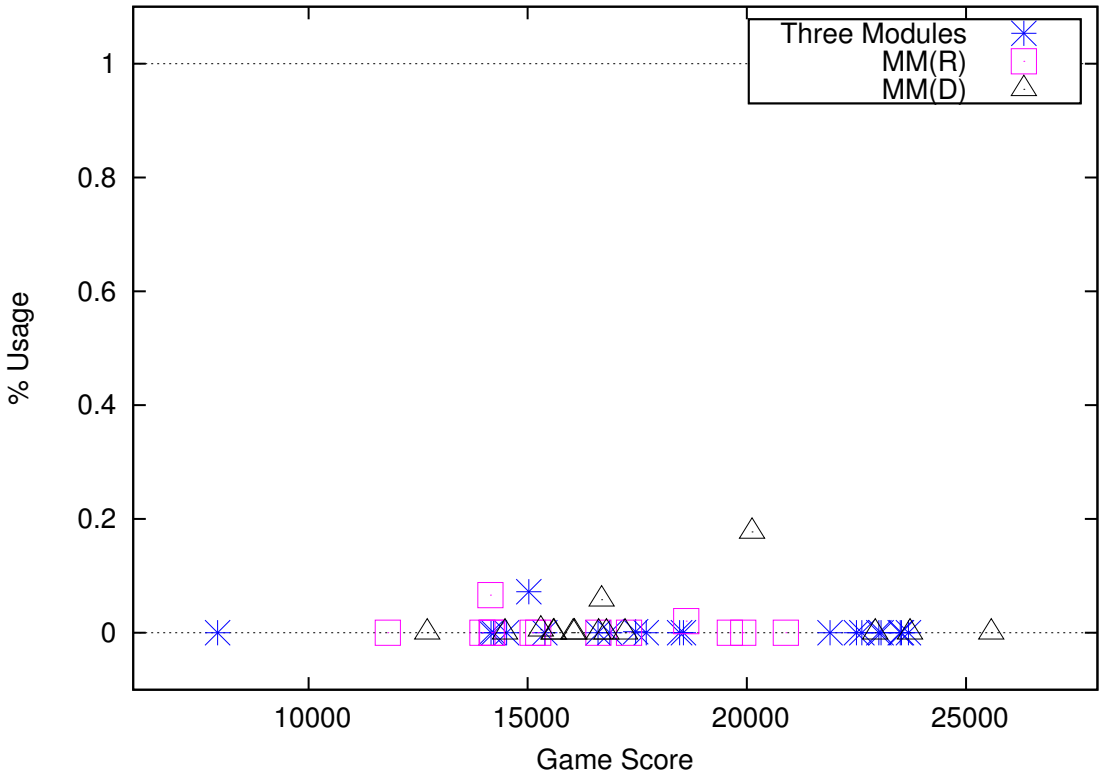


Figure 9.13: **Average Champion Game Score vs. 3rd Most Used Module Usage in Imprison Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 9.11 are shown here, but are plotted against usage of the 3rd most used module for each champion that has at least three modules. Except for a few rare exceptions, no champions use three modules. Either two modules is the ideal number for this domain, or is at least good enough that the effort to learn how to correctly use three modules is not worthwhile.

ters in the Split sensor results. All $\texttt{Multitask}_{\texttt{Con}}$ champions are in the threat/edible cluster, and the other modular methods in this cluster have learned the same threat/edible task division as $\texttt{Multitask}_{\texttt{Con}}$ (resulting in behavior similar to the $\texttt{Multitask}_{\texttt{Split}}$ behavior shown in Figure 9.3). This task division is better than using only one module (though the three $\texttt{One Module}_{\texttt{Con}}$ outliers not in the low single cluster do perform as well or better).

However, members of the luring cluster are even better. As with the best Split sensor runs, members of this cluster use a better task division that dedicates one module to luring, and another module to everything else. Though the Conflict sensors encode the game state differently, the behavior of these luring Conflict champions is essentially the same as exhibited by luring Split sensor champions in Figure 9.4. However, having one module handle "everything else" while using Conflict sensors is more impressive, since it means that one module must make Ms. Pac-Man avoid threat ghosts while eating pills, and also lead her to eat ghosts when they are edible, despite using sensors that do not distinguish between these two types of ghosts. It is difficult for these two behavioral modes to be handled by a single network module, but the few high-performing $\texttt{One Module}_{\texttt{Con}}$ outliers indicate that it is possible. However, champions in the luring cluster are even better than these $\texttt{One Module}_{\texttt{Con}}$ outliers because of their luring module.

Whether Split or Conflict sensors are used, luring behavior results in the highest scores. Modular methods using the more general Conflict sensors reach the same level of performance as methods that use Split sensors. However, $\texttt{One Module}_{\texttt{Con}}$ does not perform well because it does not have a way of splitting up the task at the level of sensors or at the level of output modules. Therefore, modular networks are successful in a situation where single module networks fail, because they can learn to utilize conflict sensors at the same level of performance as can be achieved by split sensors. Additionally, members of the luring cluster can only be generated by using modular networks. The implications of these findings are discussed in the next section.

9.6 Discussion

Having a way of splitting up the domain either at the level of sensors or at the level of output modules is advantageous in Imprison Ms. Pac-Man. Additionally, the best champions split the domain in a way that is surprising, and far from the obvious division based on threatening and edible ghosts.

This obvious division is best exemplified by using Multitask networks with either sensor configuration. $\texttt{One Module}_{\texttt{Split}}$ networks also exemplify this division. Both of these approaches quickly and reliably learn behavior that is superior to behavior learned without any kind of multimodal division (i.e. $\texttt{One Module}_{\texttt{Con}}$). However, the best way of splitting up the domain can only be learned when evolution is free to discover its own way. This division dedicates one module to luring, which is only used a very small percentage of the time, but is extremely important for maximizing game score.

However, this luring module is hard to discover, and is not found in every modular run with

preference neurons. Modular runs that do not discover it learn the obvious threat/edible split in most cases. This result means that allowing evolution to learn the domain division in Imprison Ms. Pac-Man is at least as good as a human-specified division, and sometimes better.

It is curious that champions with a luring module can handle both running from and chasing after ghosts with the other module. In fact, two high-scoring $\texttt{One Module}_{\texttt{Con}}$ outliers manage to exhibit this form of multimodal behavior, despite having only one module. The "Options From Next Junction" sensor is likely what makes this result possible. This sensor is actually not a conflict sensor, because it senses a property about junctions that specifically depends on threat ghosts. It seems that if the urge to pursue ghosts is slightly overwhelmed by the pressure to go in directions with lots of safe options, then Ms. Pac-Man will head towards ghosts when they are edible (the "Options From Next Junction" sensor will not object), and run away from them when they are threats because the influence of the "Options From Next Junction" sensor is stronger. However, such fine-tuning is not accomplished easily, which is why having distinct network modules is a more reliable way to learn these distinct behaviors.

The advantages of having a luring module, combined with the benefits of splitting the domain along the obvious threat/edible line, makes it surprising that a champion that successfully uses three modules does not emerge. With `Split` sensors, splitting edible and threat behaviors up across modules is not necessary, since the sensors accomplish this division, but such a division would conceivably be useful with `Conflict` sensors. Perhaps the "Options From Next Junction" sensor simply makes this extra division unnecessary, or more accurately, this sensor is easier to use than developing the additional modules, since the stepping stones leading to such modules may not provide as much immediate benefit.

Even Module Mutation did not discover networks that use more modules, which is surprising because module bloat was common in BREVE (Chapter 6). However, the experiments in BREVE did not use network crossover. Crossover likely reduced network bloat because newly introduced modules cannot line up structurally with components in other networks. Therefore, at most half of the children from crossover could possess such modules, and the behavior of these modules could be drastically affected by changes from crossover within the hidden layer. It is unclear whether this outcome is beneficial, detrimental, or neutral. On the one hand, bloat was reduced, which likely made search more efficient. On the other hand, the discovery of novel modules may have been more challenging due to crossover. In any case, Module Mutation did discover multimodal behavior that made use of multiple modules, and in this sense it was successful.

The results section above only considered the modules used by networks when evaluated for the direction in which Ms. Pac-Man chose to move. Although this approach led to insight into how agent behavior was split up across modules, it also hid the fact that module usage for the directions that are not chosen can have a meaningful impact on behavior. For example, a "fear" module could evolve that actively encourages Ms. Pac-Man to avoid threats. However, instead of learning to favor directions that head away from threats, the module could learn to assign a low direction preference

to directions that head towards threats. This behavior would result from the module having a high preference neuron output and a low policy neuron output (which determines direction preference) whenever there is a nearby threat in the evaluated direction. Therefore, a different module would likely be chosen for directions without threats. In this scenario, the "fear" module assures that whenever it is the chosen module for a given direction, that direction will never be chosen by Ms. Pac-Man. This module does a useful job of discouraging Ms. Pac-Man from moving towards threats, but it would never register as the "chosen" module, as defined for figures like Figure 9.5.

This hypothetical scenario indicates the types of policies that can in principle be learned with modular networks. However, even without analyzing these hidden usage patterns, interesting module usage has been uncovered: The threat/edible task division was detected, and the luring module was identified. However, because even the choices that Ms. Pac-Man does *not* make (directions not chosen) could potentially result from usage of multiple modules, it is possible that the learned behaviors are even more sophisticated than is readily apparent.

9.7 Conclusion

This chapter presented the domain of Imprison Ms. Pac-Man, a version of Ms. Pac-Man with interleaved tasks: Eaten ghosts are restricted to their lair until no edible ghosts remain. Learning in this domain provides evidence of how multimodal behavior evolves in a domain where tasks are distinct, but share a common time line.

Modular networks were compared against networks evolved with only one module using both `Split` and `Conflict` sensors. The `Split` sensors encourage a particular task division in order to bias learning in the (hopefully) right direction. The `Conflict` sensors are general, in that they do not impose a particular task division.

When using `Split` sensors, there is no distinction between modular and single-module methods, because the task is already split at the level of sensors. However, the overall best results come from modular approaches that learn a luring module, which is not directly encoded into the sensors at all.

When `Conflict` sensors were used, modular approaches outperformed traditional networks with a single module. Single-module networks have trouble behaving in different ways as the task changes. In contrast, modular methods can dedicate separate modules to each task, and like the best `Split` sensor champions, some `Conflict` sensor champions even learn a better and unexpected task division that dedicates one module to the behavior of luring ghosts, which in turn leads to high scores.

Given such success in Imprison Ms. Pac-Man, the next chapter focuses on scaling up to the full game, in which the tasks of dealing with threat and edible ghosts are blended.

Chapter 10

Evaluation: Blended Tasks in Full Ms. Pac-Man

Although the multimodal behavior in Imprison Ms. Pac-Man is impressive, it was evolved in a simplified version of the game. The remaining challenge in the full game is that both edible and threat ghosts can be present at the same time. The purpose of this chapter is to extend the methods for learning multimodal behavior to the full game, thus demonstrating that they work in a complex real-world domain with blended tasks as well.

10.1 Differences from Imprison Ms. Pac-Man

Two variants of the full game will be used for evaluation. The OL (One Life) variant only allows Ms. Pac-Man to have one life, as in Imprison Ms. Pac-Man, and in the ML (Multiple Lives) variant Ms. Pac-Man starts with three lives and gains a fourth after earning 10,000 points, just as in the original commercial game. The other difference is the per-level time limit: OL has the same 8,000 time-step limit as Imprison Ms. Pac-Man, but ML has a time limit that is so large, 30,000 time steps, that it is unlikely to ever be reached. The original game has no time limits, so the ML variant is like the original game, with a sanity check just in case an extremely unproductive agent evolves. Other than these differences, OL and ML are the same: They differ from Imprison Ms. Pac-Man in that both threat and edible ghosts can be present at the same time.

As in the original game, the amount of time that ghosts spend in the lair after being eaten is 40 time steps. This amount decreases to 90% of its previous value in each subsequent level. The time for which ghosts are edible is 200 time steps, and this time also decreases to 90% of its previous value with each level. Because the lair time is always less than the edible time, it is possible for eaten ghosts to exit the lair as threat ghosts before the remaining ghosts have switched back to being threats. This difference from Imprison Ms. Pac-Man makes the game tasks blended, which makes the game harder.

As in Imprison Ms. Pac-Man, controllers in the full game use the direction evaluating policy described in Section 9.1. In the OL variant, controllers using both `Conflict` sensors and `Split` sensors are evaluated. However, in the ML variant only `Conflict` sensors are used because these results are more interesting. As was shown in Imprison and will be shown in OL, `Split` sensors make the game easier because they are custom-designed for the domain. However, these results are not as general and interesting as those obtained with `Conflict` sensors. The specific sensors are the same as in Imprison Ms. Pac-Man (Section 9.2), but with the following differences: The "Ghosts Edible?" sensor, which is used by both the `Conflict` and `Split` sensor configurations, is changed to an "Any Ghosts Edible?" sensor. In the Imprison game, ghosts were either all threats or all edible, so this sensor provided more information, but the full-game version of this sensor only indicates if any edible ghosts remain. However, Ms. Pac-Man needs some way of knowing whether or not each individual ghost is a threat or not. Therefore, the `Conflict` sensors in the full game include one additional sensor for each of the ghosts that indicates whether or not it is edible: "1st Closest Ghost Edible?", "2nd Closest Ghost Edible?", "3rd Closest Ghost Edible?", and "4th Closest Ghost Edible?". These extra sensors, combined with the original `Conflict` sensors, provide Ms. Pac-Man with enough information to make intelligent decisions in the game. This information is hard for an evolved neural network to utilize, which is why methods for evolving modular networks are necessary.

The `Split` sensors do not need these additional sensors, because the manner in which the sensors are divided already indicates whether or not each ghost sensed is edible or a threat. Therefore, the `Split` sensors in the full game are the same as those in the Imprison variant.

Both the OL and ML variants use the same implementation for their sensors. In fact, agents evolved in the OL variant could conceivably be evaluated in the ML variant, which leads to the question of why experiments in both are being conducted. This question is answered in the next section.

10.2 One Life vs. Multiple Lives

The OL variant is useful because it is more challenging: A single mistake will end the entire evaluation. There is therefore more pressure to evolve precise policies, and the differences between methods are more clear. On the other hand, the ML variant is a reimplementation of the original game, and therefore constitutes the final real-world evaluation of the methods developed in this dissertation. It is easier than OL, which results in two interesting opportunities.

First, ML allows for the occasional mistake, and also allows the occasional sacrifice. For instance, Ms. Pac-Man may allow herself to be eaten if she is first able to eat a ghost that she would otherwise have to let escape. Such a sacrifice could actually pay off in the long run if Ms. Pac-Man has some remaining lives with which to continue evaluation. This extra flexibility could allow for more interesting multimodal behaviors to arise, or might at least make good multimodal behavior

more common.

Second, the forgiving evaluations in ML make it possible to utilize TUG (Section 3.3) in evolving complex behaviors. Note that the `Pill Score` and `Ghost Score` are intricately linked in this game: More ghosts cannot be eaten if more pills are not first eaten, in order to clear more levels and have more ghost eating chances. Turning the wrong one off in early generations can lead to being trapped in a low-scoring region of the search space. However, when TUG turns the objectives on and off correctly, Ms. Pac-Man scores can rocket upwards very quickly. Because ML allows Ms. Pac-Man to try again when a life is lost, such high scores are always achieved when modular networks are combined with TUG in the ML variant, as shall be shown in the results below.

The remaining sections in this chapter first describe experiments and results in the OL variant, and then continue with experiments and results in the ML variant. The chapter ends by comparing all of these results to those previously published in the literature.

10.3 Experimental Setup of One Life Experiments

Seven approaches were evaluated in the OL variant: Fixed networks with `One Module`, `Two Modules`, and `Three Modules`, networks with access to Module Mutation via `MM(R)` and `MM(D)`, and Multitask Learning using both two modules (`Two-Module Multitask`) and three modules (`Three-Module Multitask`). Since experiments using both `Split` and `Conflict` sensors are conducted, subscripts will once again be used to identify each approach. For example, `One Module`$_{\texttt{Split}}$ refers to `One Module` runs using `Split` sensors, and `MM(D)`$_{\texttt{Con}}$ refers to `MM(D)` results using `Conflict` sensors.

All of these approaches were evaluated in Imprison Ms. Pac-Man, except for the two Multitask Learning approaches. Because the full game has blended tasks, it is no longer obvious how to split the tasks across multitask modules. Two approaches are evaluated. `Two-Module Multitask` uses one module if any ghost is edible, and a different module at all other times. This approach is similar to the `Multitask` runs in Imprison Ms. Pac-Man, except that the module for edible ghosts will sometimes have to deal with threat ghosts. The `Three-Module Multitask` approach uses an additional module for these circumstances: one module for all threats, one module for all edible, and one module for any combination of threat and edible ghosts.

These two approaches correspond to two reasonable ways of dealing with the blended threat and edible tasks. Other divisions are possible, but incorporate increasing amounts of expert knowledge. For example, the results in Imprison Ms. Pac-Man indicate that defining a multitask division with a luring module should work well, but programming exactly when this module should be used, particularly in the more challenging full game, would not be easy.

The final difference between OL and Imprison runs is that the population size is doubled to $\mu = \lambda = 100$. Preliminary experiments indicated that this more challenging domain required larger populations in order for each method to produce consistent results. Final champion scores often

varied wildly with smaller populations.

Additional experimental parameters are identical to those in the Imprison variant: Each method is evaluated in 20 runs for 200 generations each. All networks start fully connected, except for preference neurons, which start with only a single incoming link. Mutation rates are also the same: Each network link has a 5% chance of Gaussian perturbation ($\mu = 0, \sigma = 1$), each network has a 40% chance of having a new random link added between existing neurons, and each network has a 20% chance of a new neuron being spliced along a randomly chosen link. For runs that use Module Mutation, each network has a 10% chance of the particular Module Mutation operation being applied. Crossover occurs for 50% of children produced.

Details on how agents are evaluated are also the same as in Imprison Ms. Pac-Man: Each network is evaluated 10 times, and average values of the `Pill Score` and `Ghost Score` are used by NSGA-II to select which agents reproduce and move on to the next generation. These experiments give rise to the results described next.

10.4 Results of One Life Experiments

First the results from experiments using `Split` sensors are presented, then results from `Conflict` sensor experiments. Videos of the behaviors are available online at `http://nn.cs.utexas.edu/?ol-pm`.

10.4.1 Split Sensors Results

Results in OL Ms. Pac-Man using `Split` sensors are similar to results in the Imprison variant, in that all approaches achieve close to the same level of performance. However, unlike in the Imprison game, $\texttt{Two Modules}_{\texttt{Split}}$ is significantly better than $\texttt{One Module}_{\texttt{Split}}$ ($p < 0.05$) in the final generation. None of the other modular methods are significantly different from $\texttt{One Module}_{\texttt{Split}}$, though surprisingly, some do have lower average performance: $\texttt{MM(R)}_{\texttt{Split}}$, $\texttt{Two-Module Multitask}_{\texttt{Split}}$, and $\texttt{Three-Module Multitask}_{\texttt{Split}}$.

Overall, scores are lower than they were in the Imprison game, which is to be expected since this version of the game is harder. Figure 10.1 shows the learning curves for all methods, and Figure 10.2 compares each modular method to $\texttt{One Module}_{\texttt{Split}}$ with 95% confidence intervals shown.

Plots of module usage vs. average game score are shown in Figures 10.3, 10.4, and 10.5. As in the Imprison game, the highest scoring champions use one module over 95% of the time. These individuals have discovered the same luring module that was so useful in the Imprison variant. Networks with preference neurons that do not have a luring module tend to use one module 100% of the time, because the use of `Split` sensors makes it possible to treat threat and edible ghosts differently without having separate modules dedicated to these behaviors. The only champions that

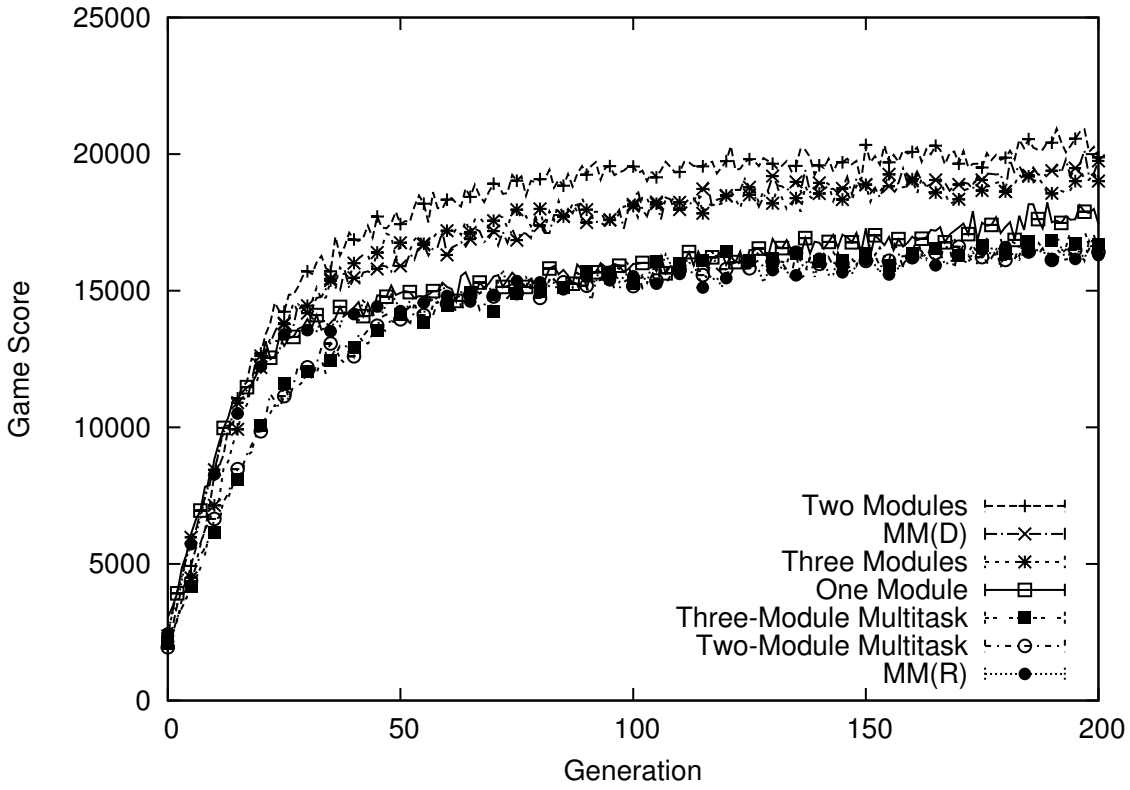
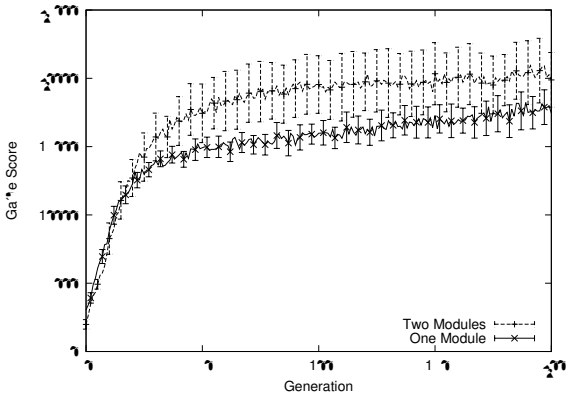
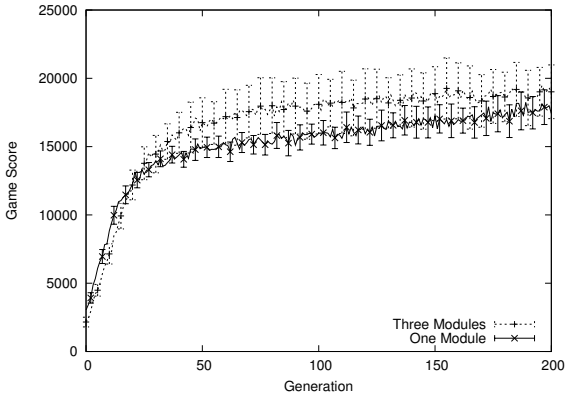


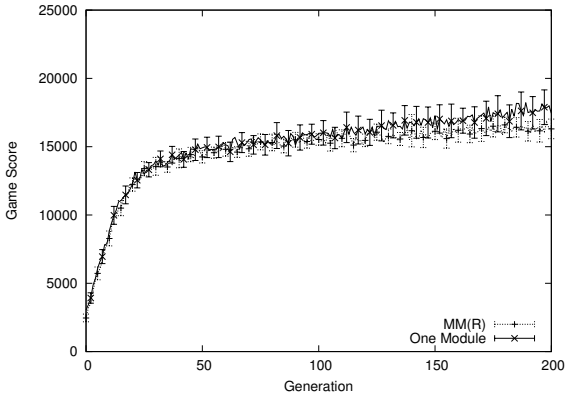
Figure 10.1: **Average Champion Game Score in One Life Ms. Pac-Man with `Split` Sensors.** For each method the average champion game score across 20 runs is shown. The `Split` sensors help $\texttt{One Module}_{\texttt{Split}}$ deal with different types of ghosts, so its scores are in the middle of those of the modular methods. The curves for $\texttt{One Module}_{\texttt{Split}}$, $\texttt{MM(R)}_{\texttt{Split}}$, $\texttt{Two-Module Multitask}_{\texttt{Split}}$, and $\texttt{Three-Module Multitask}_{\texttt{Split}}$ are on top of each other for most of evolution, though $\texttt{One Module}_{\texttt{Split}}$ pulls ahead slightly in the last few generations. However, the curves for $\texttt{MM(D)}_{\texttt{Split}}$, $\texttt{Three Modules}_{\texttt{Split}}$, and $\texttt{Two Modules}_{\texttt{Split}}$ are all consistently above $\texttt{One Module}_{\texttt{Split}}$, and $\texttt{Two Modules}_{\texttt{Split}}$ is actually significantly better up until the end, thus showing the benefit of multiple modules, even when `Split` sensors are used.



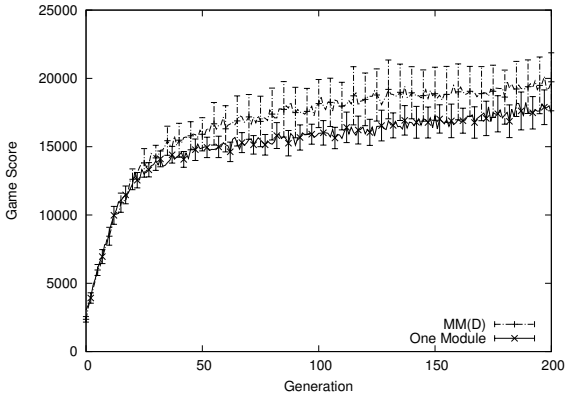
(a) One Module$_{\text{Split}}$ vs. Two Modules$_{\text{Split}}$



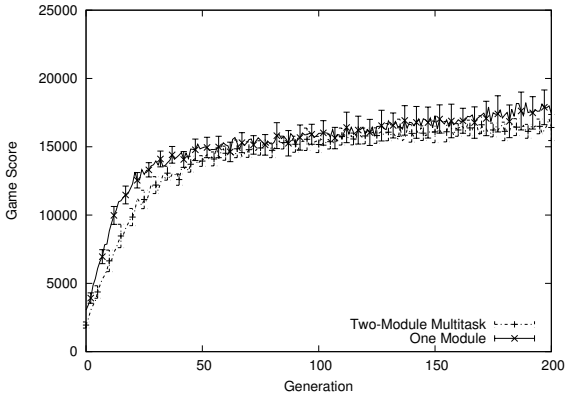
(b) One Module$_{\text{Split}}$ vs. Three Modules$_{\text{Split}}$



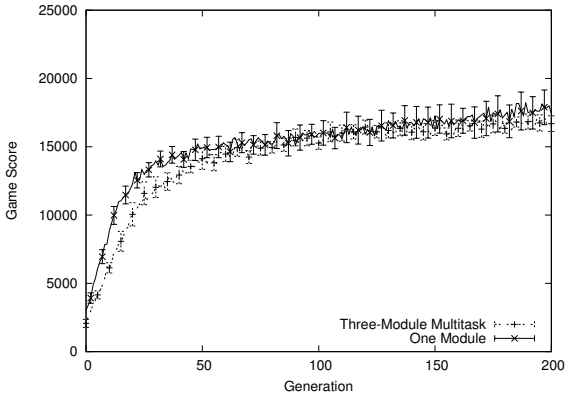
(c) One Module$_{\text{Split}}$ vs. MM(R)$_{\text{Split}}$



(d) One Module$_{\text{Split}}$ vs. MM(D)$_{\text{Split}}$



(e) One Module$_{\text{Split}}$ vs. Two-Module Multitask$_{\text{Split}}$



(f) One Module$_{\text{Split}}$ vs. Three-Module Multitask$_{\text{Split}}$

Figure 10.2: (Caption on following page)

Figure 10.2: **Comparing Modular Approaches Against One Module$_{Split}$ via Average Champion Game Score in One Life Ms. Pac-Man with Split Sensors.** Results from Figure 10.1 are shown with 95% confidence intervals comparing each modular approach against One Module$_{Split}$ individually. (a) Two Modules$_{Split}$ reaches significantly better scores than One Module$_{Split}$ early, and maintains this superiority until the end of 200 generations. (b) Three Modules$_{Split}$ also raises above One Module$_{Split}$ early, but the confidence intervals of each begin to encompass the average scores of the other in the last third of evolution. (c) MM(R)$_{Split}$ matches the performance level of One Module$_{Split}$ until the very end of evolution where One Module$_{Split}$ gains a slight advantage, but the difference is not significant. (d) In contrast, MM(D)$_{Split}$ is consistently better than One Module$_{Split}$, but as with Three Modules$_{Split}$ the difference is not significant. (e) Two-Module Multitask$_{Split}$ performance is poor: It raises slower than One Module$_{Split}$, then both curves overlap for a while, before One Module$_{Split}$ pulls slightly ahead. (f) Three-Module Multitask$_{Split}$ is no better than Two-Module Multitask$_{Split}$. An extra module for when ghosts of both types are present grants neither a benefit nor a penalty. Because the threat and edible ghost tasks are now blended, strict human-specified task divisions start to break down. The Split sensors allow One Module$_{Split}$ to recognize this task division without being forced to adhere to it as strictly as the multitask networks. However, having Split sensors and multiple modules proves to be better for MM(D)$_{Split}$ and Three Modules$_{Split}$, and significantly better for Two Modules$_{Split}$.

actually use a second module over 5% are multitask networks (both types), and these divisions result in some of the lowest scores.

It is interesting to see which of the above approaches do poorly. One Module$_{Split}$ does so because it is at a disadvantage by having only one module, but it makes decent headway using Split sensors. Three outlier runs attained scores near the lower range of scores achieved by members of the luring cluster. MM(R)$_{Split}$ performs at about the same level because it has trouble discovering new modules. Since these champions are only using one module, they get similar scores to One Module$_{Split}$. MM(D)$_{Split}$ more often discovers a useful luring module, which is why it has higher scores, but its lowest scoring champions are those that just use one module.

However, the most interesting failures are the Multitask Learning approaches. Both approaches achieve nearly identical scores consistently, and these scores are slightly under the performance level of One Module$_{Split}$. This result shows how human-specified task divisions break down in a domain with blended tasks. Even though Three-Module Multitask$_{Split}$ uses a division specifically designed to handle the blended task (Figure 10.6), it performs no differently than Two-Module Multitask$_{Split}$.

OL Ms. Pac-Man is more challenging than the Imprison game, but modular networks are still the best, because they can dedicate a module to luring, which is useful in both variants of the game. The next section presents results from similar experiments using Conflict sensors, which require networks to learn from more general information that does not bias evolution towards the

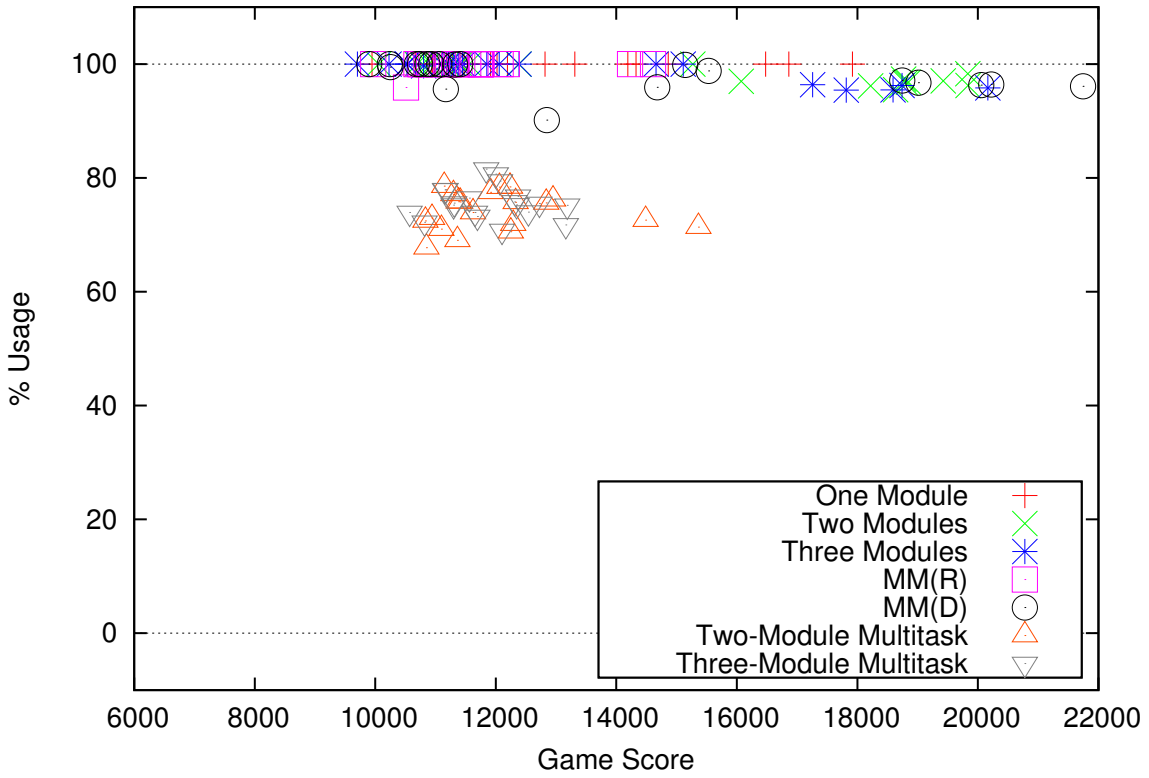


Figure 10.3: **Average Champion Game Score vs. Most Used Module Usage in One Life Ms. Pac-Man with `Split` Sensors.** The champion of each of 20 runs per method is evaluated 1,000 times (to minimize the effects of noise in evaluation), and the resulting average game scores are plotted against the percentage of time steps where the most used module was chosen (the "chosen" module is defined the same way as in the Imprison game; Figure 9.1). The highest scoring individuals (17,000–22,000) are modular networks whose most used module is chosen over 95% of the time. These champions comprise the luring cluster on the top right. There are also champions whose most used module is chosen between 65% and 85% of the time, but all of these champions use multitask networks with either a pure threat/edible division or a division between all threat, all edible, and mixed ghost types. This group is the threat/edible cluster in the lower middle area, and their scores are low (10,000–16,000). The rest of the networks use only one module all or almost all of the time, regardless of how many modules are available. This single module cluster on the top left and middle has a wide range of scores from 9,700 to 18,000. The higher scores in this range are attainable thanks to the extra information provided by `Split` sensors. However, these scores are still mostly inferior to those of the luring cluster.

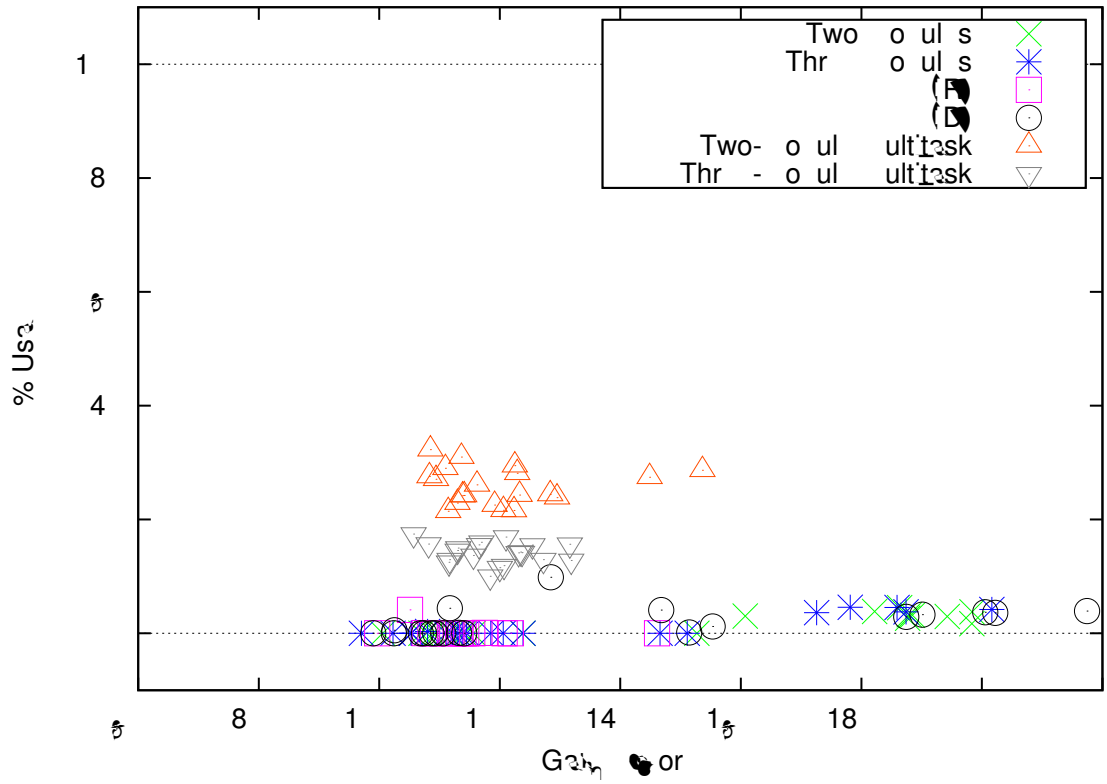


Figure 10.4: **Average Champion Game Score vs. 2nd Most Used Module Usage in One Life Ms. Pac-Man with `Split` Sensors.** The same game scores for champions described in Figure 10.3 are shown here, but are plotted against usage of the 2nd most used module for each champion that has at least two modules. Because most champions do not use more than one module, most scores are clustered near the bottom. Members of the luring cluster are barely above the 0% line (bottom right). However, this plot shows how the two multitask approaches differ. `Three-Module Multitask`$_{\texttt{Split}}$ (cluster with 5%–20% module usage) deals with edible ghosts in two different ways, so each of these modules is only used a portion of the time that any ghost is edible. However, `Two-Module Multitask`$_{\texttt{Split}}$ (cluster with 20%–37% module usage) uses one module whenever any edible ghosts are present, so its second most used module gets chosen more often than that of `Three-Module Multitask`$_{\texttt{Split}}$. The second most used module for this task division deals with cases when all ghosts are edible, since this happens slightly more often than cases where both threat and edible ghosts are present (see Figure 10.5).

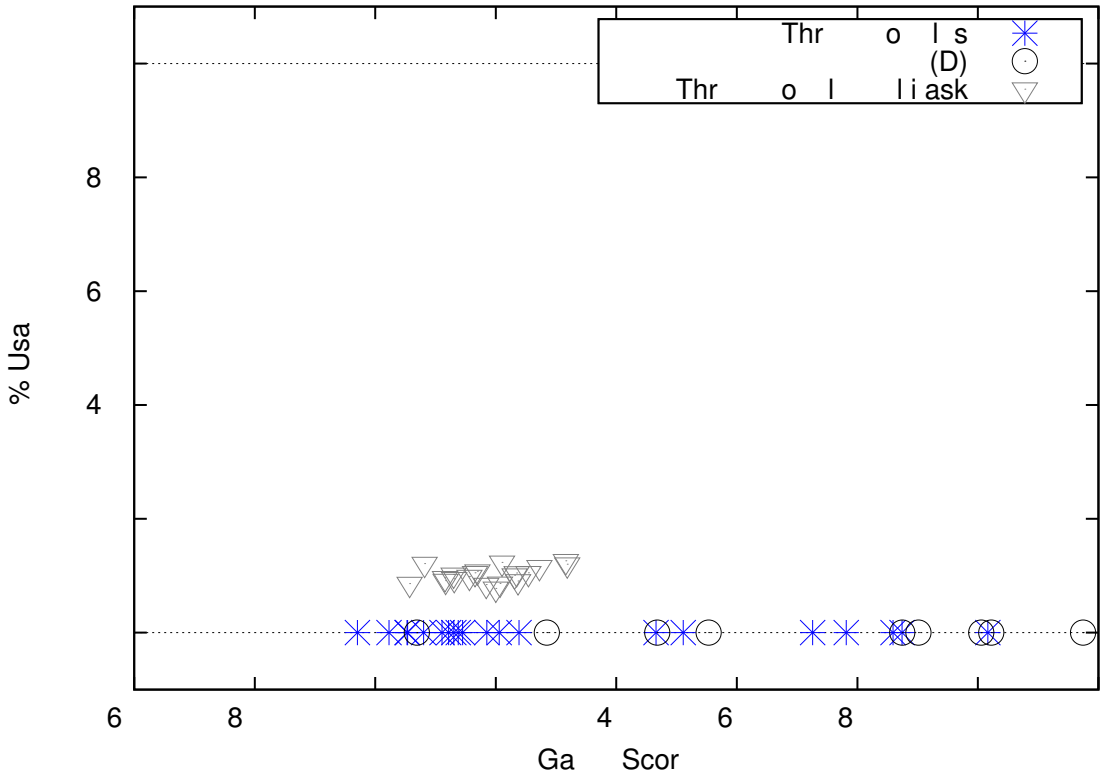


Figure 10.5: **Average Champion Game Score vs. 3rd Most Used Module Usage in One Life Ms. Pac-Man with `Split` Sensors.** The same game scores for champions described in Figure 10.3 are shown here, but are plotted against usage of the 3rd most used module for each champion that has at least three modules. Several `Three Modules`$_{\texttt{Split}}$ and `MM(D)`$_{\texttt{Split}}$ networks have a third module, but only `Three-Module Multitask`$_{\texttt{Split}}$ (top cluster in the middle) actually uses a third module, and only a small percentage of the time. Three modules are not necessary to get high scores, and forcing `Three-Module Multitask`$_{\texttt{Split}}$ to use three modules doe not result in better scores than `Two-Module Multitask`$_{\texttt{Split}}$.

(a) All Threats (b) All Edible (c) Threat and Edible

Figure 10.6: `Three-Module Multitask`$_{\texttt{Split}}$ **Task Division in One Life Ms. Pac-Man.** This agent uses a human-specified task division intended to deal with blended tasks: (a) The cyan trail shows where Ms. Pac-Man traveled while only threat ghosts were in the maze. (b) After eating a power pill, all ghosts become edible, so the network switches to the edible module (green trails). (b) After a ghost is eaten, it reemerges as a threat while other ghosts are still edible, which activates the third module (red trails). This network has a different module for each possible situation related to the threat/edible task division, including a module for when the tasks overlap, but it still achieves low scores. As in the Imprison variant, higher scores are achieved by networks that discover the unexpected luring module.

threat/edible task division.

10.4.2 Conflict Sensors Results

In the Conflict sensor experiments, several modular architectures outperform $\texttt{One Module}_{\texttt{Con}}$ networks. Specifically, $\texttt{Two Modules}_{\texttt{Con}}$, $\texttt{Three Modules}_{\texttt{Con}}$, and $\texttt{MM(D)}_{\texttt{Con}}$ are significantly better than $\texttt{One Module}_{\texttt{Con}}$ ($p < 0.05$). However, there are no significant differences between $\texttt{One Module}_{\texttt{Con}}$ and the remaining methods: $\texttt{Two-Module Multitask}_{\texttt{Con}}$, $\texttt{Three-Module Multitask}_{\texttt{Con}}$, and $\texttt{MM(R)}_{\texttt{Con}}$. Figure 10.7 shows the average scores of the champions from each method across all runs. Comparisons with 95% confidence intervals between $\texttt{One Module}_{\texttt{Con}}$ and each individual modular method are shown in Figure 10.8.

$\texttt{Two Modules}_{\texttt{Con}}$, $\texttt{Three Modules}_{\texttt{Con}}$, and $\texttt{MM(D)}_{\texttt{Con}}$ succeed in OL Ms. Pac-Man for the same reason they succeed in Imprison Ms. Pac-Man: They associate useful behavioral modes with separate network modules. There are both networks that discover a luring module and networks that learn to split the domain according to threat and edible ghosts among the champions from these runs.

The relation between module usage and game score is depicted in Figures 10.9, 10.10, and 10.11. As in the Imprison variant, there are three clusters: Low single cluster contains the runs that use one module 100% of the time, and score poorly as a result, the threat/edible cluster contains medium-scoring individuals whose most used module is chosen 60% to 90% of the time (which is higher than in the Imprison variant), and the luring cluster earns the highest scores with agents using one module over 95% of the time.

When using Split sensors, networks that did not discover a luring module only used one module. The Split sensors make it easy to have different behaviors for threat and edible ghosts, so these networks had no need to discover a threat/edible module division. Conflict sensors do sometimes discover such a division. How is this possible when ghosts of both types can be present at the same time? It turns out that in these situations, Ms. Pac-Man will generally use the edible module until threat ghosts cut off the path from her to any remaining edible ghosts, which is a sensible strategy for staying alive (Figure 10.12).

As with the Imprison game, networks using the threat/edible split are competent but not the best. These networks do not always do a great job at eating ghosts. Sometimes these networks lead Ms. Pac-Man to eat a power pill at an inopportune time. The edible time is so short that even though she immediately switches over to a module that directly pursues the ghosts, she has a hard time catching many of them before time runs out. To eat more ghosts, Ms. Pac-Man needs luring behavior, as has been demonstrated in all Ms. Pac-Man experiments so far.

As with the Split sensor experiments, neither Multitask Learning approach is particularly successful, which is not surprising given that Conflict sensors pose an even greater challenge. The behavior of the multitask networks with Conflict sensors is the same as with Split sensors, because the rigid human-specified task division makes these sensor differences irrelevant.

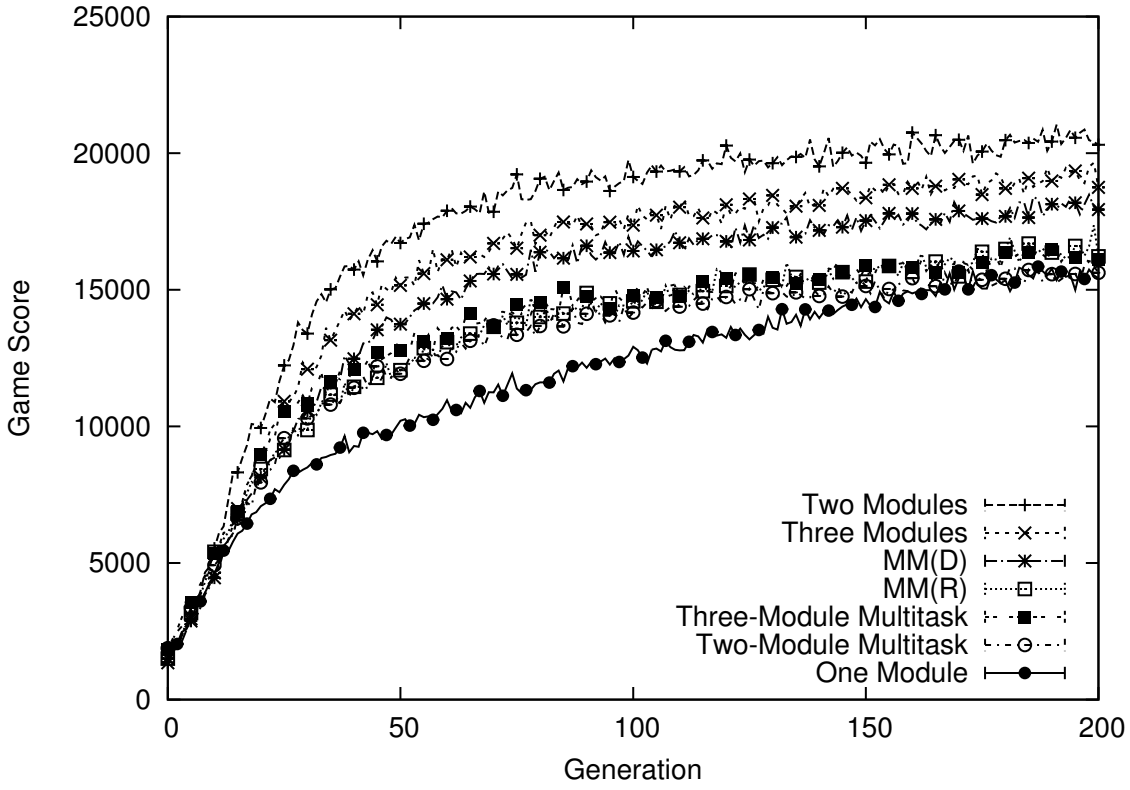
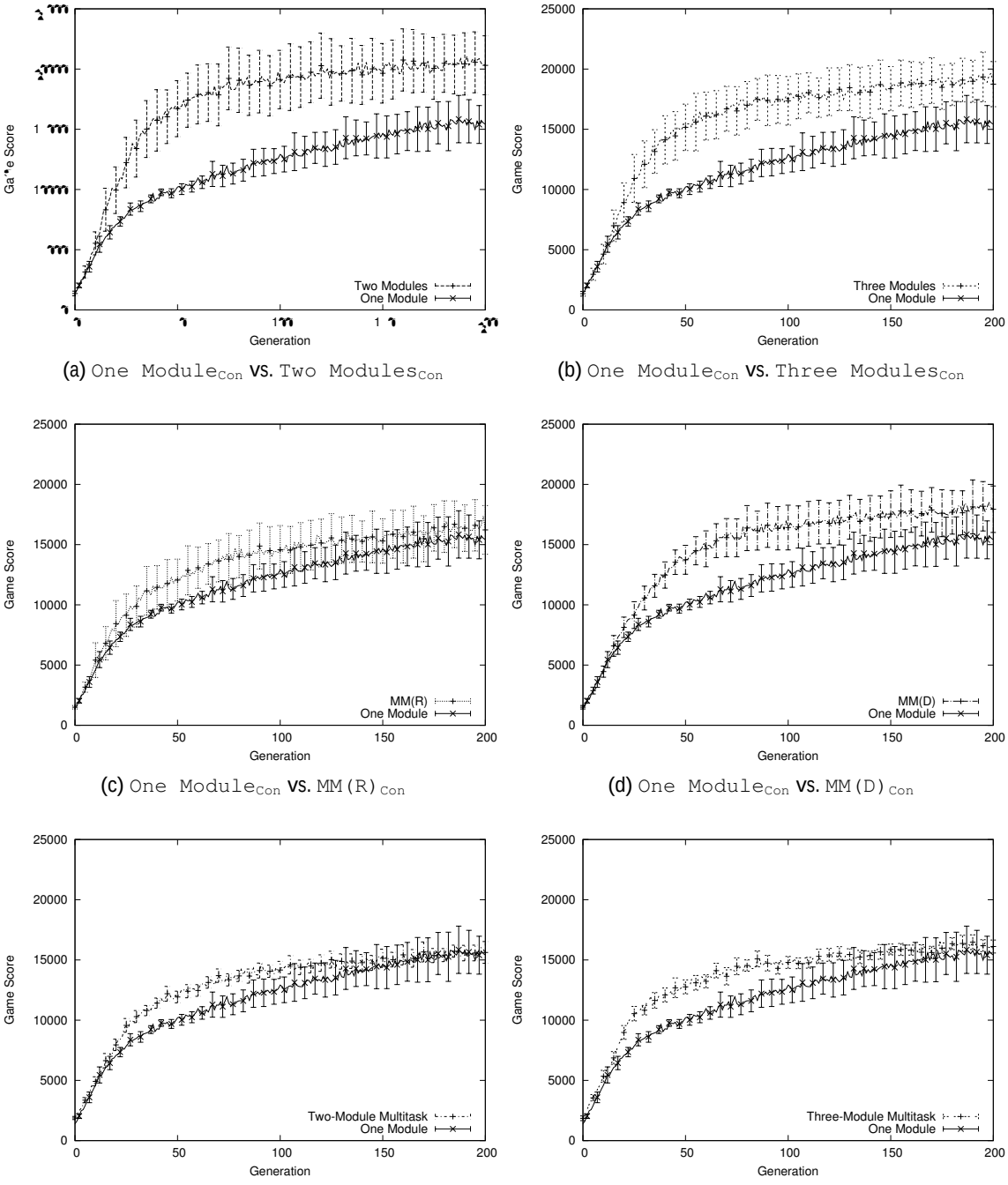


Figure 10.7: **Average Champion Game Score in One Life Ms. Pac-Man with `Conflict` Sensors.** For each method the average champion game score across 20 runs is shown. All modular approaches outperform $\texttt{One Module}_{\texttt{Con}}$ early in evolution, but it eventually catches up to $\texttt{MM(R)}_{\texttt{Con}}$, $\texttt{Two-Module Multitask}_{\texttt{Con}}$, and $\texttt{Three-Module Multitask}_{\texttt{Con}}$, which all settle at the same level of performance. In contrast, $\texttt{MM(D)}_{\texttt{Con}}$, $\texttt{Three Modules}_{\texttt{Con}}$, and $\texttt{Two Modules}_{\texttt{Con}}$ remain significantly better than $\texttt{One Module}_{\texttt{Con}}$ until the end. It is particularly impressive that fixed networks that learn their own task division are superior to fixed networks that use a human-specified task division.



(a) One Module$_{Con}$ vs. Two Modules$_{Con}$

(b) One Module$_{Con}$ vs. Three Modules$_{Con}$

(c) One Module$_{Con}$ vs. MM(R)$_{Con}$

(d) One Module$_{Con}$ vs. MM(D)$_{Con}$

(e) One Module$_{Con}$ vs. Two-Module Multitask$_{Con}$

(f) One Module$_{Con}$ vs. Three-Module Multitask$_{Con}$

Figure 10.8: (Caption on following page)

Figure 10.8: **Comparing Modular Approaches Against One Module$_{Con}$ via Average Champion Game Score in One Life Ms. Pac-Man with Conflict Sensors.** Results from Figure 10.7 are shown with 95% confidence intervals comparing each modular approach against One Module$_{Con}$ individually. (a) Two Modules$_{Con}$ quickly establishes significantly better scores than One Module$_{Con}$ that persist until the end of 200 generations. (b) Three Modules$_{Con}$ also quickly establishes significantly better scores at a level almost as good as Two Modules$_{Con}$. (c) MM(R)$_{Con}$ is not as impressive in OL as it was in Imprison, i.e. not significantly better than One Module$_{Con}$. (d) MM(D)$_{Con}$ is significantly better than One Module$_{Con}$, though by a narrow margin. It is better than MM(R)$_{Con}$, but not as good as Two Modules$_{Con}$ and Three Modules$_{Con}$. (e) Two-Module Multitask$_{Con}$ performance is consistently poor, and only reaches the same level as One Module$_{Con}$. (f) Three-Module Multitask$_{Con}$ is also not any better than One Module$_{Con}$. Both Multitask approaches reach the same level of performance consistently. The weakness of multitask's hand-designed task divisions become apparent now that the domain has blended tasks. Surprisingly, MM(R)$_{Con}$ also falters in this domain. However, MM(D)$_{Con}$, Three Modules$_{Con}$, and Two Modules$_{Con}$ show that modular networks still lead to the best scores.

The method whose performance changes the most as a result of switching to Conflict sensors is One Module$_{Con}$. Now this method has the lowest average performance, although after 200 generations it is still not significantly different from the multitask methods, or from MM(R)$_{Con}$ (which mostly uses one module as well). With neither Split sensors, nor multiple modules, it is hard to learn how to respond to the ghosts in different ways for each situation. These results demonstrate once again that having a way to split the domain into separate tasks, either via sensors or output modules, encourages skilled multimodal behavior.

However, there is still lots of variation in the performance of each method, likely due to the catastrophic effect of losing a life. Networks that make small mistakes suffer big consequences, and therefore do not get a chance to produce offspring. The next set of experiments give Ms. Pac-Man extra lives to see if a more lenient evaluation scheme gives rise to better behavior. It turns out that it does, and also that it is possible to use TUG to improve performance further.

10.5 Experimental Setup of Multiple Lives Experiments

The OL experiments confirmed that modular networks can learn intelligent task divisions even in a domain with blended tasks. However, OL is still short of the original game. Therefore, the experiments in the remainder of the chapter extend the game to multiple lives and effectively unlimited time, making it identical to the original game in nearly all important aspects (though a level cap is still maintained to impose a limit on evaluation length). In addition, it turns out that Targeting Unachieved Goals (TUG) can be used to push performance up to even higher levels in the ML variant.

Experiments in ML Ms. Pac-Man are similar to those in the OL variant. All mutation and

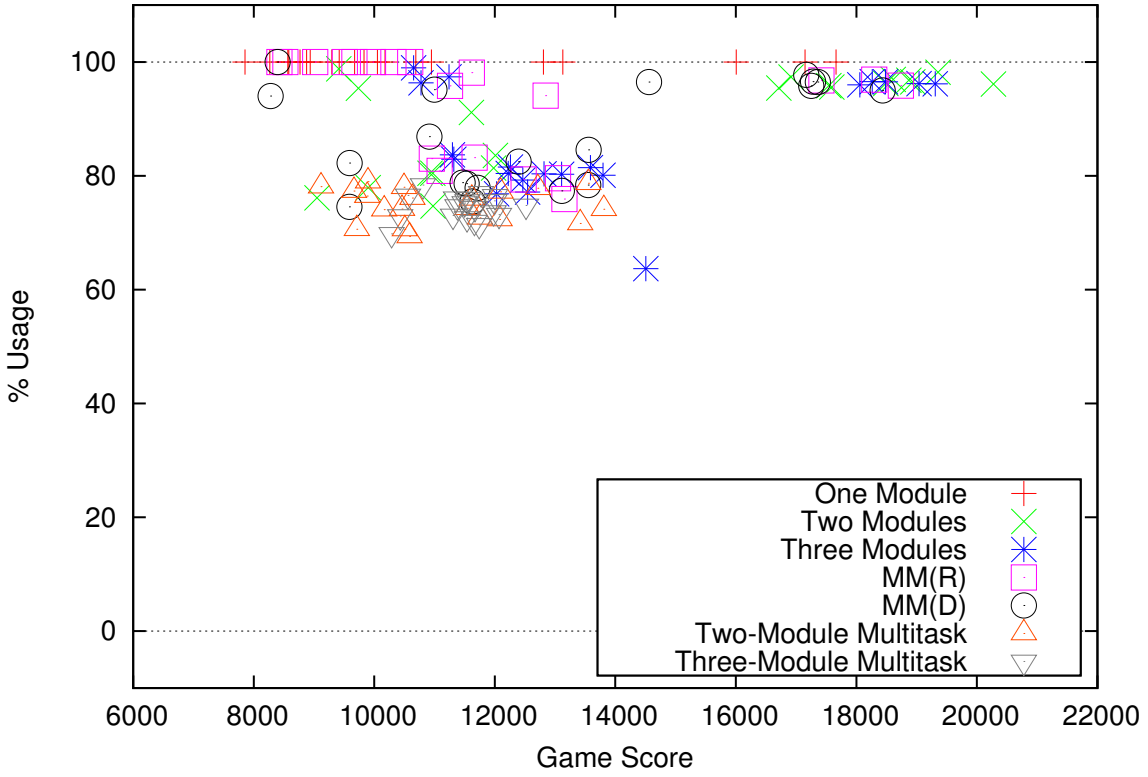


Figure 10.9: **Average Champion Game Score vs. Most Used Module Usage in One Life Ms. Pac-Man with `Conflict` Sensors.** The champion of each of 20 runs per method is evaluated 1,000 times (to minimize the effects of noise in evaluation), and the resulting average game scores are plotted against the percentage of time steps where the most used module was chosen (the "chosen" module is defined the same way as in the Imprison game; Figure 9.1). `One Module`$_{\texttt{Con}}$ champions always use their single module 100% of the time, and tend to have the lowest scores. As in the Imprison variant, modular networks with middling scores (9,000–15,000) are in the threat/edible cluster (most-used module 60%–90%; lower middle). The best scores (16,000–21,000) still come from modular networks in the luring cluster (most-used module over 95%; top right), though a few `One Module`$_{\texttt{Con}}$ outliers are on the lower edge of this score range. `Two Modules`$_{\texttt{Con}}$ has the most champions in the luring cluster, which explains its high average performance, but `Three Modules`$_{\texttt{Con}}$ and `MM(D)`$_{\texttt{Con}}$ have several representatives in this cluster as well. Even `MM(R)`$_{\texttt{Con}}$ has some champions that discover a luring module, but its average performance is brought down by the many low-scoring champions that use only one module. Both multitask approaches achieve consistently middling scores, and are also consistent in their module usage. Ultimately, being able to learn a task division is more likely to lead to the high-scoring luring behavior.

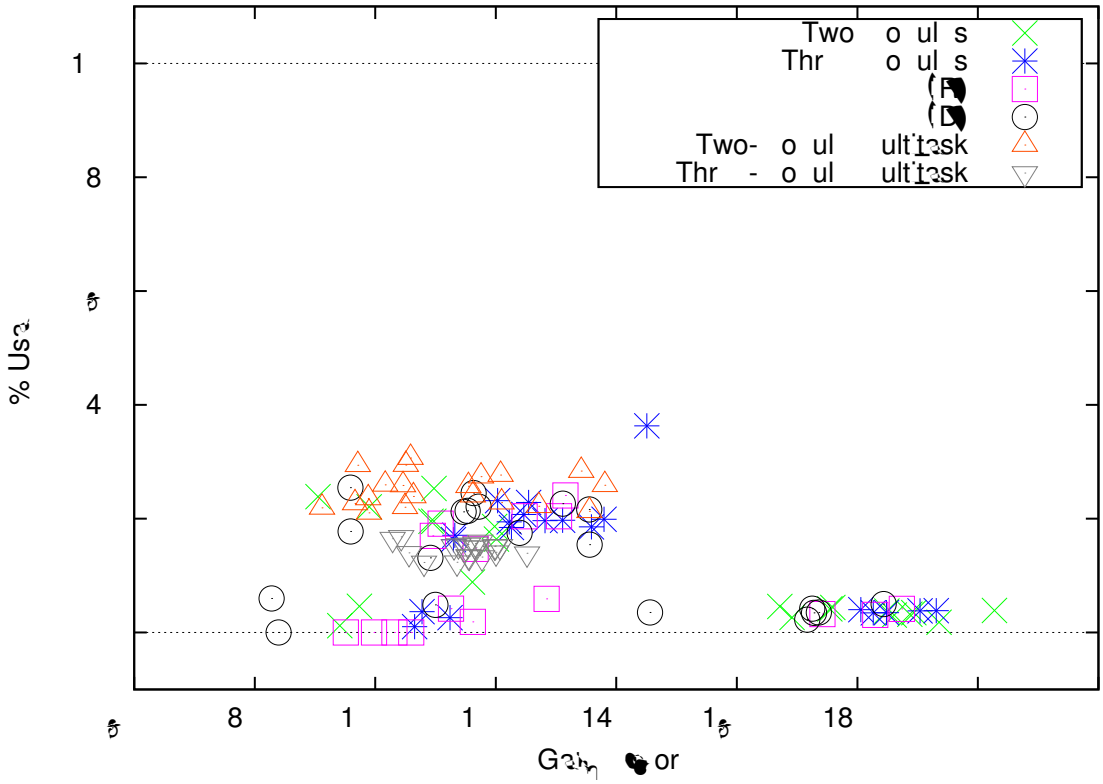


Figure 10.10: **Average Champion Game Score vs. 2nd Most Used Module Usage in One Life Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 10.9 are shown in this figure, but are plotted against usage of the 2nd most used module for each champion that has at least two modules. As with the Imprison variant, most of the usage scores here are a perfect mirror of the usage of the most used modules, which indicates that only two modules are used by most champions. The major exception to this is `Three-Module Multitask`$_{Con}$ (within the center cluster), which is required to use all three modules. However, it is once again surprising that `Three Modules`$_{Con}$ and both Module Mutation approaches do not take advantage of a third module, since it could hypothetically be even more useful in this more challenging domain. The disuse of more than two modules is confirmed in Figure 10.11.

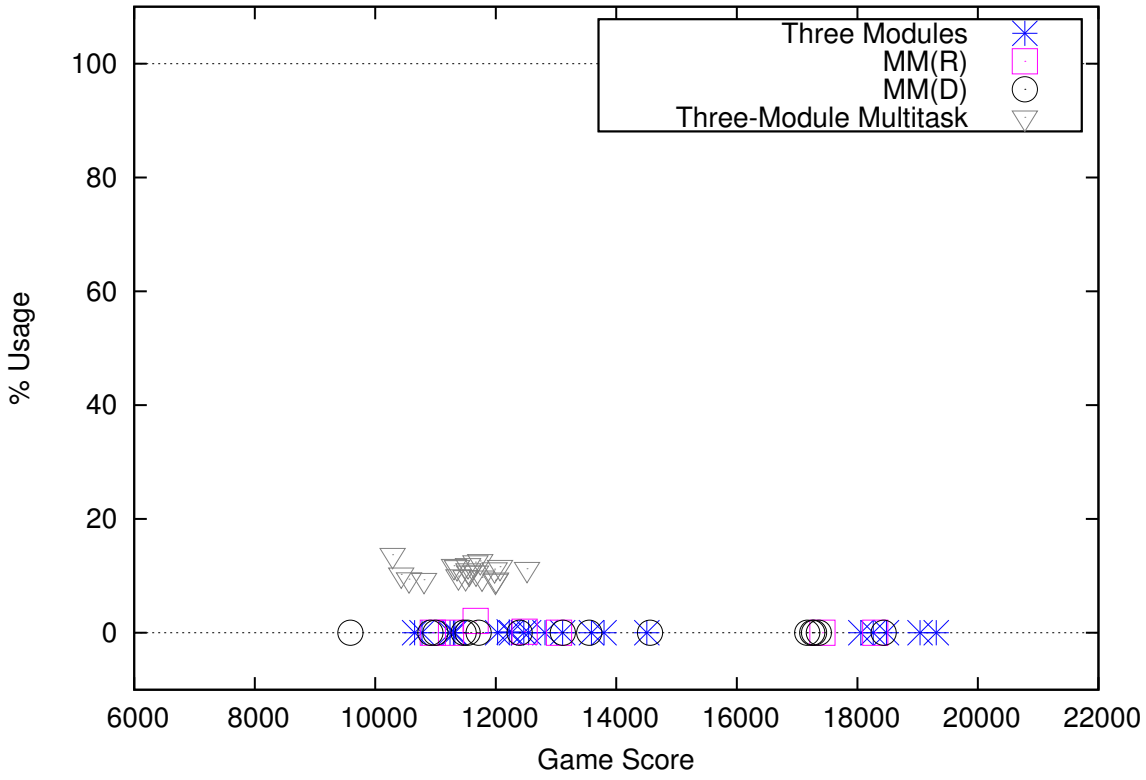
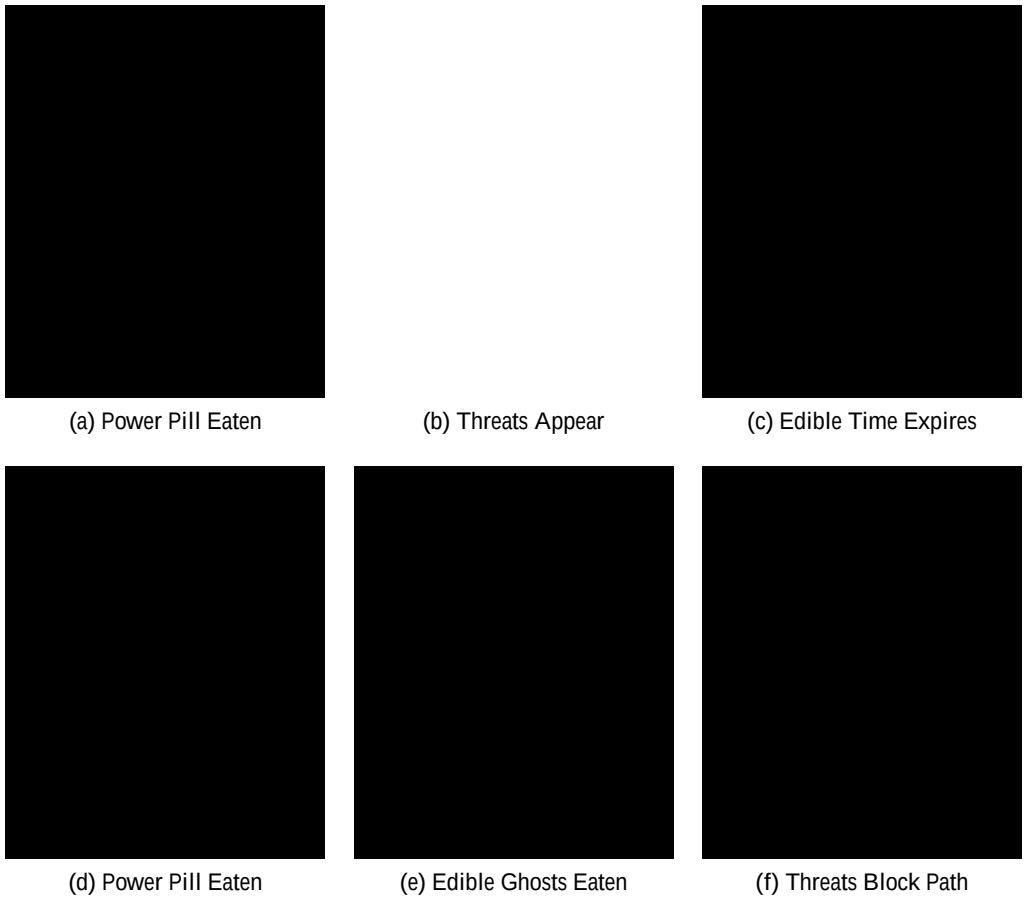


Figure 10.11: **Average Champion Game Score vs. 3rd Most Used Module Usage in One Life Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 10.9 are shown in this figure, but are plotted against usage of the 3rd most used module for each champion that has at least three modules. Except for `Three-Module Multitask`$_{\texttt{Con}}$ runs and a single `MM(R)`$_{\texttt{Con}}$ run (but just barely), all champions with three modules choose to never use the third one. Even in the full game, sticking to two modules is sufficient to learn good behavior in Ms. Pac-Man.



(a) Power Pill Eaten (b) Threats Appear (c) Edible Time Expires

(d) Power Pill Eaten (e) Edible Ghosts Eaten (f) Threats Block Path

Figure 10.12: **Threat/Edible Module Split in One Life Ms. Pac-Man.** This $\texttt{Two Modules}_{\texttt{Con}}$ network is a member of the threat/edible cluster. Each row shows a sequence of events following the eating of a different power pill. Green trails are shown in places where Ms. Pac-Man uses the module for eating edible ghosts. (a) The green module activates as soon as a power pill is eaten. Ms. Pac-Man heads towards nearby edible ghosts (the green trail loops back on itself when Ms. Pac-Man changes direction to better intercept the ghosts). (b) After some ghosts are eaten, they go to the lair for a short time before reemerging as threats. However, Ms. Pac-Man continues to use the green module. (c) Ms. Pac-Man pursues the last edible ghost, and only stops using the green module when the edible time expires. Throughout, the threat ghosts did not block the path to the last edible ghost. (d) After eating another power pill, Ms. Pac-Man pursues nearby ghosts. (e) The green module stays active as ghosts are eaten. (f) Ms. Pac-Man stops using the green module after threats reemerge because they block the path to the last edible ghost at the top of the maze. She backtracks away from the threats rather than pursue the last edible ghost. These examples show that modular networks can learn multimodal behavior even when tasks are blended. The details of the situation determine which mode of behavior should be used.

crossover parameters are the same. Each network is still evaluated 10 times to get average fitness scores. Networks still start fully connected, and each population size is still $\mu = \lambda = 100$. Each run still lasts 200 generations. However, instead of running 20 runs of each type, only 10 runs of each type are conducted, due to the increased cost of CPU time caused by Ms. Pac-Man having multiple lives in each evaluation.

The specific approaches evaluated in the ML variant are also different. Each network design is evaluated both with plain NSGA-II, and with NSGA-II combined with TUG. The specific methods evaluated are: `One Module`, `TUG One Module`, `Two Modules`, `TUG Two Modules`, `Three Modules`, `TUG Three Modules`, `MM(D)`, and `TUG MM(D)`. Module Mutation Random and the Multitask Learning approaches are not included because of their poor performance in OL. These experiments take the best modular approaches from the OL experiment, and push them further using multiple lives and TUG to see what the highest attainable scores are.

The version of TUG used in these experiments is slightly different from the version used in the Battle Domain (Chapter 7). Objectives are still switched off when goals are achieved, and goals still increase when all of them are achieved. However, instead of moving goals closer to the current maximum score in the objective, goals are increased by fixed increments. Appropriate increment sizes can be easily chosen because the maximum attainable scores in each objective are known in advance: `Pill Score` has a maximum of 948, and `Ghost Score` has a maximum of 240. The corresponding goal increments are chosen so that it will take approximately 50 goal increases to reach these maximum scores. The specific goal increments are 5 for `Ghost Score` and 20 for `Pill Score`. Preliminary experiments indicated that steadily increasing the goals in this manner leads to more reliable results in Ms. Pac-Man than the dynamic goal adjustment scheme used in the Battle Domain.

Unlike the experiments in OL, only `Conflict` sensors are included in ML. The Imprison and OL experiments indicated that the `Conflict` sensor setting is the more interesting variant because is shows how modular networks solve a problem that single-module networks cannot. The more general `Conflict` sensors do not require knowledge of how to implement a task division, and because they impose no such division, there is less bias in such sensors. However, modular approaches can still achieve high performance when using them. Such is also the case in the ML results, presented next.

10.6 Results of Multiple Lives Experiments

Scores are much higher in ML because having more lives offers multiple chances to get more points, and also makes it more likely to clear levels, providing access to more pills and ghosts to eat. However, although all scores are inflated, certain approaches are better than others. All modular approaches reach a level of performance above $\texttt{One Module}_{\texttt{Con}}$, and $\texttt{Three Modules}_{\texttt{Con}}$ is significantly better than $\texttt{One Module}_{\texttt{Con}}$ ($p < 0.05$) until the end of evolution. When combined with

TUG, all modular network approaches are significantly better than One Module$_{\text{Con}}$ ($p < 0.05$). Figure 10.13 shows the learning curves for each method during evolution, and Figure 10.14 compares each modular approach with both its TUG variant and One Module$_{\text{Con}}$ with 95% confidence intervals.

The one method not mentioned yet is TUG One Module$_{\text{Con}}$. This approach did not perform well: The average performance across 10 runs was lower than that of One Module$_{\text{Con}}$. This result is misleading, however, because the distribution of TUG One Module$_{\text{Con}}$ scores was bimodal: There were seven good runs and three bad runs (Figure 10.15). The bad runs flatline early at a low score. The good runs do much better, although not better than One Module$_{\text{Con}}$ (there is no significant difference between the two). This result can be understood by looking at the TUG goal behavior in both a good and bad run (Figure 10.16). A good run leaves the Pill Score deactivated most of the time, and focuses on the Ghost Score, but only after skilled baseline pill eating has been established. TUG runs with modular networks behave similarly. In contrast, the bad TUG One Module$_{\text{Con}}$ runs also focus mainly on the Ghost Score, but do so even when the Pill Score is low. However, recall that the best way to increase the Ghost Score early in evolution is to simply visit more levels and get more ghost-eating chances. If the Pill Score stays deactivated, then Ms. Pac-Man will never beat the first level, and will never get those ghost-eating opportunities. Thus, TUG does not help single-module networks in Ms. Pac-Man the way it did in the Battle Domain (Chapter 7).

Another interesting result is the improved performance of Three Modules$_{\text{Con}}$ relative to Two Modules$_{\text{Con}}$. In the Imprison variant these methods performed at the same level; in OL, Two Modules$_{\text{Con}}$ was slightly better, but that relationship is reversed in ML. The good performance of Three Modules$_{\text{Con}}$ is due to a tendency to discover luring more often. Perhaps having the extra module makes it more likely for one of the three to discover luring, but only if the evaluation environment forgives bad module behaviors (with extra lives) that are discovered and discarded along the way. Four champions (one Three Modules$_{\text{Con}}$, one MM(D)$_{\text{Con}}$, and two TUG MM(D)$_{\text{Con}}$) actually discover exactly the task division that surprisingly did not emerge in the Imprison domain (Section 9.6): one module for luring, a second for eating edible ghosts, and a third for dealing with threat ghosts when not luring (Figure 10.17). This three-module usage pattern was also missing from OL, but ML evaluations allowed it to be discovered. Evolving coordination between three distinct modules is difficult, and having only one life per evaluation likely discourages the evolution of networks that take such a risk. However, ML evaluations give networks multiple lives, so in these runs the networks were likely able to maintain high enough fitness scores to survive until a generation where their offspring accomplished this difficult feat.

The module usage pattern of all champions are shown in Figures 10.18, 10.19, and 10.20. Despite the presence of champions using three modules, module usage in ML is mostly the same as in Imprison and OL: The best modular approaches (scoring over 33,000) use one module a large majority of the time, over 95%, and the other 1%–5% of the time a luring module is used. Most

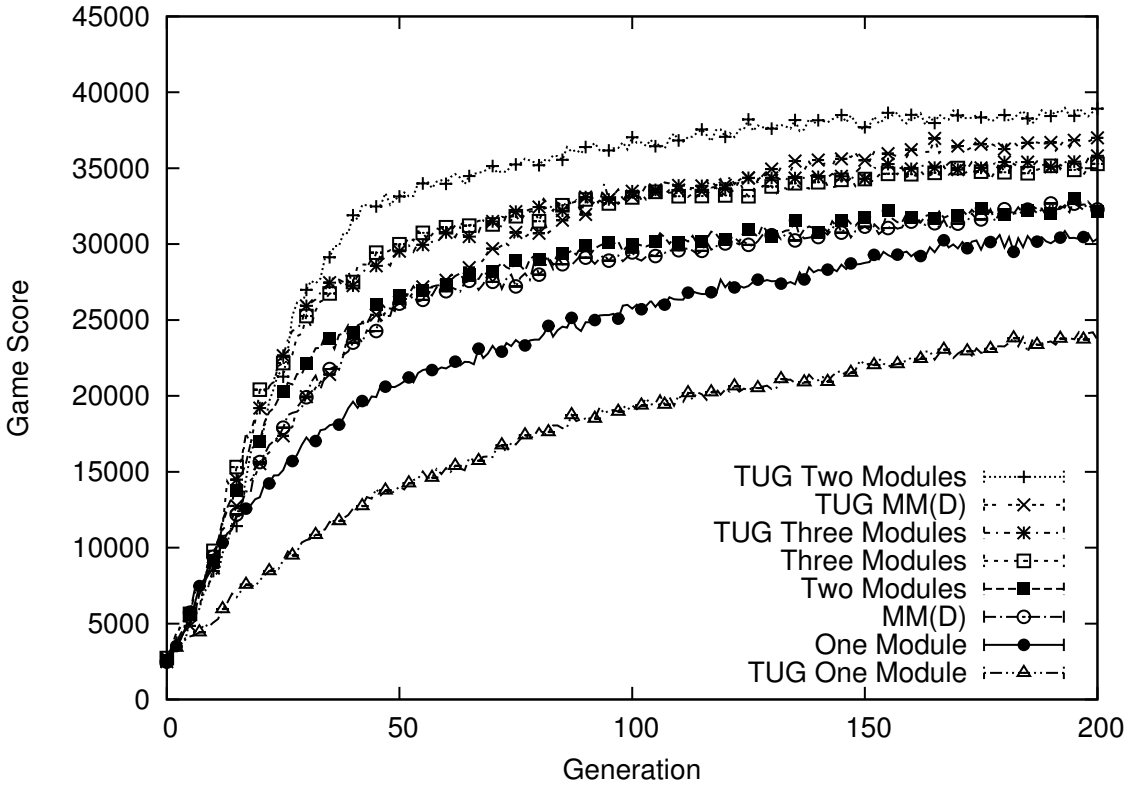
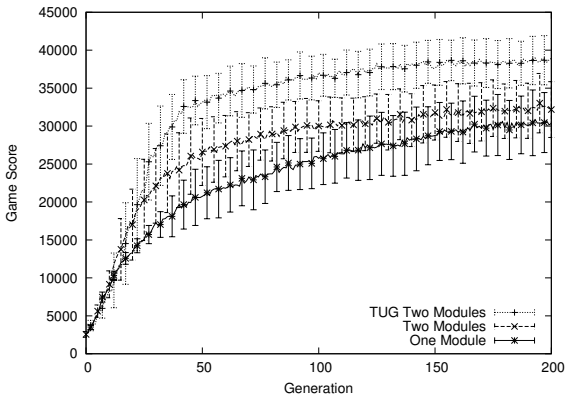
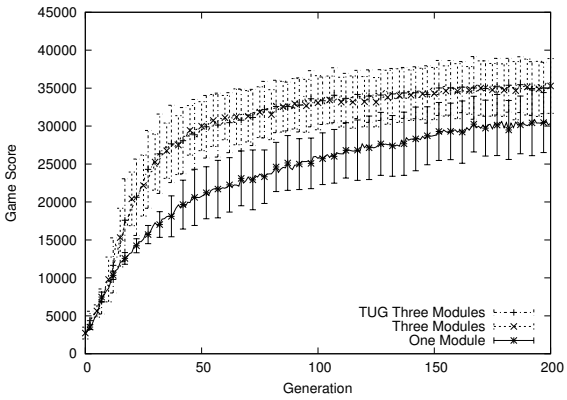


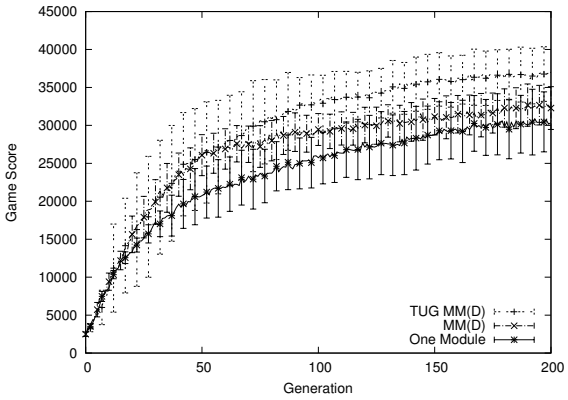
Figure 10.13: **Average Champion Game Score in Multiple Lives Ms. Pac-Man with `Conflict` Sensors.** For each method the average champion game score across 10 runs is shown. All modular approaches outperform `One Module`$_{\texttt{Con}}$. Also, the TUG variant of each modular approach results in scores that are usually better than runs without TUG. Specifically, `TUG Two Modules`$_{\texttt{Con}}$ is better than `Two Modules`$_{\texttt{Con}}$, `TUG MM(D)`$_{\texttt{Con}}$ is better than `MM(D)`$_{\texttt{Con}}$, and `TUG Three Modules`$_{\texttt{Con}}$ is equal to `Three Modules`$_{\texttt{Con}}$ in performance. Modular approaches do better than a single module, and for the most part TUG boosts scores even higher. The only method with averages worse than `One Module`$_{\texttt{Con}}$ is `TUG One Module`$_{\texttt{Con}}$. As shall be shown in Figure 10.15, this average is misleading because the distribution of `TUG One Module`$_{\texttt{Con}}$ scores is extremely bimodal.



(a) One Module$_{Con}$ vs. Two Modules$_{Con}$ and TUG Two Modules$_{Con}$



(b) One Module$_{Con}$ vs. Three Modules$_{Con}$ and TUG Three Modules$_{Con}$



(c) One Module$_{Con}$ vs. MM(D)$_{Con}$ and TUG MM(D)$_{Con}$

Figure 10.14: **Comparing Modular Approaches Against One Module$_{Con}$ via Average Champion Game Score in Multiple Lives Ms. Pac-Man with Conflict Sensors.** Results from Figure 10.13 are shown with 95% confidence intervals comparing each modular approach both with and without TUG against One Module$_{Con}$ individually. (a) Both Two Modules$_{Con}$ and TUG Two Modules$_{Con}$ are significantly better than One Module$_{Con}$ early in evolution, but only TUG Two Modules$_{Con}$ stays significantly better all the way to the end of evolution. (b) Three Modules$_{Con}$ and TUG Three Modules$_{Con}$ are both significantly better than One Module$_{Con}$, but not different from each other. TUG neither helps nor hinders evolution in this case. (c) MM(D)$_{Con}$ is like Two Modules$_{Con}$ in that it is significantly better than One Module$_{Con}$ early in evolution, but its average is enveloped by the confidence intervals of One Module$_{Con}$ as evolution progresses. TUG MM(D)$_{Con}$ starts out about equal to MM(D)$_{Con}$, but with wide confidence intervals. However, TUG MM(D)$_{Con}$ performance steadily improves and the intervals become narrower until it is significantly better than both One Module$_{Con}$ and MM(D)$_{Con}$. These results show how TUG gives a significant boost to modular networks so that they can get higher scores than networks with a single module.

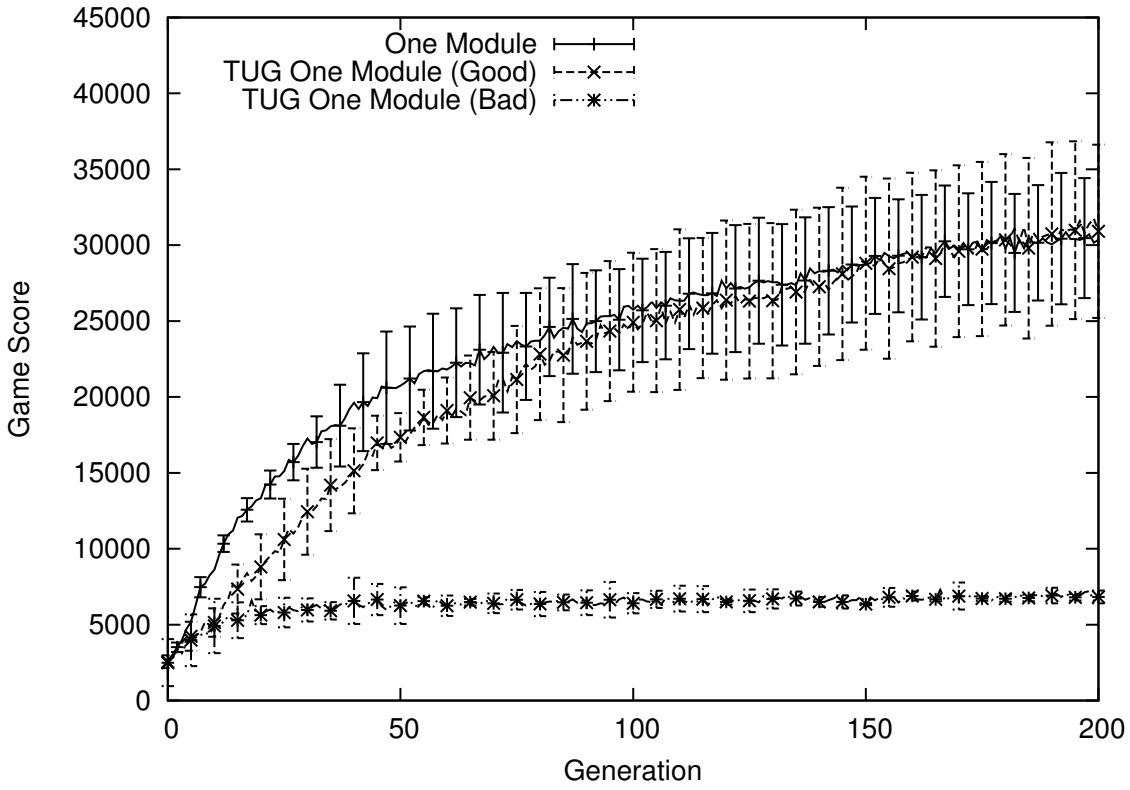
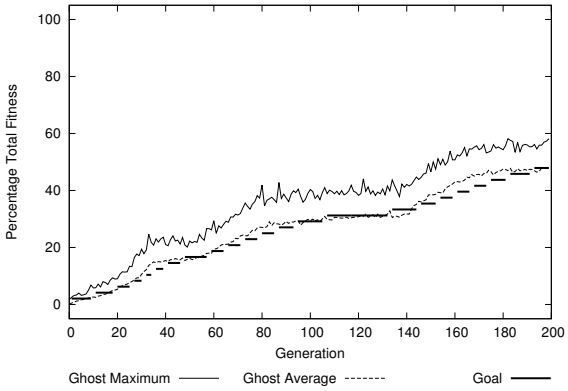
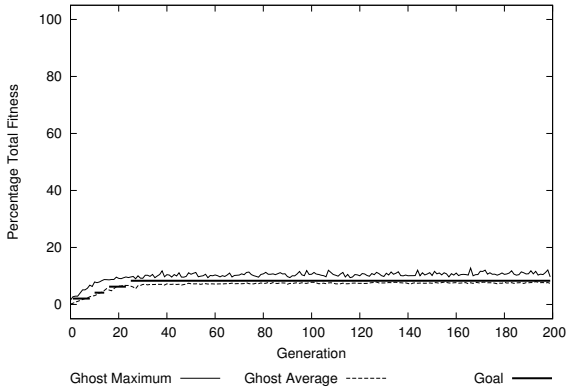


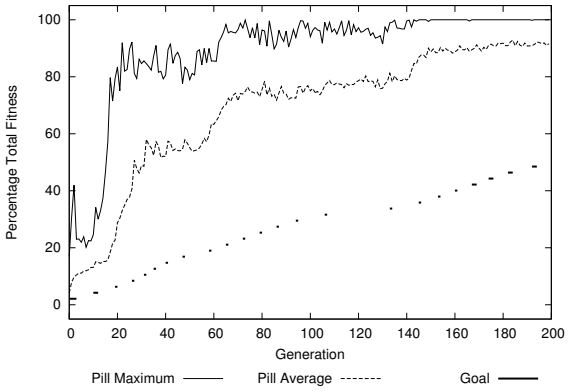
Figure 10.15: **Comparing** `One Module`$_{\texttt{Con}}$ **to** `TUG One Module`$_{\texttt{Con}}$ **in Multiple Lives Ms. Pac-Man with** `Conflict` **Sensors.** Of the 10 `TUG One Module`$_{\texttt{Con}}$ runs, three performed poorly and seven performed at the same level as `One Module`$_{\texttt{Con}}$. The average champion score of each of these groups is plotted separately with 95% confidence intervals. Performance is consistent within each group. For comparison, the average champion score across all 10 runs of `One Module`$_{\texttt{Con}}$ is also plotted with 95% confidence intervals, demonstrating that its performance is not significantly different from the good runs with TUG. Thus, in ML Ms. Pac-Man, when restricted to a single module with `Conflict` sensors, using TUG actually hurts performance. The reason is that it prematurely focuses too much attention on the `Ghost Score` (Figure 10.16).



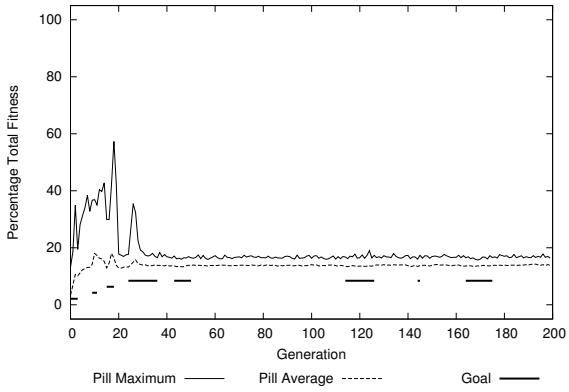
(a) Good `TUG One Module`$_{Con}$ `Ghost Score` Goal Behavior



(b) Bad `TUG One Module`$_{Con}$ `Ghost Score` Goal Behavior



(c) Good `TUG One Module`$_{Con}$ `Pill Score` Goal Behavior



(d) Bad `TUG One Module`$_{Con}$ `Pill Score` Goal Behavior

Figure 10.16: **Comparison of Good and Bad `TUG One Module`$_{Con}$ Runs.** Each plot shows the average and maximum population scores for one of the objectives in a run of `TUG One Module`$_{Con}$. Scores are normalized, since the maximum possible score in each objective is known. The current goal used by TUG is shown in each plot, but only for the generations when the goal is not currently achieved; that is, an objective is active on generations where a goal value is displayed, and inactive on other generations. (a) In a good run, `Ghost Score` steadily increases. Every time the `Ghost Score` goal is achieved, the goal is increased without the objective having a chance to deactivate. (b) In a bad run, the `Ghost Score` also remains active, but cannot increase. (c) `Pill Score` for the good run takes off early, and achieves its goal so quickly that this objective is seldom used (goal line is dashed). This behavior is good as long as the score stays high. (d) In a bad run, the `Pill Score` goal is achieved a few times in the first few generations, but before Ms. Pac-Man learns how to clear the first level, `Pill Score` gets stuck just above the goal, but just below what is needed to clear the first maze. The `Pill Score` is deactivated because its goal is achieved, even though more pills are needed to clear the maze and open up more ghost-eating opportunities. The low `Pill Score` is therefore the reason why the bad runs fail.

(a) Run From Threats (b) Lure To Power Pill (c) Eat Ghosts

Figure 10.17: **Intelligent Use of Three Modules by a** $\texttt{Three Modules}_{\texttt{Con}}$ **Network.** This network has separate modules for luring and eating ghosts. (a) Ms. Pac-Man uses the red module while allowing the ghosts to get close. (b) Once Ms. Pac-Man is close to a power pill, the luring module (green) activates and leads her down the corridor toward it. (c) After eating the power pill, the edible module (cyan) activates and Ms. Pac-Man quickly eats the three nearest ghosts. The results thus far have repeatedly demonstrated that luring and ghost-eating modules are useful, but only ML was able to produce networks with separate modules for each of these behaviors, thanks to its lenient evaluation scheme.

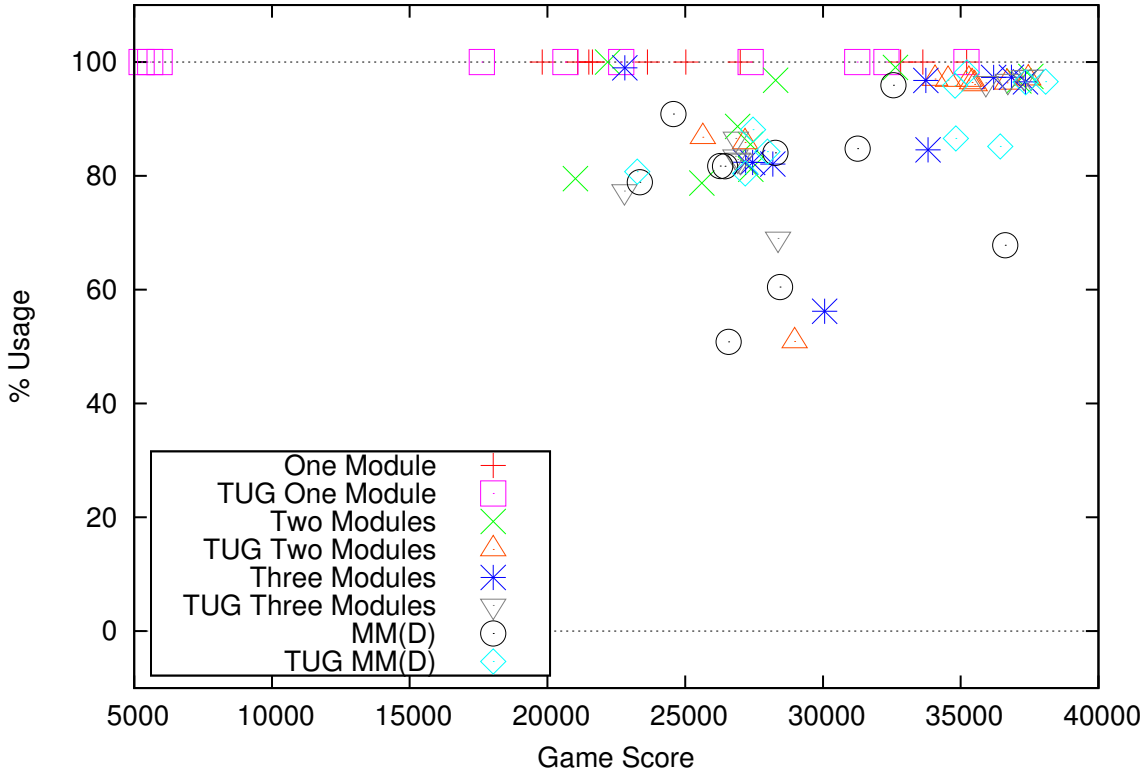


Figure 10.18: **Average Champion Game Score vs. Most Used Module Usage in Multiple Lives Ms. Pac-Man with `Conflict` Sensors.** The champion of each of 10 runs per method is evaluated 100 times (to minimize the effects of noise in evaluation), and the resulting average game scores are plotted against the percentage of time steps where the most used module was chosen (the "chosen" module is defined the same way as in Figures 9.1 and 10.9). The luring cluster is still in the top right, and contains many champions from TUG runs. As usual, a few $\texttt{One Module}_{\texttt{Con}}$ outliers are mixed in at the lower edge of this cluster, but the majority have lower scores. However, the lowest scores are achieved by bad $\texttt{TUG One Module}_{\texttt{Con}}$ runs at the upper left. The threat/edible cluster is in the middle as usual (scores 21,000–28,000, usage 75%–90%). However, there are a few champions with a different module usage pattern that score slightly better than the threat/edible cluster. These champions choose one module roughly 50% of the time, and score from 28,000 to 30,000, even though the task division they use is strange (Figure 10.21). Four champions that intelligently use three modules (Figure 10.17) have scores from 31,000 to 37,000 and usage around 85% (the module for avoiding threats). Thus, the success of modular methods that learn luring behavior is repeated in ML Ms. Pac-Man, and TUG variants are shown to be especially good at learning luring in this variant.

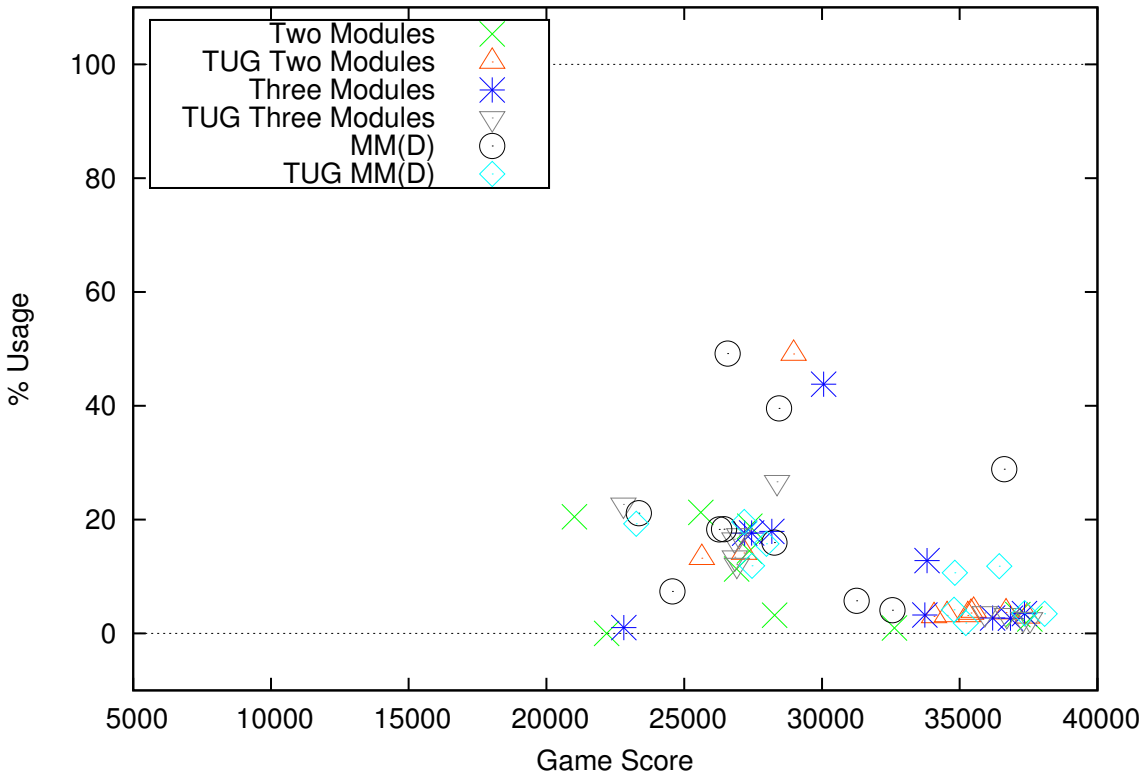


Figure 10.19: **Average Champion Game Score vs. 2nd Most Used Module Usage in Multiple Lives Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 10.18 are shown in this figure, but are plotted against usage of the 2nd most used module for each champion that has at least two modules. As in Imprison and OL, modular champions mostly favor two modules even when more are available. The luring cluster is at the lower right (usage below 5%), and the threat/edible cluster is in the center (scores from 20,000–28,000) with usage of the second module between 10% and 25%. With scores between 28,000 and 30,000 are the even split networks (usage about 50%) that use an unusual task division. Runs that use three distinct modules (luring, eating edible ghosts, and avoiding threats) are directly above the luring cluster: They use the edible module the second most (usage around 12%), and have scores from 31,000 to 37,000.

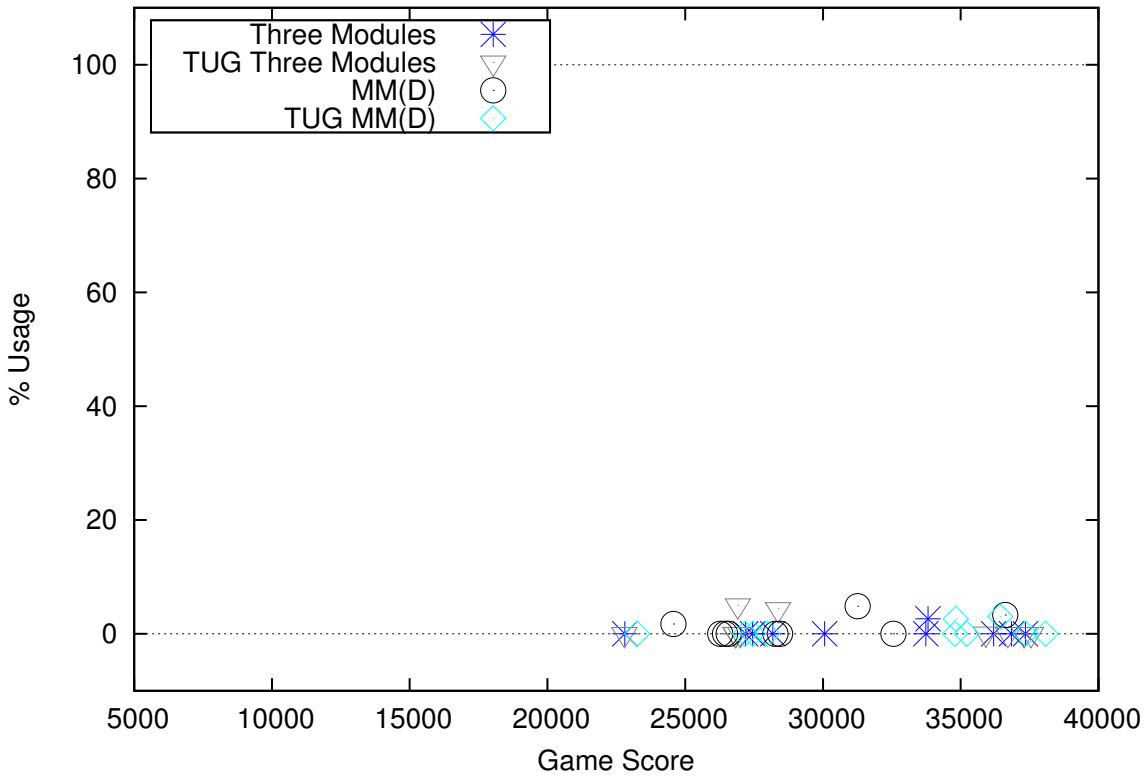
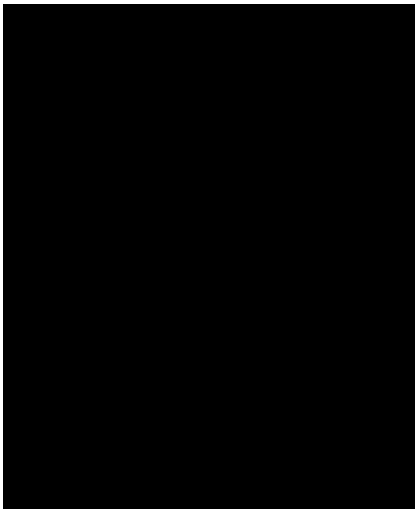


Figure 10.20: **Average Champion Game Score vs. 3rd Most Used Module Usage in Multiple Lives Ms. Pac-Man with `Conflict` Sensors.** The same game scores for champions described in Figure 10.18 are shown in this figure, but are plotted against usage of the 3rd most used module for each champion that has at least three modules. Most networks with a third module do not use it, but some champions use their third module for luring just under 3% of the time.



(a) Module 1 (b) Module 2

Figure 10.21: **Indiscriminate Even Split by** `TUG Two Modules`$_{\texttt{Con}}$ **Network.** Similar to behavior discovered by MM(D) in the Imprison variant (Figure 9.10), the behavior shown above involves switching back and forth between modules without a clear reason why. (a) The green trails show where Ms. Pac-Man was using Module 1, and the gaps are places where Module 2 was used. (b) The Module 2 locations are shown in cyan. Recall that the brightness of the trails increases as Ms. Pac-Man spends more time using a particular module in a location, and fades with time. It is not clear for what aspects of the domain each module is responsible. However, its performance is better than that of networks using a threat/edible division, which indicates that TUG did well to discover it.

champions still favor two modules. Most champions with middling scores (25,000–29,000) display a threat/edible task division resulting in the threat module being used 75%–90% of the time. The worst scores (below 25,000) still result primarily from using only one module, with the worst `TUG One Module`$_{\texttt{Con}}$ runs as an extreme example.

However, there is another type of task division resulting in an even 50%/50% split between two modules (Figure 10.21). These networks use their modules rather indiscriminately, similar to behavior discovered in the Imprison variant (Figure 9.10). This behavior results in scores around 30,000, between the threat/edible and luring clusters. However, it is not clear why this particular module division results in this level of performance. The behavior itself is good, but there is no clear difference in how each module behaves. Videos of this and other behaviors evolved in ML are available online at `http://nn.cs.utexas.edu/?ml-pm`.

This module usage pattern is uncommon. Most module usage patterns in ML are the same as in OL. However, there are interesting behavioral differences. Because ML agents can afford to lose lives, even the successful agents tend to do so. However, agents are only slightly less cautious than OL agents, since there is a limit on the number of lives they can lose. If an agent clears all

levels, it does not matter how many lives were lost, since its fitness values are not affected at all. As a result, many successful agents lose a few lives. Such losses could probably be prevented if an additional objective discouraged the loss of lives.

Another evaluation difference between OL and ML is level time limits. Final OL champions never ran afoul of the 8,000 step time limit, and although ML runs had a much larger time limit, the final champions do not spend more than 8,000 time steps per maze either. However, ML agents tend to dawdle more than OL agents (Figure 10.22), especially champions using only one module. Such behavior was discouraged in OL and Imprison by having a lower time limit. In order to discourage it in ML, an additional objective could be added to reward completing levels quickly, or there could be an overall time limit on evaluations, meaning that agents that beat levels quicker will get to visit more of them. Removing the four level limit from evaluations could also discourage loss of lives, but this change would result in extremely long evaluation times.

Overall, ML behavior is very skilled. The next section will show how well the methods of this dissertation perform not just with respect to each other, but with respect to the best scores put forth in the literature.

10.7 Comparison with Previous Research

In order to compare performance with the literature, slight changes need to be made in the game setup. Although many researchers have used the same simulator, different publications have different evaluation rules. The evaluation scheme in this dissertation so far involves visiting each of the four mazes once and ending evaluation after that. This variant has been used by others, and will be called `Four Maze` in the results below.

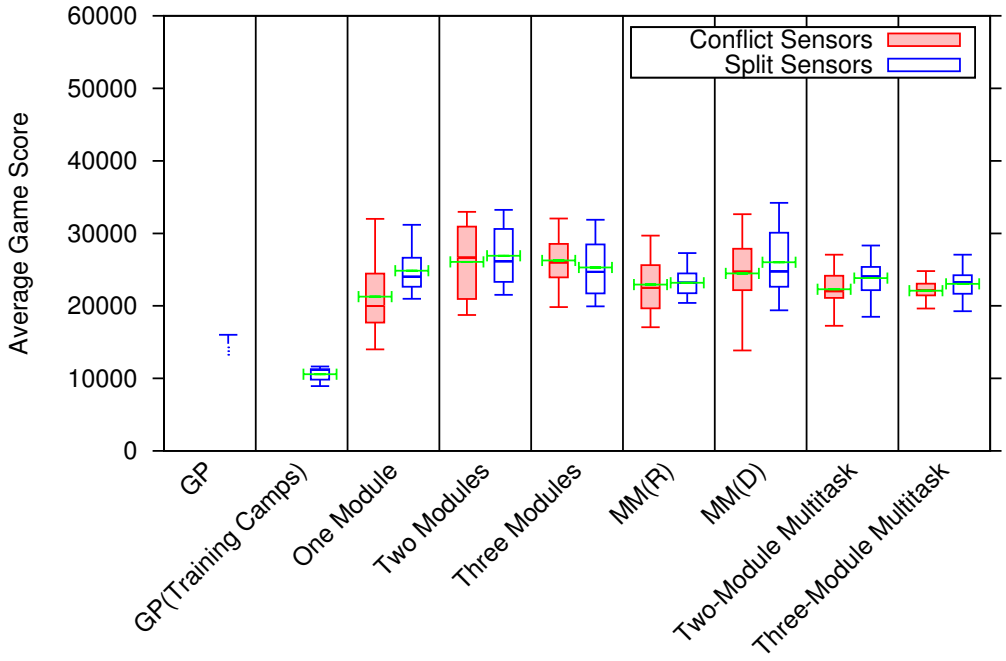
However, much of the literature describes entrants in the Ms. Pac-Man vs. Ghosts competitions (`MPMvsG` variant). These scores were achieved under the following additional rules: (1) Clearing the fourth maze leads back to the first maze, until each maze is visited four times (resulting it a total of 16 levels); (2) The per-level time limit is 3,000 time steps, but running out of time advances Ms. Pac-Man to the next level instead of killing her; and (3) Ms. Pac-Man is awarded half the score from remaining pills in a level when time runs out. These were the rules in the 2011 competitions. Although the rules changed slightly for the 2012 competition, no published results have yet made use of this updated rule set.

The `Four Maze` and `MPMvsG` variants are of interest to the larger community: They have each been used in more than one previous study. Therefore, they will be used below as well. However, there is one more feature that both methods share. Unlike evaluations during evolution, the evaluations in `Four Maze` and `MPMvsG` are timed, meaning that Ms. Pac-Man only has 40ms to decide on each action. This time limit is seldom a problem for the evolved networks, but whenever an action is not returned in time, the action made on the previous time step is repeated.

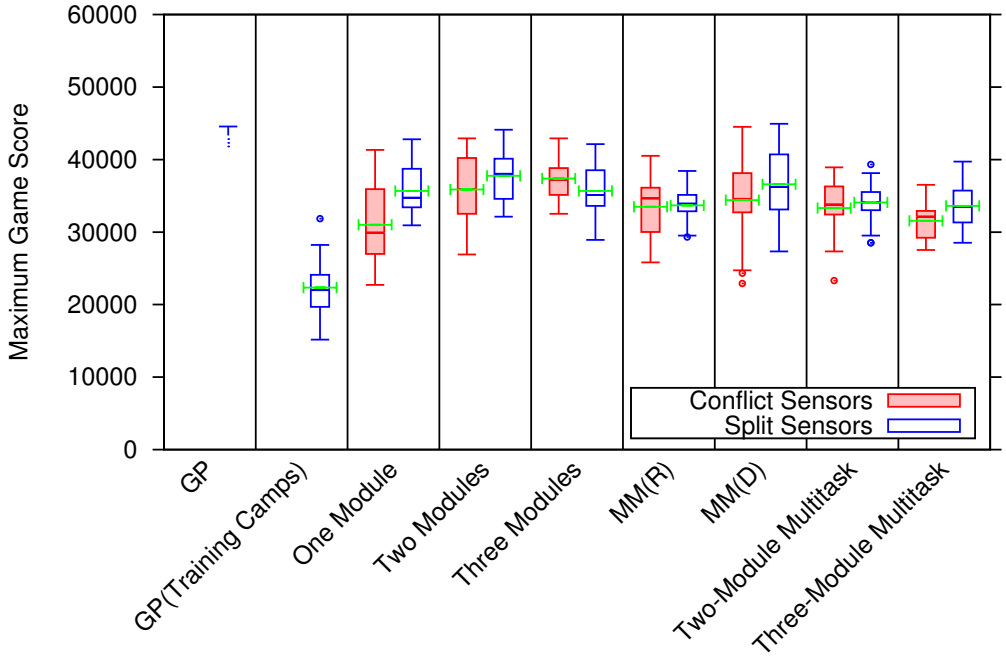
To compare scores achieved in this dissertation with those in the literature, champions from

(a) Maze 0 (b) Maze 1 (c) Maze 3

Figure 10.22: **Stalling Behavior by** One Module$_{\text{Con}}$ **Network.** ML agents with one module often resort to stalling. If the ghosts are in positions that make it unsafe to pursue certain pills, then Ms. Pac-Man can simply go around in circles and wait for the situation to change. The non-determinism of the game guarantees that ghosts will eventually reverse direction, and these events sometimes create openings to go for the hard-to-get pills. (a) The green trail shows locations Ms. Pac-Man has recently visited, with highlighting for older visits eventually fading to black. This trail in Maze 0 has been going in circles for a long time, because the trail no longer shows how Ms. Pac-Man initially entered this loop. (b) The same network stalling for time in Maze 1. (c) The same network stalling for time again in Maze 3. Although luring depends on eating power pills at the right time, quick action still needs to be taken in order to eat ghosts. Additionally, although random ghost reversals can sometimes create useful openings, they can also lead to surprise capture. Stalling behavior is therefore a double-edged sword, and only marginally successful.



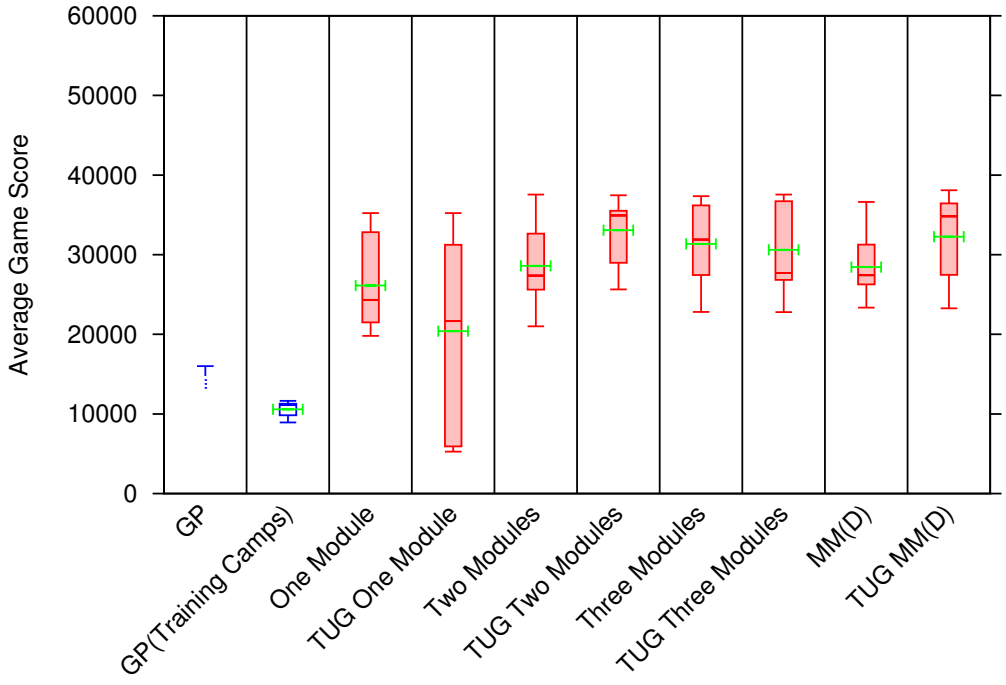
(a) Average OL Scores with `Four Maze` Evaluations



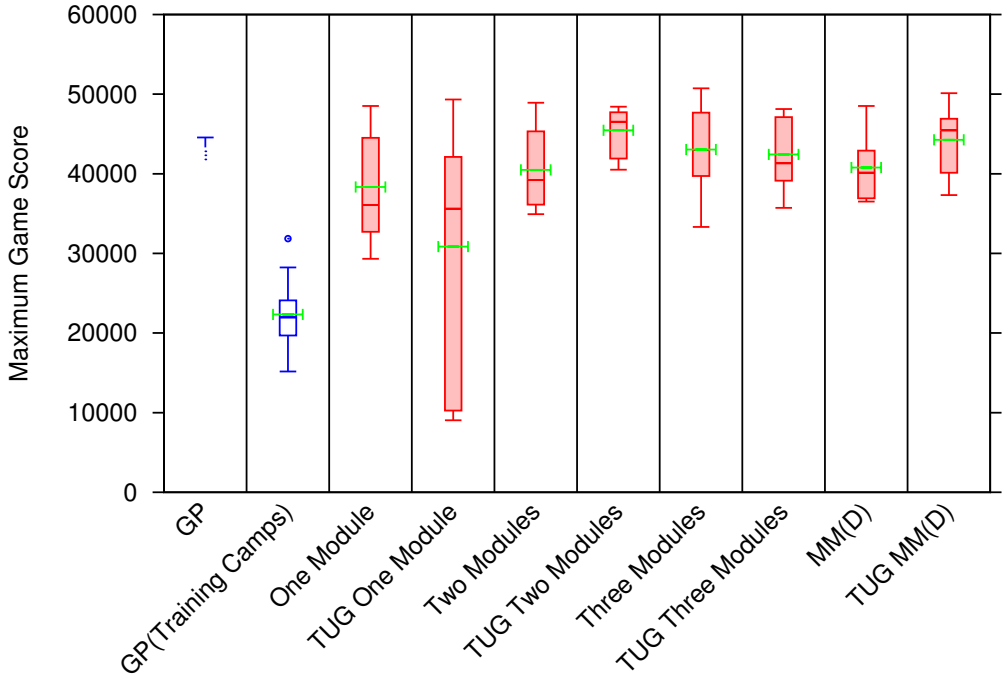
(b) Maximum OL Scores with `Four Maze` Evaluations

Figure 10.23: (Caption on following page)

Figure 10.23: **Comparison of Scores From One Life Ms. Pac-Man Experiments to Previous Scores in the Literature Using** `Four Maze` **Evaluation Rules.** Previous results include GP (Genetic Programming; Alhejali and Lucas, 2010) and GP plus Training Camps (Alhejali and Lucas,



(a) Average ML Scores with `Four Maze` Evaluations



(b) Maximum ML Scores with `Four Maze` Evaluations

Figure 10.24: (Caption on following page)

Figure 10.24: **Comparison of Scores From Multiple Lives Ms. Pac-Man Experiments to Previous Scores in the Literature Using `Four Maze` Evaluation Rules.** ML results are even better than OL results (Figure 10.23), because their behavior has been optimized for `Four Maze` rules. Average and maximum scores are higher than those generated by OL, and the maximum scores of methods from this dissertation are higher than even those of GP.

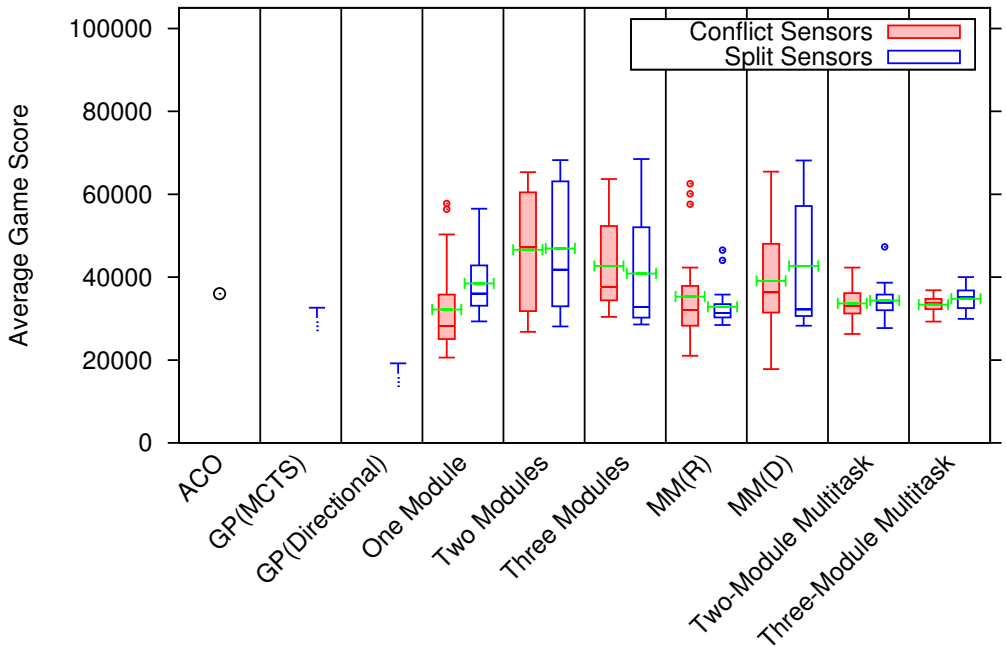
intersecting each box depicts the average.

Previous results from the literature are shown at the left of each subfigure. The original publications vary in what results they included, some providing more information than others. However, each previous study provided at least one, and often only one, set of statistics produced by a particular "best" agent. Depending on the study, this agent was evaluated anywhere from 100 to 1,000 times, and the average and maximum scores of the agent were provided. These scores are plotted in Figures 10.23–10.26. For the GP plus Training Camps approach of Alhejali and Lucas (2011), data from 10 different champions produced by 10 different evolutionary runs was available in a bar chart, so a box-and-whisker plot was created from this data[1]. For plain GP (Alhejali and Lucas, 2010), GP with a direction-evaluating policy (Brandstetter and Ahmadi, 2012), and GP used to learn a default policy for MCTS (Alhejali and Lucas, 2013) results were available only for the champions of the best evolutionary runs. All of these evolutionary methods treated threat and edible ghosts differently, and therefore made use of split sensors. Ant Colony Optimization (Recio et al., 2012) directly produces Ms. Pac-Man behavior, so it does not make sense to discuss multiple champions in this case. Therefore a single data point is shown for this method in each plot.

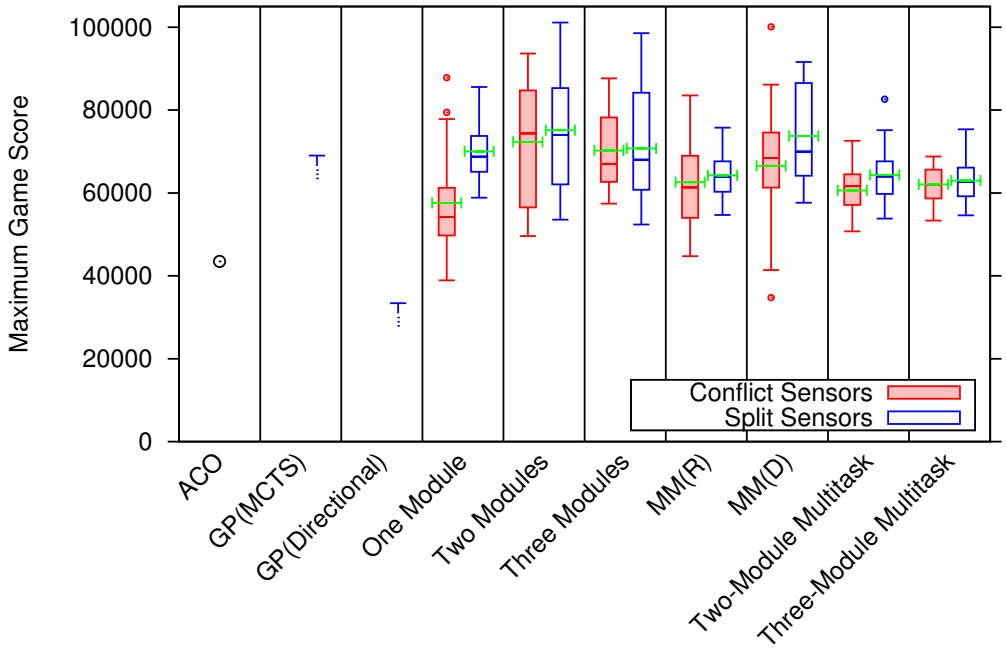
The main conclusion from these figures is that the methods in this dissertation result in better scores than the other methods in the literature. In `Four Maze` evaluations, all average champion scores from this dissertation are higher than those from previous work, as are many of the maximum scores. In `MPMvsG`, almost all methods score higher than agents in previous work. The highest scores are the result of modular networks that discover a luring module, but even champions that use a single module are mostly better than the previous results.

It is especially interesting that networks with a single module do better than the GP approach by Brandstetter and Ahmadi (2012). Their approach inspired the one used in this dissertation, but there are several differences. The most obvious difference is the type of function approximator being evolved, i.e. neural network vs. function tree. Another important difference is the sensors used. Most sensors were the same, but the GP approach sensed only the closest threat and the closest edible ghost. Therefore, the `Split` sensors were better than these, since they provided the same information in addition to information about the other ghosts. The `Conflict` sensors used in this dissertation were both at a disadvantage because they did not distinguish between threat and edible ghosts, and an advantage because they sensed all four ghosts instead of just the closest. Another important sensor used in this dissertation was the "Options From Next Junction" sensor

[1]The raw numerical data was obtained from the paper's first author.



(a) Average OL Scores with `MPMvsG` Evaluations



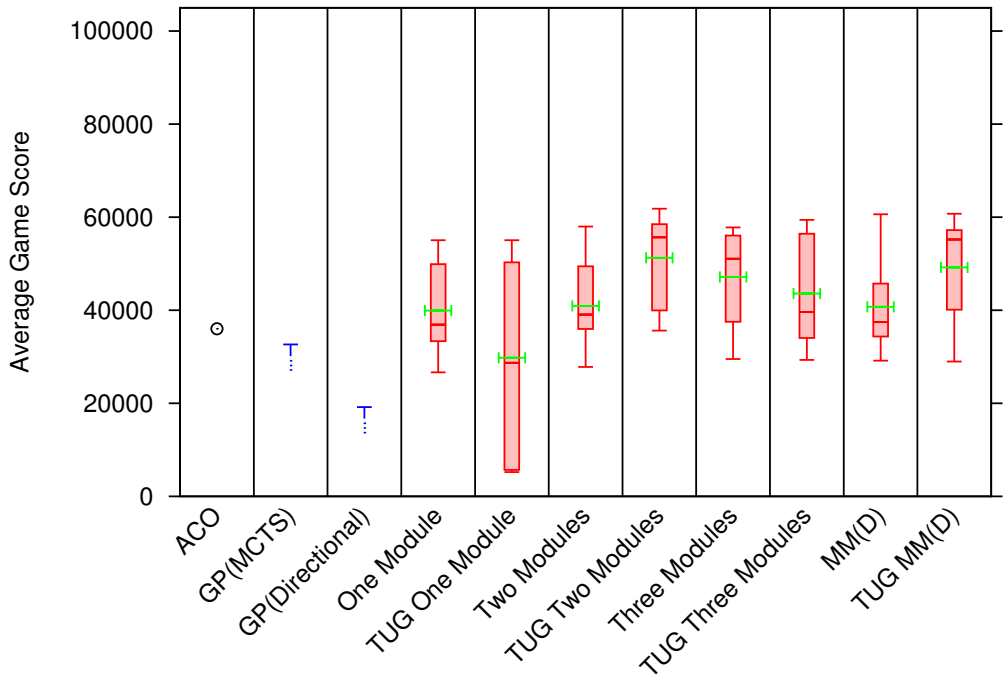
(b) Maximum OL Scores with `MPMvsG` Evaluations

Figure 10.25: (Caption on following page)

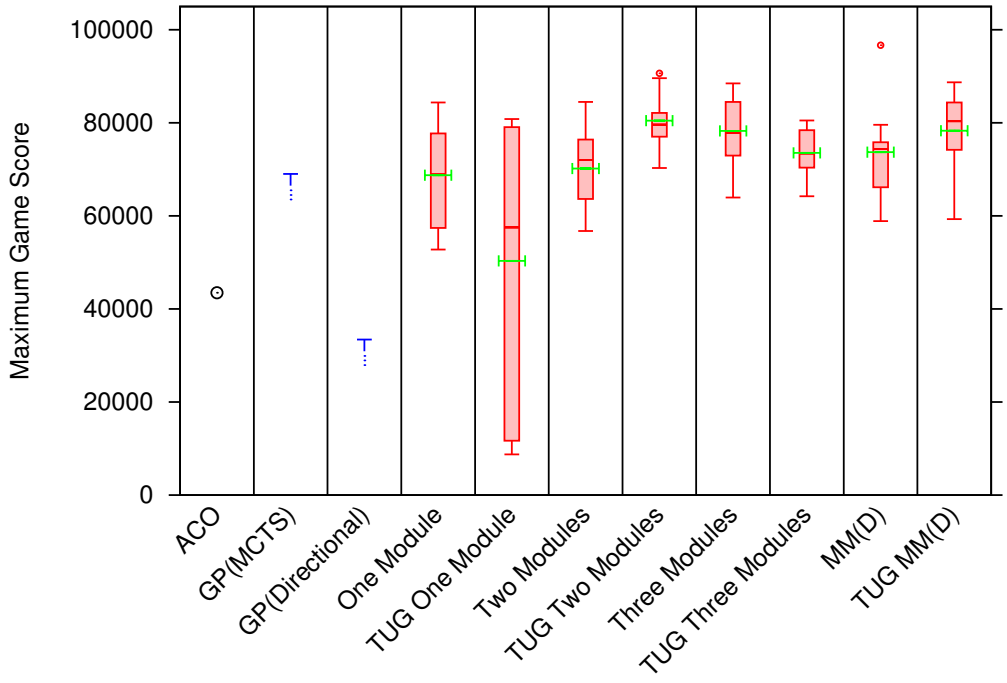
Figure 10.25: **Comparison of Scores From One Life Ms. Pac-Man Experiments to Previous Scores in the Literature Using MPMvsG Evaluation Rules.** Previous results include GP used to learn a direction-evaluating policy (Brandstetter and Ahmadi, 2012), as was done in this dissertation; GP used to learn a default policy for Monte-Carlo Tree Search (MCTS; Alhejali and Lucas, 2013); and Ant Colony Optimization (ACO; Recio et al., 2012). Scores with MPMvsG rules are much higher than with Four Maze rules (Figure 10.23) because the evaluation scheme is more lenient, and Ms. Pac-Man can visit more levels. The best scores from this dissertation are again better than all previously published results. In terms of average champion scores, the best previous approach is ACO, but all methods from this dissertation, except for Three-Module Multitask$_{Con}$ and Three-Module Multitask$_{Split}$, achieve higher average champion scores. In terms of maximum champion scores, the best previous approach is GP combined with MCTS, but once again the methods from this dissertation, except for Three-Module Multitask$_{Con}$, perform better. One Module$_{Con}$ and One Module$_{Split}$ also perform better than previous works, but the best approaches are again modular networks with preference neurons. In fact, the best Two Modules$_{Split}$, Three Modules$_{Split}$, and MM(D)$_{Split}$ champions outperform previous work by a wide margin. The multiobjective neuroevolution approach of this dissertation is thus stronger than previous approaches, and is enhanced even further by evolving modular architectures.

(Section 9.2), which does a better job of helping Ms. Pac-Man plan safe routes than the GP sensor that only checked the safety of the route up to the next junction. These sensors no doubt account for some of the difference in performance, but there were other important differences that likely also had an impact. The GP results were evolved using MPMvsG rules, which allowed Ms. Pac-Man to skip to the next level even if she had a hard time eating all pills. In fact, Brandstetter and Ahmadi specifically mentioned that their agents often do not try to eat difficult-to-reach pills, because such behavior is risky. Instead of a clever adaptation to the MPMvsG rules, this behavior might simply be a local optimum. These controllers are losing points that the neural networks of this dissertation were required to earn in order to advance from one level to the next. This comparison is thus an interesting example of how small differences in fitness incentives can result in different play styles.

Results from the ML experiments are another such example. These runs do better than OL in Four Maze evaluations, but worse than OL in MPMvsG evaluations. ML results were evolved using Four Maze rules, so it makes sense that their performance in these evaluations is strong. However, it seems that ML behavior is overfit to these rules, which hinders MPMvsG performance. In particular, ML results were evolved in an environment where agents can afford to lose up to three lives (once the fourth is earned) across four levels without any consequence. However, such behavior has consequences in MPMvsG evaluations, which can last up to 16 levels. The tendency of certain ML results to stall probably also results in many lost points, since pill and ghost eating chances are lost if each maze is not cleared within 3,000 time steps under MPMvsG rules. However, the skill of ML networks in Four Maze evaluations indicates that networks evolved using MPMvsG rules would likely learn specialized behaviors for this environment as well. Whether this would lead to



(a) Average ML Scores with `MPMvsG` Evaluations



(b) Maximum ML Scores with `MPMvsG` Evaluations

Figure 10.26: (Caption on following page)

Figure 10.26: **Comparison of Scores From Multiple Lives Ms. Pac-Man Experiments to Previous Scores in the Literature Using `MPMvsG` Evaluation Rules.** The champions from ML experiments score slightly lower than champions from OL experiments (Figure 10.25). Optimizing performance in `Four Maze` evaluations has detracted from `MPMvsG` performance, because losing lives has worse consequences in `MPMvsG`. However, the ML results are still better than previous results in the literature, once again emphasizing the success of multiobjective neuroevolution in general and modular networks plus TUG in particular.

even better scores, or overly cautious behavior as exhibited by Brandstetter and Ahmadi's GP results is unclear, but perhaps worth finding out in the future.

10.8 Conclusion

This chapter scaled up the methods developed in this dissertation to the original game of Ms. Pac-Man. Not only did these methods succeed at generating skilled multimodal behavior, but they were shown to be superior to previous approaches. Modular networks enable Ms. Pac-Man to discover a

Chapter 11

Related Work

Given the results presented so far in this dissertation, it is now time to put them in proper perspective by comparing them with related work in multimodal behavior. Such behavior is often labelled in the literature as strategic, high-level, hierarchical, or simply complex.

The following sections divide these approaches into three categories: (1) Design principles for hand-coding multimodal behavior, (2) approaches in which sub-behaviors are learned and then combined into a hand-designed hierarchy, and (3) methods in which multiple modes of behavior are learned by implicit hierarchies encoded within modular control policies.

11.1 Design Approaches

The most common approach to designing complex behavior for agents in both robotics and in simulated worlds (such as video games) is to simply hard-code all behaviors. This approach allows designers to be aware of how their agents should respond in many different situations, although obviously not in situations they did not anticipate. Thus hard-coded agents are reliable in typical situations but do not handle surprises well. Designing such agents also requires much advanced knowledge about the domain in which the agent will function, as well as design time and effort from programmers.

Because hard-coded designs are complex, it is important to employ design practices that assure that the design is functional and easy to understand. Two popular design paradigms are hierarchical subsumption architectures and behavior trees.

11.1.1 Hierarchical Subsumption Architecture

The subsumption architecture (Brooks, 1986) has remained a cornerstone of robotics ever since its inception. A more recent summary of its design is given by Prescott et al. (1999):

1. *Distributed, layered control:* Control is distributed across several layers operating in parallel, with no central control within a layer.

2. *Behavioral decomposition:* Each layer is designed to achieve some particular goal or task, thus different layers are oriented towards different behaviors of the overall multimodal system.

3. *Increasing "levels of competence":* Each higher layer of the architecture has access to the lower layers, and is able to arbitrate between them, but the lower layers function unaware of the higher layers. Thus, higher layers are able to refine collections of lower-level behaviors into a meaningful, multimodal behavior.

4. *Incremental construction:* Lower layers are frozen before higher layers are built on top of them.

5. *Subsumption:* Higher layers can subsume the roles of lower layers by taking over, suppressing, or replacing lower-level outputs with higher-level outputs.

Using the subsumption architecture means identifying sub-behaviors that are parts of larger behaviors and incorporating them into a hierarchy. The lowest level consists of several simple behaviors. The next layer can therefore perform multimodal behavior by arbitrating between and blending the different behaviors exhibited by each lower layer component. Though effective, this approach has the obvious drawback of requiring both the individual behaviors and the hierarchy of subsumption layers to be hand designed.

The bottom-up subsumption approach is popular in robotics, and these design principles actually guide much work in learning such hierarchical behavior. However, other methods, such as behavior trees, are favored in commercial video games.

11.1.2 Behavior Trees

Despite the advent of computational learning techniques, AI in commercial games typically depends on old-fashioned AI techniques such as rule-based systems and finite-state machines (Diller et al., 2004). However, a slightly more sophisticated, yet still hand-designed, AI mechanism popularized by the first-person shooter game Halo 2 is behavior trees (Isla, 2005). Behavior trees offer another way of designing agents hierarchically, though the design process is top-down instead of bottom-up as with the subsumption architecture.

A behavior tree starts at a root, under which there are several prioritized high-level behaviors. Each behavior has a firing condition that determines whether or not it should be active. The first behavior in the list whose firing condition is active will then be enacted by the agent. In the subtree beneath each high-level behavior is another prioritized list of behaviors, each with their

own firing conditions. The behavior selection process is repeated deeper into the tree until terminal low-level behaviors are chosen and carried out by the agent.

The behavior tree approach allows multimodal behavior to develop because distinct lower-level behaviors are chosen at different times based on priority and firing mechanism. However, like the subsumption architecture method, this approach requires careful human design of several component sub-behaviors. Such hand-coding can be successful if it is done according to well-thought-out design principles, and the programmers are able and willing to put in lots of time and effort. However, there is always a chance that some important behavior has been neglected. However, the programming burden can be lessened if parts of the behavior are learned rather than designed. The next section discusses approaches that still fit sub-behaviors into an explicit hierarchy, but allow learning to optimize the behavior.

11.2 Learning With Hierarchies

Designing complex controllers by hand is always challenging and sometimes infeasible because it is unclear how certain behaviors can be coded and combined. As a result, Machine Learning has been used to discover multimodal behavior through Hierarchical Reinforcement Learning, and through learned subsumption architectures.

11.2.1 Hierarchical Reinforcement Learning

The MAXQ (Dietterich, 1998) method is an early example of Hierarchical Reinforcement Learning (HRL). It learns behavior for sub-tasks given a hand-designed hierarchy. Many extensions to this work have been created since, including a multi-agent variant (Cheng et al., 2007) and a version that learns the hierarchy in addition to the sub-tasks (Hengst, 2002). However, MAXQ and its descendants have only been applied to domains with finite state and action spaces, typically large grid worlds.

In fact, MAXQ's success hinges partly on its reliance on state abstraction: Each sub-policy throws away what it deems are irrelevant portions of the state representation so that it can focus on a reduced state space. A state abstraction results in increased perceptual aliasing, which makes the state space look smaller, and can speed up learning if done properly. This approach differs from those developed in this dissertation in that each module of an evolved modular network has access to the full state representation. Allowing evolved networks to see the full state requires them to learn for themselves which aspects of the state representation are relevant, and also allows them to make use of the same information in multiple different ways. Two distinct modes of behavior may use all of the same information, but in different ways.

Another important concept in the realm of HRL is options, which are temporally-extended actions (Sutton et al., 1999). Options were mentioned briefly earlier as an example of a technique that had been applied to Ms. Pac-Man (Subramanian et al., 2011). This particular example

learned options from human subjects, which has also been done in domains with continuous state spaces (Konidaris et al., 2010). Methods for automatically learning options also exist (Stolle and Precup, 2002; Konidaris and Barto, 2009).

In practice, trajectories through state space are typically segmented into collections of options. As a result, options are a means of getting from point A to point B in state space. Formally, options will continue executing until a termination predicate activates, but it is common to think of an option as leading an agent to a termination state. Options define sub-policies that execute until told to stop. The network modules of this dissertation serve a role similar to options: They are sub-policies whose termination condition is either human-specified (Multitask Learning) or determined by the behavior of preference neurons. However, the design perspective behind modules differs in that each module vies for control of the agent on every time step. Also, because options are often used to traverse specific regions of state space, their behavior only needs to be defined on those regions.

Some examples of learned options from the Ms. Pac-Man experiments (Subramanian et al., 2011) were "Avoid ghost," and "Go to the nearest power pill." These high-level actions are very similar to those used in early Genetic Programming approaches to Ms. Pac-Man (Alhejali and Lucas, 2010, 2011), but the results in Chapter 10 showed that these approaches were not as powerful as the modular neural networks developed in this dissertation. Additionally, Subramanian et al.'s agents did not learn particularly impressive ghost-eating behavior because no option for luring was learned. Creating a system that could learn such an option automatically is difficult, which is why many researchers prefer to provide a hierarchy of behaviors to the agent directly, or at least explicitly outline how such a hierarchy should be learned. Such approaches are described next.

11.2.2 Learned Subsumption Architectures

A more general-purpose method by Stone and Veloso (2000a) called Layered Learning takes the general idea of hierarchical, layered control, and adds to it the ability to learn the sub-controllers for individual layers. The design principles of a subsumption architecture are still used, but arbitrary ML algorithms can be used to learn behavior at each layer of a given control architecture. This hybrid approach led to victory in the 1999 RoboCup Soccer simulator league (Stone, 2000).

A similar approach by Togelius (2004) makes exclusive use of neuroevolution to learn both sub-behaviors and how to combine them in a hierarchy. This evolved subsumption architecture has been applied to games such as Unreal Tournament (van Hoorn et al., 2009) and EvoTanks (Thompson et al., 2009), which both require multimodal behavior.

A more recent approach by Lessin et al. (2013) evolves body morphologies for agents in addition to discovering complex behavior. As with Layered Learning and evolved subsumption architectures, this complex behavior has to be broken down into several simple sub-behaviors and learned from the bottom up. Simple behaviors like turning left and right, and moving forward are combined in a hierarchy to produce agents that chase after and attack targets.

These approaches are appealing from an engineering standpoint because the controllers are hierarchical, and each individual component has a clear purpose. Each sub-controller specializes in a particular task, so these approaches should work well in domains with isolated tasks. They also work in domains with interleaved or blended tasks because additional learning takes place at higher levels of the hierarchy. This extra learning allows controllers to determine how to switch between or even merge the behaviors of the sub-controllers.

However, much human expertise is still needed to divide a domain into sub-tasks properly; not only the hierarchy needs to be specified, but also the different training scenarios to develop and integrate all of the learned components. There is also the chance that a human-specified hierarchy will not leave room for the best possible behaviors. After all, the best Ms. Pac-Man behavior dedicated a module to luring, which was not anticipated in the human-specified task division.

Therefore, instead of manually combining learned components, methods have been developed in which the learned controllers have a built-in capacity to split up into separate modules, as will be discussed next.

11.3 Modular Architectures

Modular networks can be designed in a variety of ways, and through an analogy to the brain, modular structure is assumed to give rise to functional modularity. Such modules can be explicitly defined, or can arise as a result of a developmental process.

11.3.1 Explicit Modules

The research described in this section is most closely related to the methods developed in this dissertation. The purpose of preference neurons (Section 3.2.2) is to arbitrate between modules, whether in a fixed-module network or one evolved using Module Mutation (Section 3.2.3). Multi-task Learning (Section 3.2.1) also requires neural networks that have a modular architecture. Work by Calabretta et al. (2000), which uses a duplication operator to make module-like components, was also already mentioned in Section 3.2.3. These methods all focus on modules at the level of output neurons, because at this level it is clear how each module is influencing the behavior.

Other types of modular networks have also been used in combination with supervised learning. For example, Khare et al. (2005) co-evolved the connectivity of individual modules along with their organization within a combining network; the modules themselves were trained through supervised learning. The Neural-Based Learning Classifier System of Dam et al. (2008) also combines evolution and supervised learning, but instead of combining networks into a modular system, each network is associated with a particular region of the state space, and is trained only on data from this region. Whenever the system must make a decision, all networks whose region of expertise contains the current state combine into an ensemble that determines the system's decision.

Modular components can also be encapsulated and reused in systems not based on neural networks. Specifically, Genetic Programming (GP) is a way of generating executable program trees via evolution, for which the issue of modularity has received much attention. For example, Koza (1994) explored the benefits of defining modular structures (ADFs, or Automatically Defined Functions) that had the potential to be reused. Because the position of modules within the programming tree was fixed, this approach is similar to Module Mutation, but for a different type of evolved representation. Rosca and Ballard (1994) encouraged modularity in GP in a different way. Their Adaptive Representation (AR) approach identified sub-trees that had the potential to be good modules using block fitness functions (heuristics identifying useful modules). They later improved their approach with Adaptive Representation through Learning (ARL; Rosca and Ballard, 1996; Rosca, 1996), which culled modules from program trees based on differential parent/child fitness. This approach was applied to a simplified Pac-Man simulator (Section 8.1), and even discovered a module "used for attracting monsters." This module might have behaved similarly to the luring module discovered by networks in this dissertation, but the authors did not elaborate on what it did. However, despite the promise of this method, it made use of more sophisticated high-level actions (in contrast to the primitive actions of this dissertation), and has not been applied to the more challenging Ms. Pac-Man simulator used in this dissertation.

The approaches above explicitly encapsulate modules so that they are clearly identified, but other broader concepts of modularity have also been used in the literature. A module is less strictly defined in these contexts. Within neural networks, such a module is simply a cluster of tightly interconnected neurons with few connections to neurons in other clusters. This concept is particularly important in the study of Generative and Developmental Systems, discussed next.

11.3.2 Generative and Developmental Systems

Generative and Developmental Systems (GDS) evolve genotypes that are then used to create complex phenotypes, such as neural networks (Gruau, 1994; Stanley et al., 2009; Suchorzewski and Clune, 2011). Each evolved genotype is said to indirectly encode the phenotype that it generates. A touted (but disputed; Clune et al., 2010) benefit of such methods is their ability to create modular networks.

A particularly interesting method is HyperNEAT (D'Ambrosio and Stanley, 2007), which has been used to learn intelligent agent behavior in a few instances: Multi-agent foraging (D'Ambrosio and Stanley, 2008) and Keep-away in RoboCup soccer (Verbancsics and Stanley, 2011) are some notable examples. Perhaps the best example of multimodal behavior so far was done by D'Ambrosio et al. (2011), but the task switching in this paper was done manually, as with the Multitask Learning approach used in this dissertation. The question of whether HyperNEAT and similar methods are particularly good at learning multimodal behavior without a human-specified task division is still unanswered.

Clune et al. (2013) showed that modularity is encouraged in evolved neural networks if

connections have a fitness cost. This result was demonstrated using directly-encoded networks, but has consequences for indirectly-encoded networks produced by a GDS as well. Combining these elements into a single algorithm would likely encourage the discovery of functionally distinct network modules, but so far these techniques have not been used to evolve agent behavior.

Though these methods have promise, the results presented earlier in this dissertation demonstrate how modular networks that exhibit multimodal behavior can be evolved in a straightforward manner even without a generative process.

11.4 Conclusion

This chapter reviewed previous approaches to learning multimodal behavior. Despite the accomplishments of these other methods, the contributions made in this dissertation are distinct and advance the state of the art. The next chapter takes a high-level view of the results from the dissertation in the context of this larger body of research, and provides directions for future research.

Chapter 12

Discussion and Future Work

This chapter discusses and expands on the results from the dissertation in aggregate. First recommendations are made indicating which of the methods from this dissertation apply to each of the domain types defined in Chapter 4 best. Then each of these methods is discussed individually,

stance, if tasks are interleaved, then there is not an obvious way to train specific controllers for each task. A human-specified task division can still be used as part of a Multitask Learning approach in such domains, as was done in Imprison Ms. Pac-Man. This approach worked reasonably well, although it ignores how one task influences others. This blind spot is the reason that networks with preference neurons sometimes discovered a better task division involving a luring module. Therefore, a sensible recommendation for domains with interleaved tasks is to evolve networks with preference neurons that arbitrate between a number of fixed modules equal to the number of interleaved tasks. If the number of tasks is not clear, then Module Mutation can be used.

The hardest domains have blended tasks. The Battle Domain provides an example of TUG solving blended tasks with single-module networks. OL Ms. Pac-Man provides an example of conquering blended tasks without TUG, but with the help of modular networks, specifically those using preference neurons. The human-specified task divisions of Multitask Learning no longer worked in this difficult domain. ML Ms. Pac-Man showed the benefits of combining preference-neuron networks with TUG. Because Multitask Learning failed in Ms. Pac-Man, networks with preference neurons are more emphatically recommended for domains with blended tasks. If modular networks are already in use, then TUG is also recommended, since these methods work well in combination.

This section provided recommendations for each type of domain, but there are many interesting issues to discuss that are related to each specific method. Some of these issues generalize across domains. For example, split sensors are generally better in each type of domain, if they are available. This issue is discussed in detail next.

12.2 The Role of Sensors

The type of sensors used (conflict vs. split) was important in Ms. Pac-Man. Conflict sensors are more general, and impose less of a bias, but it is harder to learn multimodal behavior with them. Split sensors can provide a helpful bias, but with traditional single-module networks this bias will prevent evolution from discovering alternative, potentially better task divisions.

Although alternative sensor configurations were not used in the BREVE experiments, the choice of sensors affected the difficulty of learning intelligent behavior. Both Front/Back Ramming and Predator/Prey used conflict sensors; the exact same sensors were used in each isolated task of each domain. In contrast, the separate controllers of the Multinetwork approach could treat the bot sensors differently, as with split sensors, and therefore did well in both domains.

In contrast, all controllers in the Battle Domain had split sensors because extra sensors were added specifically to detect the bot's dangerous bat. However, simply having split sensors did not make the task easy to learn. Careful tuning of network structures and weights was still required to learn how to balance the competing influences of each type of sensor. Caution needed to be learned with respect to the bat so that monsters could avoid being hit, but boldness needed to be learned as well, so that the monsters would attack at opportune moments, despite sensing the oncoming swing

of the bat. TUG was vital in helping balance the influence of each type of sensor.

Even though it is clear that sensor design is important, it is not always clear what the best sensors are. The sensors used in this dissertation were mostly simple, but there were exceptions, such as Ms. Pac-Man's "Options From Next Junction" sensor. Other sensors, some even more sophisticated, were tried and discarded in preliminary experiments. For example, several sensors actually performed forward simulation, as in MCTS. Agents using these sensors were good at escaping ghosts. However, this approach had two problems: First, these sensors became the primary focus of learning, and as a result, the evolved agents had trouble learning how to chase edible ghosts, which is the key to maximizing scores. Second, simulation was costly, and as a result these experiments progressed very slowly. These problems are part of the reason that simpler sensors whose values could be quickly computed turned out better.

In complex environments, there may be a seemingly infinite set of reathe bestimeamm [(soph12(simp
n(fo-273([(which321(this)-322(hich(senso(and)-27334fnreuen-(slecd,)-379(Ho)240(,)-279(this)-273((96(clea-27 [(rs)-306(whose)]TJ 0Why

sensor copy that is added via mutation could have another randomly selected sensor act as a control switch: The value of the copied sensor would be either 0 or 1 if the control switch sensor is above or below some threshold, and the copied sensor's normal value would be returned otherwise. Another possibility would be to multiply the copied sensor value by the value of its control switch to get its final result.

However, these methods for creating split sensors still assume that the sensors have information required to determine the task division before any additional processing. For instance, with the threat/edible split learned in Ms. Pac-Man there were sensors that directly provided this information. However, the decision of when to use a luring module is more complicated, and seems to be based on information that is not directly available in the raw input sensors. Luring behavior was learned because it was associated with a network module dedicated to this behavior. Therefore, although adjusting the sensors in the ways proposed may make the discovery of multimodal behavior possible, providing extra network modules directly enables additional policies to be represented, each of which can exhibit a different mode of behavior.

12.3 Modular Networks

The experiments in Ms. Pac-Man showed that only modular networks could discover how to combine the behaviors of luring, chasing edible ghosts, and escaping threats in a competent manner. Single-module networks with split sensors learned the task division that was explicitly defined in how the sensors were split. Therefore, modular networks offer an advantage regardless of what sensors are used.

However, different types of modular networks were successful in different domains. MM(P) and MM(R) were equal in Front/Back Ramming, but both were outperformed by Multitask Learning in this domain. MM(R) was superior to both MM(P) and Multitask Learning in Predator/Prey. Evolving separate networks for each task also worked well in each of these domains. In Imprison Ms. Pac-Man with conflict sensors, Multitask Learning, MM(R), and MM(D) all reached roughly the same level of performance, but simply having two or three fixed modules with preference neurons performed even better (though the differences between fixed-module and Module Mutation networks were not statistically significant). The above domains either had isolated or interleaved tasks, and the results indicate that a variety of techniques work in these domains. Because the task division is clear with both isolated and interleaved tasks, there are many different ways to make successful use of multiple modules. Human-specified divisions and evolved divisions both work.

However, the hardest domain in which modular networks were evolved was the full Ms. Pac-Man game, which has blended tasks. In this domain, fixed-module networks using preference neurons were the best, with MM(D) close behind. However, the performance of Multitask Learning and MM(R) in this domain reached only the level of single-module networks. Even the Multitask Learning variant that dedicated a separate module to the blended task of facing threat and edible

ghosts at the same time did not do well. These results indicate that when tasks are blended, effective human-specified task divisions are harder to develop, and therefore less likely to be effective.

Despite variation across specific methods, modular networks in general have a clear advantage over traditional networks with a single module. Approaches using preference neurons have the most promise, because they can both learn human-specified divisions (like the threat/edible split in Imprison Ms. Pac-Man), and discover superior, unexpected task divisions (the luring module discovered in all Ms. Pac-Man experiments). The ability to discover new task divisions is particularly important in domains with blended tasks, because human-specified divisions are unreliable when the border between tasks is not clear.

However, it would be useful to have one method that works consistently well across multiple domains with isolated, interleaved, and blended tasks. The evolved networks of this dissertation can be improved by incorporating human knowledge while still allowing evolution to learn a task division, and with better ways of creating new modules whose behavior is learned by evolution.

12.3.1 Incorporating Human Knowledge

It is possible to use a human-specified task division from Multitask Learning, but still allow evolution to customize the task division. The first half of learning could be conducted using multitask networks, and then preference neurons could be added for each module and used from that point on to determine module usage. Alternatively, KBANN (Towell and Shavlik, 1994) could be used to translate the rules defining the multitask division into sub-networks that connect to preference neurons, so that learning starts with a human-specified task division programmed directly into the network, but evolution would still be capable of modifying that division.

Multitask Learning could also be combined with Module Mutation to create a multi-tiered control hierarchy. For example, the threat/edible division in Ms. Pac-Man is useful, but luring is a useful sub-task within the threat task. How could this sub-task be learned while still using knowledge of the threat/edible division? One way is to allow Module Mutation to function within each module defined by Multitask Learning. In this example, a single application of Module Mutation would add a module that was specifically associated with either the threat task or the edible task. If two threat-task modules were present, then when facing threat ghosts the network would first narrow its focus to these two modules according to the human-specified division, and then choose between these two modules based on preference neurons (Figure 12.1). The resulting controller would represent a two-level hierarchy, in which module arbitration at the higher level is based on a human-specified division, and arbitration at the lower level is done through preference neurons configured by evolution.

These approaches would allow evolution to refine a task division specified by a human, but such expert knowledge still introduces a bias. Such bias is the reason that this dissertation also focuses on modular networks created solely by evolution. The next section discusses ways of improving this approach.

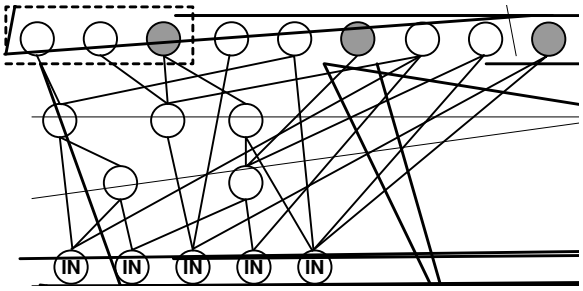


Figure 12.1: **Combination Multitask/Preference Neuron Hierarchy.** This network has two multitask modules specified by the output neurons enclosed in dashed rectangles. The left rectangle contains a single sub-module in a red box, so this one module will always be used whenever the multitask division chooses to use the left rectangle. The right rectangle contains two sub-modules, each in its own red box, so after the multitask division decides to use the right rectangle, it is then up to preference neurons to determine which of these two sub-modules define the behavior of the agent. More sub-modules could potentially be added to each top-level module via Module Mutation. This network architecture allows for a two-tiered hierarchy that leverages human knowledge while still allowing evolution to discover novel sub-tasks within each high-level task.

12.3.2 Learned Modules

Preference neurons provide the basis for evolution to discover how to use multiple modules on its own. Module Mutation allows evolution to discover how many modules to use as well, but simply having two fixed modules ended up working better than Module Mutation in all variants of Ms. Pac-Man. This result could simply be a peculiar property of this particular domain. After all, successful Module Mutation solutions in FBR used three modules extensively, in addition to occasionally using others. Still, given enough time, Module Mutation should be able to rise to the same level of performance as the fixed module networks.

Each form of Module Mutation connects new modules to the network in a different way, and each approach has pros and cons. MM(P) elaborates on pre-existing structures to refine the behavior currently exhibited, but links all modules in a way that makes it difficult for them to specialize. MM(R) quickly explores new, novel behaviors, but since many such behaviors may be bad, it can have a hard time creating good modules on which to elaborate. MM(D) creates new modules that do not actually change the behavior of the agent, which makes it easy for new modules to survive into the next generation, but potentially slows down the search process. Other forms of Module Mutation are no doubt possible.

However, there is no reason that these different approaches could not be combined. If each type of Module Mutation had a chance of occurring, then whichever type best fit the space of behaviors for the domain would likely win out. Of course, having so many forms of Module Mutation available could also result in module bloat, or simply slow down evolution by giving networks too much new structure to optimize at once. Whether such an approach would be successful is ulti-

mately an empirical question.

Another option for incorporating new modules is to start with plenty of extra modules, but leave them disconnected. In other words, networks would start with several free-floating output neurons that could be incorporated into the network via regular link mutations. This approach to adding modules might impose less of an initial bias than any of the Module Mutation approaches, which could in turn help new behaviors emerge gradually. However, one potential problem with this approach is that links to the policy neurons of a module whose preference neuron remains disconnected will not affect network behavior at all. This problem exists with all preference-neuron-based networks, but seems more likely to cause issues when the modules are initially disconnected. A module with essentially random behavior could be suddenly connected when a link is added to its preference neuron. However, such modules would likely be no worse than those created by MM(R).

In fact, MM(R) worked best in PP while using feature selection, meaning that new MM(R) modules had only a single link per neuron. Perhaps new MM(R) modules are more likely to succeed when they are less complicated.

Another concept that could be useful in the construction of modular networks is a freeze operation. The Cascade Correlation supervised learning network architecture uses this idea (Fahlman and Lebiere, 1990): New components are added to a network incrementally, trained, and then frozen so that they do not change when subsequent components are trained. This concept could be extended to learning multimodal behavior with modular networks. If a network module already exhibits a useful mode of behavior, then freezing the neurons and links that make it up will protect that behavior, while letting other modules improve. The trick is knowing which components to freeze and when.

The manner in which MM(P) adds new modules is similar to Cascade Correlation: Old structure leads into new structure, refining what is already there. Perhaps everything but the new module could be frozen whenever MM(P) is performed. Such an approach might also make sense with MM(D), since the original behavior would be preserved while exploring ways to improve it in a different module. A more general approach for modular networks is to have a mutation operator that freezes all components associated with a particular module (Figure 12.2). Care would need to be taken to assure that some portion of the network is always unfrozen. A corresponding thaw operator could be used to unfreeze modules so that learning can focus alternately on different modules. It might be easier to search the space of policies if evolution is restricted to altering one module at a time. If certain modules are linked to particular objectives, then it might even make sense to synchronize the freezing and thawing of particular modules with the activation and deactivation of particular objectives. Such objective management could be incorporated into TUG, which is discussed next.

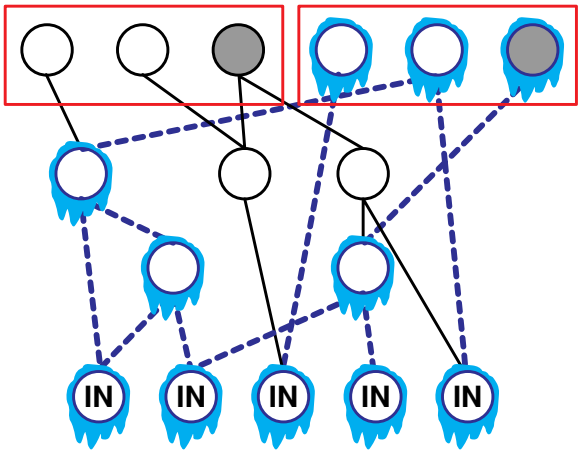


Figure 12.2: **Freeze Mutation.** Inputs in this network are labelled at the bottom and each output module is contained in its own red box. A mutation operator picks a random module (module on the right in this example), and backtracks through the network, freezing every neuron and link that leads into the frozen module. A frozen link (dashed) cannot have its weight changed, and a frozen neuron (icy edges) cannot accept new incoming links. These restrictions assure that the behavior of the frozen module cannot change. However, new links can still exit frozen neurons, so it is still possible to build on existing components in order to change the behavior of modules that are not frozen. Such a freeze mutation could protect good behaviors while evolution continues to refine other behaviors.

12.4 Enhancing TUG

TUG is able to direct multiobjective search in a way that pushes it towards the best solutions. TUG was successful in the Battle Domain, as well as Ms. Pac-Man when combined with modular networks. However, when combined with single-module networks, the result was mixed: In some runs the performance shot up quickly, but in other runs it flatlined early and remained low. This phenomenon was also seen in some preliminary experiments. The problem resulted from the fact that although different behaviors are required to increase the `Pill Score` and the `Ghost Score`, they are linked in the sense that more ghosts cannot be eaten without visiting more levels, which can only be done by eating more pills. The challenge is to figure out how stagnation can be prevented despite this problem.

First, it should be pointed out that any starting population that is large enough should have the potential to achieve high scores via skilled multimodal behavior. However, some runs do worse than others because useful stepping stones on the way to the best solutions get discarded out of hand in favor of solutions that have higher scores, but are doomed to lead the search towards local optima. This problem is referred to as deception (Grefenstette, 1992).

Many methods have been designed to deal with deception, including Fitness-sharing (Sareni

and Krähenbühl, 1998), Novelty Search (Lehman and Stanley, 2011), and Behavioral Diversity (Mouret and Doncieux, 2009). Essentially, all of these methods work by selecting solutions for the next generation that are not strictly favored by the fitness criteria. TUG does so as well, because it sometimes ignores certain objectives.

It is possible that TUG could be improved by combining it with some of the above methods. Behavioral Diversity is particularly appealing because it is already designed to work in a multiobjective setting: This method includes an extra objective in the search process that favors behaviorally distinct solutions (for some human-specified behavior characterization). This objective could be managed by TUG's existing mechanisms. Alternatively, this objective could be given special privilege, so that it functions regardless of which other objectives TUG is using, or perhaps extra diversity should only be injected when TUG performance stagnates.

Preventing stagnation in general would improve TUG, but detecting it is a challenge. By the time stagnation is obvious, it is likely that the population has already lost candidate solutions that could lead it out of the local optimum in which it is trapped. Stagnation needs to be detected preemptively, so that the right objectives are activated in time.

However, the stagnation problem could also be solved by rewinding evolution. If previous populations were saved, then at the point when stagnation is detected the search could go back to a previous generation before the stagnation occurred. To prevent stagnation from occurring again, the search needs to go in a different direction. Simply supplying different random numbers could be enough in some cases, but changing which objectives are used for selection addresses the problem more directly. Specifically, although turning off objectives sometimes helps evolution discover good behavior faster, reactivating all objectives after rewinding from a stagnation event is more likely to assure a proper diversity of solutions within the population.

An alternative approach to improving TUG is to change what it means for a goal to be considered achieved. In the version of TUG presented in this dissertation, goals are achieved when the average population performance persists above the goal value. However, other definitions of goal achievement could be superior. Perhaps a different statistic, such as the median or upper quartile would be better, especially if the distribution of scores in the population does not follow a normal distribution. Even if average performance is still used to determine goal achievement, additional restrictions could be used to assure that goals do not increase unless the population is steadily making progress. For example, in addition to achieving all goals, the population could be required to dominate a steadily increasing portion of objective space. That is, the hypervolume metric could be required to increase as goals increase, which would discourage the easier objectives from being completely ignored.

The ideas discussed so far have mostly been improvements and variations on the algorithms used throughout this dissertation. The next section proposes ways of learning multimodal behavior with a different approach.

12.5 Multimodal Behavior with HyperNEAT

Neural networks were used to define control policies in this dissertation because they are universal function approximators with a good history of solving challenging control problems. The representation used to evolve these networks, the genotype, maps directly to the space of networks, the phenotype space. Every component of the genotype, on which mutations and crossover operate, directly corresponds to a part of the phenotype. This approach is obviously successful, but it also has its drawbacks.

Mutation operators for direct encodings generally make only small, local changes to genotypes. The Module Mutation operations defined in this dissertation are unusual in that they introduce lots of new structure with a single operation. However, small edits are still required to fine-tune new modules. Also, there is no operation that makes large edits to modules that already exist. For example, MM(D) copies an existing module so that a preexisting behavior can diverge in two different directions, but after that split, adjustments that would be beneficial to both modules must be performed individually to each module, which is unlikely to occur via random mutations.

Generative and Developmental Systems (GDS), which use indirect encodings to produce phenotypes, can potentially overcome this problem. In such systems the genotype represents a process or a program that is executed to create the final phenotype. Parts of an indirectly encoded genotype are generally reused several times while creating the phenotype, which means that a single change to the genotype can affect several parts of the phenotype easily. In fact, one of the main benefits of indirect encodings is that they give rise to regularity and modularity (Stanley and Miikkulainen, 2003), which are qualities well-suited to the generation of multimodal behavior.

A GDS specifically tailored to creating neural network phenotypes is HyperNEAT (D'Ambrosio and Stanley, 2007). HyperNEAT evolves one type of neural network, called a Compositional Pattern-Producing Network (CPPN; Stanley, 2006), and uses it to create another neural network that is then evaluated in a task. Like all neural networks, CPPNs are function approximators, but they are special in that they can contain activation functions like sine, cosine, and Gaussian functions, which encourage regularity and symmetry in the function output. Function trees (evolved by Genetic Programming) can also be used in place of CPPNs if these special functions are included as building blocks (Buk et al., 2009). Regardless of which function approximator is used, these methods generate neural networks in two steps.

First, a general connectivity pattern, called a substrate, is defined for a particular problem. The substrate defines the number of inputs and outputs the network has, but also determines the positions of all potential neurons. Second, the CPPN is queried with the coordinate positions of each pair of potential neurons in the substrate to get an output that indicates whether the neurons are linked and what the weight is. If the substrate is defined in a clever way, a single genotype can be used to create several different but related neural network controllers. Section 11.3.2 already mentioned work in a multi-agent domain where each distinct team member was spawned by the same source CPPN (D'Ambrosio and Stanley, 2008). This result was accomplished by situating

the different neural controllers in different positions within the substrate according to each agent's starting position in the environment. By linking the geometry of the environment to the geometry of the substrate, multiple distinct cooperating controllers are produced by a single genotype.

Later work (D'Ambrosio et al., 2011) extends this idea to allow agents to choose a particular policy depending on the current state the agent is in. Creating agents that use different policies in different situations is precisely what the methods developed in this dissertation are designed to do, so D'Ambrosio et al.'s work is a step in the same direction. However, the multiple policies possessed by these agents were arbitrated according to a human-specified signal, as in Multitask Learning. The next logical step would be to allow arbitration between multiple policies based on preference neurons, or some other learned signal. This extension would presumably allow HyperNEAT to discover novel task divisions, such as the luring behavior discovered in Ms. Pac-Man.

Allowing HyperNEAT to create policies that choose when to be active is similar to having networks with multiple fixed modules arbitrated by preference neurons. Such networks did very well in Ms. Pac-Man, but a fully general learning method should also be able to decide how many different modules to have, as Module Mutation does. Adding this extension to HyperNEAT would require an extra level of ingenuity. For instance, a dedicated CPPN output or an additional evolved parameter could be added. However, even if this issue is resolved, the harder decision to make is how to situate new policies within the substrate geometry.

In fact, the geometric focus of HyperNEAT is both a strength and a disadvantage. Nearly all problems involving agent behavior require some degree of geometric awareness. HyperNEAT has an advantage when dealing with geometrically organized sensors because it can easily incorporate concepts of symmetry and repetition. However, HyperNEAT requires that all sensors and network outputs are geometrically embedded in the network substrate. The sensors in the HyperNEAT experiments discussed so far were simple rangefinders, so sensors in the left of the substrate corresponded to sensors on the left side of the agent, and vice versa for sensors on the right. If these sensors are sufficient to learn interesting behavior, then this restriction is not a problem, but many of the sensors used in the BREVE (e.g. "Any Monster Dealt Damage") and Ms. Pac-Man (e.g. "Proportion Edible Time") experiments of this dissertation do not have an obvious geometric interpretation. A possible solution is to assign each such sensor its own spatial dimension within the substrate, so that geometry should not matter for them. However, all existing work with HyperNEAT focuses on problem geometry, so it is unclear whether this approach would work with a large number of sensors.

Because HyperNEAT is a relatively new algorithm, researchers are still learning how to apply it. Currently, the question of how to design a substrate to suit a particular problem is tricky, but there are some general guiding principles. The question of how to situate multiple policies in a substrate is even trickier. In D'Ambrosio et al.'s multi-policy example above, the two behaviors were advancing into and evacuating from a maze, and an extra substrate dimension was defined for each separate policy. The corresponding substrate inputs for these policies were 1 and -1. This design makes sense because evacuating is the opposite of advancing, so the two policies can be

thought of as contrasting with each other. However, in Ms. Pac-Man, what would be the geometric distinction between a policy for luring and a policy for collecting pills? It is hard to imagine these behaviors existing along the same dimension, let alone to imagine what dimensions exist to define Ms. Pac-Man behavior. Perhaps as practitioners gain more experience with HyperNEAT and similar methods, ideas for how to spatially embed multiple policies for a single agent will become more common. In the meantime, the methods introduced in this dissertation for learning directly encoded modules provide an obvious and effective way of learning multimodal behavior.

12.6 Conclusion

This chapter presented ideas for future research in the field of discovering multimodal behavior. Several ideas for improving the methods in this dissertation were described. Specifically, ways of combining and improving the various Module Mutation methods were discussed, as well as variations on TUG that could encourage it to succeed more consistently even with non-modular networks.

Several new ideas were also proposed. Ways of automatically adjusting the sensors to make the discovery of multimodal behavior easier were considered. Other ways of creating new output modules were proposed, including ways for human knowledge of how to divide a task to be supplemented by what evolution discovers. Finally, ways in which HyperNEAT could be used to generate multiple policies were considered, with a specific focus on what has already been tried, and what problems must be solved in order to accomplish more. The ideas presented in this chapter should thus lead to interesting projects in the future.

Chapter 13

Conclusion

This dissertation focuses on the discovery of multimodal behavior. Such behavior is exhibited by all intelligent animals, but not by many artificial agents that learn their behavior through interaction with the environment. When multimodal behavior has been learned in previous research, it has often depended on human-specified task divisions, or other extensive domain knowledge. This dissertation introduced methods that learn novel task divisions on their own, even with general conflict sensors. This final chapter reviews these contributions, and assesses their potential future impact.

13.1 Contributions

The main technical contributions of the dissertation were introduced in Chapter 3. The first contribution is an understanding of the distinction between split and conflict sensors, and how they affect the learning of multimodal behavior. The second contribution consists of several ways of allowing networks to use multiple output modules. In particular, preference neurons were designed as a way to allow networks to learn their own task division, and Module Mutation combined with preference neurons is a way to allow evolution to discover how many modules to use. These new methods that learn their own task division were contrasted with Multitask Learning, an established approach that depends on a human-specified task division. Interestingly, automatically discovered task divisions were found to be better than human-specified divisions in all but one domain (FBR). The third contribution is TUG, which enhances multiobjective evolution by focusing on the objectives that need it most.

Chapter 4 provided an overview of different types of domains requiring multimodal behavior. In particular, different ways of splitting up tasks in a domain were identified: isolated tasks, interleaved tasks, and blended tasks. Isolated tasks are completely separate, but a single agent is expected to accomplish all of them. Interleaved tasks are more strongly coupled, in that an agent switches between tasks in a single evaluation. As a result, actions in one task have consequences

for other tasks. Blended tasks are intertwined to the point that the boundary between them is unclear. There may be certain moments during evaluation when it is possible to clearly identify the task, but there are also periods of time when the distinction is blurred, such that multiple tasks are occurring at the same time. In such situations, it is particularly important for intelligent agents to decide what the current task is so they know how best to behave. Experiments were conducted in several domains that exemplify these different task divisions.

The first three domains were designed in the BREVE simulator (Chapter 5), which made it possible to evaluate each method in domains with isolated and blended tasks. The first two BREVE domains were Front/Back Ramming (FBR) and Predator/Prey (PP). These domains both consist of isolated tasks, and modular networks were superior to single-module networks in both domains (Chapter 6). Specifically, Multitask Learning did well in FBR because the two tasks are equally difficult, and having separate policies for each task made it easy to have a different behavior for each one. Module Mutation also did well, but had to overcome the harder challenge of determining what the current task was, because the sensors did not provide this information. Module Mutation networks determined the current task by using recurrent connections to remember how they received damage. This clever strategy worked, but cost some damage. In PP the relative difficulty of the tasks is skewed, which confused Multitask Learning and caused it to fail in this domain. However, Module Mutation Random did well in this domain even though it did not sense in what task it was directly.

The third BREVE domain was the Battle Domain (BD), which has blended attack and defense tasks. Networks evolved in this domain had only a single module, but good behavior could still be achieved by using TUG (Chapter 7). These networks had the capacity to deal with the separate tasks of this domain because they used split sensors, but TUG was still required to help the networks discover intelligent behaviors needed to succeed.

Having established the basic characteristics of these methods in BREVE simulations, they were scaled up to the real-world game of Ms. Pac-Man in Chapters 9 and 10. In Chapter 9, a simplified Imprison version of Ms. Pac-Man was created in order to test the methods in a domain with interleaved tasks: This slight variant on the original arcade game creates a clear distinction between threat and edible ghost tasks. Therefore, Multitask Learning does well in this domain, as do single-module networks using split sensors (the sensors create the same task division). This same task division is also learned by approaches with preference neurons. However, a more interesting result is that the best approaches discover a task division that is distinctly different from human task divisions, and performs better: It is based on luring. Only modular approaches with preference neurons can discover a luring module, and it results in the best scores.

Chapter 10 scales up to the full game of Ms. Pac-Man in two steps: First, Ms. Pac-Man had just one life in order to quickly test a variety of methods in a domain that is just one step removed from Imprison Ms. Pac-Man. Second, Ms. Pac-Man had multiple lives, as in the original game, in order to scale up to the domain used by other researchers in the literature. In these experiments,

the best performance is still achieved by modular networks that use preference neurons to discover a luring module. However, because the full game has blended tasks (i.e. the threat and edible ghosts can be present at the same time), human-specified task divisions break down: Multitask Learning performs poorly in this domain, and even with split sensors, networks with one module are significantly worse than networks with two modules using preference neurons. In the evaluations with multiple lives, which are more forgiving, it turned out that TUG could be effectively used to improve the behavior. Thus, combining modular network architectures with fitness shaping via TUG lead to skilled multimodal behavior that surpassed all previous attempts to learn behavior in Ms. Pac-Man.

13.2 Conclusion

Intelligent agents can exhibit multiple modes of behavior. However, artificial agents are typically optimized to focus on a single narrowly defined task. When artificial agents do exhibit multimodal behavior, it is often because a human designer has created or learned multiple policies and fit them into a hierarchy. This dissertation represents a step away from such human-specified task divisions, and a step towards allowing learning methods like neuroevolution to discover for themselves how to learn complex multimodal behavior.

These methods were successful in several artificial worlds, including the popular classic arcade game of Ms. Pac-Man. Therefore, these methods should prove immediately useful in other artificial worlds, such as training simulators, video games, and Artificial Life simulations. However, multimodal behavior is also needed in robotic systems, so these methods will hopefully be useful in real-world agents as well. Multimodal behavior can be evolved in simulation and then adapted to a real robot, or in some cases learned on the robot itself. Applying this research in these ways should lead to a future with intelligent and versatile artificial agents that can serve as companions and assistants to humans in various ways.

Bibliography

Alhejali, A. M., and Lucas, S. M. (2010). Evolving diverse Ms. Pac-Man playing agents using genetic programming. In *UK Workshop on Computational Intelligence (UKCI 2010)*, 1–6.

Alhejali, A. M., and Lucas, S. M. (2011). Using a Training Camp with Genetic Programming to evolve Ms Pac-Man agents. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, 118–125. IEEE.

Alhejali, A. M., and Lucas, S. M. (2013). Using Genetic Programming to evolve heuristics for a Monte Carlo Tree Search Ms Pac-Man agent. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2013)*, 1–8. IEEE.

Alur, R., Das, A. K., Esposito, J. M., Fierro, R. B., Grudic, G. Z., Hur, Y., Kumar, V., Lee, I., Lee, J. P., Ostrowski, J. P., Pappas, G. J., Southall, B., Spletzer, J. R., and Taylor, C. J. (2000). A framework and architecture for multirobot coordination. In Rus, D., and Singh, S., editors, *ISER*, vol. 271 of *Lecture Notes in Control and Information Sciences*, 303–312. Springer.

Bakker, B., and Heskes, T. (2003). Task Clustering and Gating for Bayesian Multitask Learning. *Journal of Machine Learning Research*, 4:83–99.

Barto, A. G., and Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(1–2):41–77.

Bellman, R. (1957). *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press. First edition.

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement Learning to Train Ms. Pac-Man Using Higher-order Action-relative Inputs. In *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*.

Brandstetter, M. F., and Ahmadi, S. (2012). Reactive control of Ms. Pac Man using information retrieval based on Genetic Programming. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2012)*, 250–256. IEEE.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(10).

Bryant, B. D., and Miikkulainen, R. (2006). Evolving Stochastic Controller Networks for Intelligent Game Agents. In *Congress on Evolutionary Computation*. Piscataway, NJ: IEEE.

Buehler, M., Iagnemma, K., and Singh, S. (2009). *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Springer Publishing Company, Incorporated. First edition.

Buk, Z., Koutník, J., and Snorek, M. (2009). NEAT in HyperNEAT substituted with Genetic Programming. In *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms (ICANNGA)*, 243–252.

Burrow, P., and Lucas, S. M. (2009). Evolution versus Temporal Difference Learning for learning to play Ms. Pac-Man. In Lanzi, P. L., editor, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2009)*, 53–60. IEEE.

Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. (2000). Duplication of Modules Facilitates the Evolution of Functional Specialization. *Artificial Life*, 6(1):69–84.

Cardamone, L., Loiacono, D., and Lanzi, P. L. (2009). Evolving Competitive Car Controllers for Racing Games with Neuroevolution. In *Genetic and Evolutionary Computation Conference*, 1179–1186.

Caruana, R. A. (1993). Multitask Learning: A knowledge-based source of inductive bias. In *International Conference on Machine Learning*, 41–48.

Caruana, R. A. (1997). *Multitask Learning*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213.

Chen, X., Stone, P., Sucar, L. E., and van der Zant, T., editors (2013). *RoboCup 2012: Robot Soccer World Cup XVI [papers from the 16th Annual RoboCup International Symposium, Mexico City, Mexico, June 18-24, 2012]*, vol. 7500 of *Lecture Notes in Computer Science*. Springer.

Cheng, X., Shen, J., Liu, H., and Gu, G. (2007). Multi-robot Cooperation Based on Hierarchical Reinforcement Learning. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part III*, 90–97. Berlin, Heidelberg: Springer-Verlag.

Choi, J., and Kim, K.-E. (2013). Bayesian Nonparametric Feature Construction for Inverse Reinforcement Learning. In Rossi, F., editor, *International Joint Conferences on Artificial Intelligence*. IJCAI/AAAI.

Clune, J., Beckmann, B. E., McKinley, P. K., and Ofria, C. (2010). Investigating whether HyperNEAT produces modular neural networks. In *Genetic and Evolutionary Computation Conference*, 635–642.

Clune, J., Mouret, J.-B., and Lipson, H. (2013). The evolutionary origins of modularity. *Proceedings of the Royal Society B: Biological Sciences*, 280(1755):20122863–20122863.

Coello, C. A. C. (1999). A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1(3):129–156.

Collobert, R., and Weston, J. (2008). A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, 160–167. New York, NY, USA: ACM.

Corne, D. W., Jerram, N. R., Knowles, J. D., and Oates, M. J. (2001). PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Genetic and Evolutionary Computation Conference*, 283–290.

Corne, D. W., Knowles, J. D., and Oates, M. J. (2000). The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization. In *Parallel Problem Solving from Nature*, 839–848. Springer.

Dam, H. H., Abbass, H. A., and Lokan, C. (2008). Neural-Based Learning Classifier Systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):26–39.

D'Ambrosio, D. B., Lehman, J., Risi, S., and Stanley, K. O. (2011). Task switching in multirobot learning through indirect encoding. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011*, 2802–2809.

D'Ambrosio, D. B., and Stanley, K. O. (2007). A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, 974–981. New York, NY, USA: ACM.

D'Ambrosio, D. B., and Stanley, K. O. (2008). Generative encoding for multiagent learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

de Boer, P.-T., Kroese, D. P., Mannor, S., and Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Evolutionary Computation*, 6:182–197.

Deisenroth, M., Calandra, R., Seyfarth, A., and Peters, J. (2012). Toward Fast Policy Search for Learning Legged Locomotion. In *Proceedings of the International Conference on Robot Systems (IROS)*.

DeNero, J., and Klein, D. (2010). Teaching Introductory Artificial Intelligence with Pac-Man. In *Proceedings of the Symposium on Educational Advances in Artificial Intelligence (EAAI)*.

Dietterich, T. G. (1998). The MAXQ Method for Hierarchical Reinforcement Learning. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML98)*.

Diller, D. E., Ferguson, W., Leung, A. M., Benyo, B., and Foley, D. (2004). Behavior modeling in commercial games. In *Behavior Representation in Modeling and Simulation (BRIMS)*.

Fahlman, S. E., and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, 524–532. San Francisco: Morgan Kaufmann.

Floreano, D., and Urzelai, J. (2000). Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13:431–4434.

Foderaro, G., Swingler, A., and Ferrari, S. (2012). A model-based cell decomposition approach to on-line pursuit-evasion path planning and the video game Ms. Pac-Man. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2012)*, 281–287. IEEE.

Fonseca, C. M., and Fleming, P. J. (1995). An Overview of Evolutionary Algorithms in Multiobjective Optimization. *Evolutionary Computation*, 3:1–16.

Gallagher, M., and Ledwich, M. (2007). Evolving Pac-Man Players: Can We Learn from Raw Input? In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2007)*, 282–287. IEEE.

Gaskett, C., Wettergreen, D., and Zelinsky, A. (1999). Q-learning in continuous state and action spaces. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence*, AI '99, 417–428. London, UK, UK: Springer-Verlag.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2006). Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning*, 654–662. Berlin: Springer.

Grefenstette, J. J. (1992). Deception considered harmful. In *Foundations of Genetic Algorithms 2*, 75–91. Morgan Kaufmann.

Gruau, F. (1994). Automatic definition of modular neural networks. *Adaptive Behavior*, 3(2):151–183.

Handa, H., and Isozaki, M. (2008). Evolutionary fuzzy systems for generating better Ms.PacMan players. In *FUZZ-IEEE*, 2182–2185. IEEE.

Haykin, S. (1999). *Neural Networks, A Comprehensive Foundation*. Upper Saddle River, New Jersey: Prentice Hall.

Hengst, B. (2002). Discovering Hierarchy in Reinforcement Learning with HEXQ. In *International Conference on Machine Learning*, 243–250. Morgan Kaufmann.

Hingston, P. (2012). *Believable Bots: Can Computers Play Like People?*. Springer Berlin Heidelberg.

Huber, M., and Grupen, R. A. (1997). A feedback control structure for on-line learning tasks. *Robotics and Autonomous Systems*, 22:22–23.

Ikehata, N., and Ito, T. (2011). Monte-Carlo Tree Search in Ms. Pac-Man. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, 39–46. IEEE.

Isla, D. (2005). Managing Complexity in the Halo 2 AI System. In *Proceedings of the Game Developers Conference*. San Francisco, CA.

Kalyanakrishnan, S., and Stone, P. (2009). An Empirical Analysis of Value Function-Based and Policy Search Reinforcement Learning. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Kang, Z., Grauman, K., and Sha, F. (2011). Learning with whom to share in multi-task feature learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 521–528.

Khare, V., Yao, X., Sendhoff, B., Jin, Y., and Wersing, H. (2005). Co-evolutionary Modular Neural Networks for Automatic Problem Decomposition. In *Congress on Evolutionary Computation*, vol. 3, 2691–2698.

Klein, J. (2003). BREVE: A 3D Environment for the Simulation of Decentralized Systems and Artificial Life. *Artificial Life*, 329–334.

Knowles, J., Thiele, L., and Zitzler, E. (2006). A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, TIK, ETH Zurich.

Kohl, N., and Miikkulainen, R. (2009). Evolving Neural Networks for Strategic Decision-Making Problems. *Neural Networks, Special issue on Goal-Directed Neural Systems*.

Kolter, J. Z., and Ng, A. Y. (2009). Regularization and Feature Selection in Least-squares Temporal Difference Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, 521–528. New York, NY, USA: ACM.

Konidaris, G., and Barto, A. (2009). Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, 1015–1023.

Konidaris, G., Kuindersma, S., Barto, A., and Grupen, R. (2010). Constructing Skill Trees For Reinforcement Learning Agents From Demonstration Trajectories. In *In Advances in Neural Information Processing Systems (NIPS)*.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge Massachusetts: MIT Press.

Lehman, J., and Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223.

Lessin, D., Fussell, D., and Miikkulainen, R. (2013). Open-ended behavioral complexity for evolved virtual creatures. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2013*.

Levine, S., Popovic, Z., and Koltun, V. (2010). Feature Construction For Inverse Reinforcement Learning. *Advances in Neural Information Processing Systems*, 23.

Liu, Q., Liao, X., Li, H., Stack, J., and Carin, L. (2009). Semisupervised Multitask Learning. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(6):1074–1086.

Lochtefeld, D., and Ciarallo, F. (2012). Multiobjectivization via helper-objectives with the tunable objectives problem. *Evolutionary Computation, IEEE Transactions on*, 16(3):373–390.

Lucas, S. M. (2005). Evolving a neural network location evaluator to play Ms. Pac-Man. In Kendall, G., and Lucas, S., editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2005)*, 203–210. IEEE.

Martín, E., Martínez, M., Recio, G., and Sáez, Y. (2010). Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2010)*, 458–464.

Michie, D., and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E., and Michie, D., editors, *Machine Intelligence*. Edinburgh, UK: Oliver and Boyd.

Miikkulainen, R., Feasley, E., Johnson, L., Karpov, I., Rajagopalan, P., Rawal, A., and Tansey, W. (2012). Multiagent learning through neuroevolution. In et al., J. L., editor, *Advances in Computational Intelligence*, vol. LNCS 7311, 24–46. Berlin, Heidelberg:: Springer.

Moriarty, D. E., Schultz, A. C., and Grefenstette, J. J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276.

Mouret, J.-B., and Doncieux, S. (2009). Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In *Genetic and Evolutionary Computation Conference*, 627–634. ACM.

Murphy, R. R., editor (2014). *Disaster Robotics*. MIT Press.

Murphy, R. R., Tadokoro, S., Nardi, D., Jacoff, A., Fiorini, P., Choset, H., and Erkmen, A. M. (2008). Search and Rescue Robotics. In *Springer Handbook of Robotics*, 1151–1173. Springer.

National Highway Traffic Safety Administration (2008). National motor vehicle crash causation survey: Report to congress. Technical Report DOT HS 811 059, U.S. Department of Transportation (NHTSA). Accessed Feb. 11th, 2014.

Nguyen, T., Li, Z., Silander, T., and Leong, T. Y. (2013). Online Feature Selection for Model-based Reinforcement Learning. In Dasgupta, S., and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 498–506. JMLR Workshop and Conference Proceedings.

Pepels, T., and Winands, M. H. M. (2012). Enhancements for Monte-Carlo Tree Search in Ms Pac-Man. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2012)*, 265–272. IEEE.

Prescott, T. J., Redgrave, P., and Gurney, K. (1999). Layered control architectures in robots and vertebrates. *Adaptive Behavior*, 7(1):99–127.

Rajagopalan, P., Rawal, A., Miikkulainen, R., Wiseman, M. A., and Holekamp, K. E. (2011). The Role of Reward Structure, Coordination Mechanism and Net Return in the Evolution of Cooperation. In *Computational Intelligence and Games*. Seoul, South Korea.

Recio, G., Martín, E., Estébanez, C., and Sáez, Y. (2012). Antbot: Ant colonies for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):295–308.

Robles, D., and Lucas, S. M. (2009). A simple tree search method for playing Ms. Pac-Man. In Lanzi, P. L., editor, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2009)*, 249–255. IEEE.

Rosca, J. (1996). Generality Versus Size in Genetic Programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 381–387. Stanford University, CA, USA: MIT Press.

Rosca, J. P., and Ballard, D. H. (1994). Genetic Programming with Adaptive Representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA.

Rosca, J. P., and Ballard, D. H. (1996). Discovery of Subroutines in Genetic Programming. In Angeline, P. J., and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 9, 177–202. Cambridge, MA, USA: MIT Press.

Samothrakis, S., Robles, D., and Lucas, S. M. (2011). Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154.

Sareni, B., and Krähenbühl, L. (1998). Fitness sharing and niching methods revisited.. *IEEE Trans. Evolutionary Computation*, 2(3):97–106.

Schrum, J., and Miikkulainen, R. (2008). Constructing Complex NPC Behavior via Multi-Objective Neuroevolution. In *Artificial Intelligence and Interactive Digital Entertainment*.

Schrum, J., and Miikkulainen, R. (2009). Evolving Multi-modal Behavior in NPCs. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2009)*, 325–332.

Schrum, J., and Miikkulainen, R. (2010). Evolving Agent Behavior In Multiobjective Domains Using Fitness-Based Shaping. In *Genetic and Evolutionary Computation Conference*.

Schrum, J., and Miikkulainen, R. (2011). Evolving Multimodal Networks for Multitask Games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, 102–109. Seoul, South Korea: IEEE.

Schrum, J., and Miikkulainen, R. (2012). Evolving Multimodal Networks for Multitask Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):94–111.

Stanley, K. O. (2003). *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX.

Stanley, K. O. (2006). Exploiting Regularity without Development. In *Proceedings of the AAAI Fall Symposium on Developmental Systems*. Menlo Park, CA: AAAI Press.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. In *Proceedings of the 2003 Congress on Evolutionary Computation*. Piscataway, NJ: IEEE.

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A Hypercube-Based encoding for evolving Large-Scale neural networks. *Artificial Life*, 15(2):185–212.

Stanley, K. O., and Miikkulainen, R. (2002). Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10:99–127.

Stanley, K. O., and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

Stolle, M., and Precup, D. (2002). Learning Options in Reinforcement Learning. In Koenig, S., and Holte, R. C., editors, *Abstraction, Reformulation, and Approximation*, vol. 2371 of *Lecture Notes in Computer Science*, 212–223. Springer Berlin Heidelberg.

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. Cambridge, MA: MIT Press.

Stone, P., and Veloso, M. (2000a). Layered Learning. In *Proceedings of the Eleventh European Conference on Machine Learning*, 369–381. Springer Verlag.

Stone, P., and Veloso, M. (2000b). Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robotics*, 8(3).

Subramanian, K., Isbell, C., and Thomaz, A. (2011). Learning Options through Human Interaction. In *Workshop on Agents Learning Interactively from Human Teachers at IJCAI*.

Suchorzewski, M., and Clune, J. (2011). A novel generative encoding for evolving modular, regular and scalable networks. In *Genetic and Evolutionary Computation Conference*, 1523–1530.

Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Sutton, R. S., Precup, D., and Singh, S. P. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2):181–211.

Svensson, J., and Johansson, S. J. (2012). Influence Map-based controllers for Ms. PacMan and the ghosts. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2012)*, 257–264. IEEE.

Szita, I., and Lőrincz, A. (2006). Learning tetris using the noisy cross-entropy method. *Neural Computation*, 18.

Szita, I., and Lőrincz, A. (2007). Learning to play using low-complexity rule-based policies: Illustrations through Ms. Pac-Man. *Journal of Artificial Intelligence Research*, 30:659–684.

Tan, M. (1993). Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents. In *International Conference on Machine Learning*, 330–337. Morgan Kaufmann.

Thawonmas, R., and Ashida, T. (2010). Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2010)*, 235–240.

Thawonmas, R., and Matsumoto, H. (2009). Automatic Controller of Ms. Pac-Man and Its Performance: Winner of the IEEE CEC 2009 Software Agent Ms. Pac-Man Competition. In *Proceedings of Asia Simulation Conference (JSST 2009)*.

Thompson, T., Milne, F., Andrew, A., and Levine, J. (2009). Improving control through subsumption in the evotanks domain. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2009)*, 363–370.

Thrun, S., and O'Sullivan, J. (1998). Clustering learning tasks and the selective cross-task transfer of knowledge. In Thrun, S., and Pratt, L., editors, *Learning to Learn*, 235–257. Springer US.

Togelius, J. (2004). Evolution of a Subsumption Architecture Neurocontroller. *Journal of Intelligent and Fuzzy Systems*, 15–20.

Tong, B. K.-B., Ma, C. M., and Sung, C. W. (2011). A Monte-Carlo approach for the endgame of Ms. Pac-Man. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, 9–15. IEEE.

Towell, G. G., and Shavlik, J. W. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165.

van Hoorn, N., Togelius, J., and Schmidhuber, J. (2009). Hierarchical Controller Learning in a First-Person Shooter. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2009)*, 294–301.

Verbancsics, P., and Stanley, K. O. (2011). Constraining Connectivity to Encourage Modularity in HyperNEAT. In *Genetic and Evolutionary Computation Conference*, 1483–1490. New York, NY, USA: ACM.

Waibel, M., Keller, L., and Floreano, D. (2009). Genetic Team Composition and Level of Selection in the Evolution of Multi-Agent Systems. *IEEE Transactions on Evolutionary Computation*, 13(3):648–660.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005). Automatic Feature Selection in Neuroevolution. In *Genetic and Evolutionary Computation Conference*.

Wirth, N., and Gallagher, M. (2008). An influence map model for playing Ms. Pac-Man. In Hingston, P., and Barone, L., editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2008)*, 228–233. IEEE.

Yao, X., and Liu, Y. (1997). A New Evolutionary System For Evolving Artificial Neural Networks. *IEEE Transactions on Neural Networks*, 8(3):694–713.

Yong, C. H., and Miikkulainen, R. (2010). Coevolution of Role-Based Cooperation in Multi-Agent Systems. *IEEE Transactions on Autonomous Mental Development*, 1:170–186.

Zitzler, E., Brockhoff, D., and Thiele, L. (2007). The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration. In *Evolutionary Multi-Criterion Optimization*.

Zitzler, E., and Thiele, L. (1999). Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *Evolutionary Computation*.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2002). Performance Assessment of Multiobjective Optimizers: An Analysis and Review. TIK Report 139, TIK, ETH Zurich.

Vita

Jacob B. Schrum was born in Honolulu, Hawaii on December 12th, 1983. He graduated from Kalani High School in 2002, and graduated from Southwestern University in 2006 with a B.S. in Computer Science (with honors), Math, and German (with honors). He enrolled at the University of Texas at Austin in 2006 to pursue a Ph.D. in Computer Science. On the way to this goal, he received his M.S. in Computer Science in 2009.

Permanent Address: 6020 Wiser Ave.
Fort Worth, Texas 76133
schrum2@cs.utexas.edu
http://www.cs.utexas.edu/users/schrum2/

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.