Evolving Multimodal Networks for Multitask Games

Jacob Schrum and Risto Miikkulainen

***Abstract*—Intelligent opponent behavior makes video games interesting to human players. Evolutionary computation can discover such behavior, however, it is challenging to evolve behavior that consists of multiple separate tasks. This paper evaluates three ways of meeting this challenge via neuroevolution: (1) Multinetwork learns separate controllers for each task, which are then combined manually. (2) Multitask evolves separate output units for each task, but shares information within the network's hidden layer. (3) Mode Mutation evolves new output modes, and includes a way to arbitrate between them. Whereas the first two methods require that the task division is known, Mode Mutation does not. Results in Front/Back Ramming and Predator/Prey games show that each of these methods has different strengths. Multinetwork is good in both domains, taking advantage of the clear division between tasks. Multitask performs well in Front/Back Ramming, in which the relative difficulty of the tasks is even, but poorly in Predator/Prey, in which it is lopsided. Interestingly, Mode Mutation adapts to this asymmetry and performs well in Predator/Prey. This result demonstrates how a human-specified task division is not always the best. Altogether the results suggest how human knowledge and learning can be combined most effectively to evolve multimodal behavior.**

***Index Terms*—Multiobjective, Multiagent, Multimodal, Multitask, Neuroevolution, Predator-prey games**

I. Introduction

VIDEO games often feature computer-controlled opponents, or Non-Player Characters (NPCs), which human players must defeat to do well in the game. Creating intelligent behavior for such NPCs traditionally requires extensive hand-coding and troubleshooting by programmers. However, there has been much interest, and some success, in the academic community in evolving NPC behavior for games [1], [2], [3], [4]. These approaches each treat the game as a single task, and optimize the NPC behavior for that task.

However, many entertaining games consist of multiple tasks. Even the classic game of Pac-Man involves two tasks: NPC ghosts chase after Pac-Man and try to catch him, but as soon as he eats a power pellet the task switches, and the ghosts must run from Pac-Man or be eaten.

Recently, the multitask nature of games has been recognized, and a few methods have been developed to deal with them. Two methods that directly motivate the work in this paper are Multitask Learning [5] and Mode Mutation [6].

Multitask Learning is an approach primarily applied in a supervised learning context, originally with neural networks using the backpropogation algorithm [5]: One network has multiple sets of outputs, where each set corresponds to a different, yet related, task. Each set of outputs 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 because knowledge shared across tasks is only learned once and shared, rather than learned independently multiple times. However, this approach has not yet been used to evolve agent behavior. Therefore, multitask evolution for games is one approach evaluated in this paper.

When learning agent behavior, Multitask Learning depends on knowing which task the agent is currently facing, so that the correct network outputs can be used to control the agent. While it may be easy to divide a game into its constituent tasks in some cases, games are typically complex enough that an appropriate division is not always obvious. Not all games have a clear task division like Pac-Man. One approach that addresses this problem is Mode Mutation [6], a mutation operator that adds a new output mode that the network can use when it wants to, so that evolution decides how many modes to have *and* when to use them. Unlike Multitask Learning, Mode Mutation can be used without knowledge of the domain's task division. Interestingly, as shown in this paper, even if such a division is available, Mode Mutation may sometimes learn superior behavior without it. In this paper, the original Mode Mutation is evaluated, as well as an enhanced version that improves it.

A third approach studied in this paper is Multinetwork, in which a separate controller is evolved for each task, and combined in a finished controller that knows which network to use in which task. Multinetwork is loosely based on previous approaches in which networks evolved in separate tasks are combined in a final controller [7], [8], but is simpler than these approaches because the multitask domains in this work are independent (as will be clarified in Section III-A), and therefore there is no uncertainty as to which of the subnetworks needs to be used at any given moment during evaluation. Both the Multinetwork and Multitask Learning approaches rely on an oracle to tell them the current task. Such an oracle will not always be available in practice, but is used here to see what the best attainable performance is.

Multinetwork creates agents with multiple networks, and therefore multiple modes of behavior. Multitask Learning and Mode Mutation create individual networks with multiple modes of behavior, namely multimodal networks. Similar approaches have been explored by researchers in the past. The next section discusses some of these methods and points out the similarities and differences between past work and the methods used in this paper. Following the related work section,

Jacob Schrum and Risto Miikkulainen are with the Department of Computer Science, University of Texas, Austin, TX, 78712 USA (e-mail: {schrum2,risto}@cs.utexas.edu)

two multitask games designed for this paper are discussed. After these games are reviewed, the evolutionary method used to learn agent behavior in these games is described, along with specific details on Multitask Learning and Mode Mutation. The method section is followed by a description of experiments, results, and ideas for future work.

II. Related Work

Domains involving multiple tasks are common in both videogame research and robotics research. Therefore various approaches have been implemented to deal with such domains. This related work is divided into approaches that involve separately evolved controllers, single controllers with modular architectures, and controllers that achieve functional modularity via recurrency.

A. Separately Evolved Controllers

As mentioned above, the Multinetwork approach is similar to previous work in which separate controllers are combined in a single agent. One example is in the TORCS race car game [7], where the goal is to drive around each track as quickly as possible. However, a different behavior is needed to maneuver around opponents than to just speed around the empty track segments. Therefore, Cardamone et al. [7] evolved a racing behavior and a passing behavior in separate evolutionary scenarios, and combined them into an agent that was programmed to use the passing behavior when sufficiently close to an opponent car, and the racing behavior otherwise.

This example shows how multiple networks can be combined in a single agent. Notice, however, that the racing domain is not explicitly divided into racing and passing tasks. The decision to evolve these behaviors separately, and only these two behaviors, was based on human expertise. Furthermore, when to use one behavior instead of the other is based on a mixture of human expertise and trial-and-error testing. The manner in which the multitask games of this paper differ from TORCS will be further defined in Section III-A.

Another separation approach is Togelius's *evolved* subsumption architecture [9], which extends the hand-designed subsumption architecture approach of Brooks [10] by using evolved neural networks to create a hierarchical controller. This approach has been applied to games such as Unreal Tournament [8] and EvoTanks [11]. Like the car racing example above, this approach requires the programmer to divide a domain into constituent tasks and develop effective training scenarios to evolve separate network controllers for each task. This method differs from that of Cardamone et al. in that these controllers are then combined into a hierarchical controller, which is also evolved, thus letting evolution decide how to use the separate controllers.

The subsumption approach is appealing from an engineering standpoint because the controller is hierarchical, and each individual component has a clear purpose. However, the level of human expertise needed to properly divide a domain into subtasks is still restrictive. Therefore, instead of manually combining evolved components, some researchers have also developed methods in which the evolved controllers have built-in capacity to split the task up across separate modules.

B. Modular Architectures

Modular approaches automatically determine which components of the architecture to associate with which task. Mode Mutation falls into this category, since it learns to associate particular control modules with particular situations.

In work by Calabretta et al. [12], neural networks were evolved to control robots using a "duplication operator", which creates a copy of one output neuron with all of its connections and weights. The network then has two output neurons that correspond to the same actuator on the robot, and it needs a means to arbitrate between them. This goal is achieved via "selector units": The output neuron with the highest corresponding selector unit activation is chosen.

The duplication operator differs from Mode 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, but Mode Mutation works at the level of groups of output neurons. While selected, each Mode Mutation "module" is entirely responsible for an agent's behavior (Section IV-C2).

More generally, there is much interest in evolving modular networks, particularly using developmental and generative methods [13], [14], [15]. These approaches evolve modular neural networks based on the assumption that distributing aspects of a problem across particular modules within a network makes optimization of task-specific behavior easier, and therefore faster. The concept of a module is usually less strictly defined in these contexts, but the modules produced by these methods generally need not consist exclusively of output neurons. Rather, a module is a cluster of tightly interconnected neurons with few connections to neurons in other clusters.

Modular networks have also been used in combination with supervised learning. For example, Khare et al. [16] used coevolution to learn the connectivity of individual modules along with their organization within a combining network; the modules themselves were trained through supervised learning. Dam et al.'s Neural-Based Learning Classifier System [17] also combines evolution and supervised learning, but instead of combining network components into a modular network, each network/module 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.

Hierarchical reinforcement learning methods [18], [19] also learn modular architectures, namely hierarchical value functions in which separate components are used in different regions of the state space. In contrast, a method by Sprague and Ballard [20] deals with *non*-hierarchically structured domains by learning separate competing policies using a different reward function for each task. There is also the field of "multitask reinforcement learning" [21], [22] that is concerned with tackling *distributions* of similar but different tasks.

However, modular architectures are not the only way to generate modular behavior. "Functional modularity" can also be obtained with the use of recurrent network connections, as will be described next.

C. Diachronic Behavior

Recurrent connections transmit signals that are not processed by the network until the following time step, which gives them a form of memory. Recurrency is especially useful when brief environmental cues indicate a need for an agent to switch to a behavior that must be carried out beyond the time during which the cues are present. Both Ziemke [23] and Stanley et al. [24] presented results indicating that non-modular architectures with recurrent connections can achieve modular behavior, sometimes superior to the behavior of modular architectures. Such behavior is described as diachronic, because it changes over time in response to the sequence of inputs and the history of internal recurrent activations.

Methods with memory of past states have an advantage in partially observable domains. In such domains, the current observed state cannot be distinguished from other observed states without memory of past states [25]. Recurrent connections help in these situations because they encode and transmit memory of past states; a property that can help a network determine which of several tasks it currently faces.

If recurrency on its own can produce modular behavior, then it should be even more useful when combined with modular architectures. The networks in this paper allow recurrent connections to evolve, so the concept of diachronic behavior will be important in explaining some of the results presented later. These results concern the evolution of agent behavior in two multitask games described in the next section.

III. Multitask Games

This section defines multitask games in order to show what separates the domains of this paper from several other games that involve multiple tasks. Then two such games designed for the experiments of this paper are described.

A. Definition of Multitask Games

In multitask games, NPCs perform two or more separate tasks, each with their own measures of performance. In the extreme case, performance in one task is unrelated to performance in the other tasks, i.e. the tasks are independent. This extreme view makes it easy to analyze task performance independently of other tasks, and is therefore the basis of the domains in this paper. However, multitask games are only interesting if it is desirable to have NPCs capable of performing all tasks. Therefore, all tasks in this paper place the NPCs in the same environment with the same sensors.

There is an important distinction between multitask games as defined in this paper and other games with multiple tasks. In a multitask game, tasks are isolated and it is always clear what the current task is. In contrast, Pac-Man has a clear task division (because eating a power pellet causes a task switch), but the tasks are not isolated (because the positions of agents at the task switch affect performance in the next task). Furthermore, in a game like Unreal Tournament the tasks are not even separated: NPCs choose which task to perform when, such as gathering items and fighting, and may even do multiple tasks simultaneously. Note that an NPC's *internal* state is irrelevant for determining whether the game has separable tasks; the multitask nature of a game is a property of the game, and not of the NPCs programmed to play it. This paper focuses solely on multitask games, though it will be possible to apply the methods in this paper to games with less strict task divisions in the future.

Although the task division is clear in a multitask game, the NPCs may not have access to this knowledge. Therefore, some methods in this paper are designed to control the agents despite not knowing which task they face. Although these methods face a more difficult challenge, they can potentially scale up to games with unknown or ambiguous task divisions.

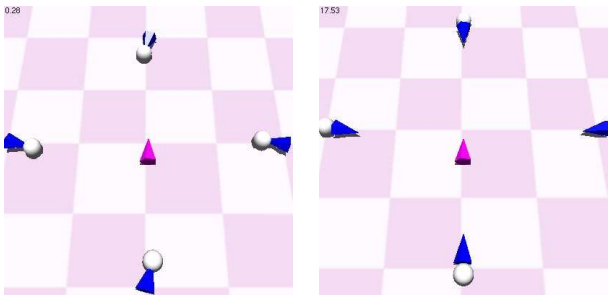
To assure that the test domains in this paper are challenging, they are designed to involve tasks in which good behavior in one task is bad behavior in another task. The separate tasks still have underlying similarities, but different behaviors are needed across tasks to be effective in the game as a whole. The tasks are designed using the simulation environment BREVE [26]. In each task, evaluation begins with a team of NPCs surrounding the player on an infinite plane in continuous space. NPCs start facing the player. Task evaluations have limited duration and are independent from each other. All agents can move forward and backward and can turn left and right with respect to their current heading.

Although the player stands for a human player in principle, a scripted, task-dependent agent was used in the experiments to make a large number of evaluations possible. This "player" agent will be referred to as the "enemy" throughout this paper. The initial heading of the enemy is always random, which requires NPCs to learn situational behavior and makes it detrimental to memorize enemy trajectories. Informal experiments show that NPC behavior evolved against the scripted enemy is still interesting and challenging for humans to overcome.

The multitask games designed for this work are Front/Back Ramming (FBR), which requires NPCs to be alternately aggressive with and protective of different parts of their body depending on the task, and Predator/Prey (PP), which contrasts attacking the enemy with running away from it. Each domain is explained in detail next.

B. Front/Back Ramming Game

This game requires both offensive and defensive behavior in each task, but under different circumstances. Each NPC has a sphere-shaped battering ram affixed to its body, and is therefore called a rammer (Fig. 1). If a ram hits the enemy, then the enemy is damaged, but if the enemy hits any part of the rammer other than its ram, then the rammer takes damage. Rammers do not physically interact with each other (they can occupy the same space). Whenever any agent takes damage, it is temporarily invulnerable for a brief period of time during which it is knocked backwards, which is common in games (e.g. Sonic the Hedgehog, Super Mario Bros.). Such protection gives agents a chance to recover from a mistake. All agents start with 50 hit points, and every hit removes 10 hit points. If the enemy dies, it respawns, and all agents are reset to their starting locations, thus giving the evolving NPCs a chance to accrue additional fitness in whatever evaluation time remains. The resurrection of the enemy models the common occurrence



(a) Front Ramming task (b) Back Ramming task

Fig. 1: Front/Back Ramming game. (a) The start of a Front Ramming task; (b) the start of a Back Ramming task. In both tasks the NPCs start pointed at the enemy in the center. The rams are depicted by white orbs attached to the NPCs. In the Front Ramming task, NPCs can start attacking the enemy 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 (NPCs cannot sense where on their bodies the rams are affixed).

in games of the player having multiple lives. In contrast, when NPCs die, they are dead for the rest of the evaluation.

The game consists of Front Ramming and Back Ramming tasks. When Front Ramming, rams are attached to the fronts of rammers' bodies, and the game starts with rams pointed at the surrounded enemy. When Back Ramming, the rams are attached to the rear ends of the NPCs, and they start facing away from the enemy, so that rammers must execute a 180 degree turn before ramming.

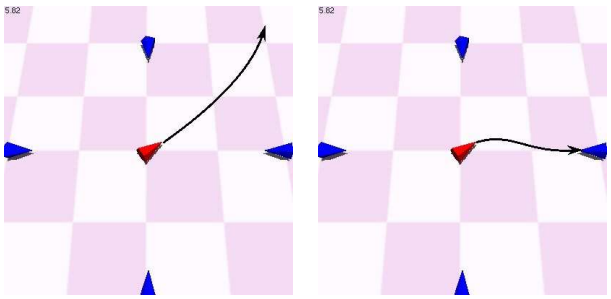
Enemy behavior is essentially the same in both tasks: It will try to circle around the rammers to hit them from the unprotected side if possible, but if threatened by the rams, it will prefer to run and avoid damage.

This game has six objectives. Each task has its own instance of the same three objectives: deal damage to the enemy, avoid damage from the enemy, and stay alive as long as possible. Damage dealt to the enemy is shared by NPCs on a 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 NPCs die, then time alive indicates how long each avoided death, but if no NPCs die, then damage received indicates which team is better.

Even though the NPCs cannot sense how their rams are attached, they need to be alternately offensive and defensive in each task, which makes this game very challenging. The large number of fitness objectives is another cause for difficulty. The second game has fewer objectives, but is nonetheless challenging because NPCs are required to exhibit opposite behaviors in different tasks in order to succeed.

C. Predator/Prey Game

In contrast to FBR, offensive and defensive behaviors are needed in separate tasks within this game. NPCs are either predators or prey depending on the task, and the enemy takes on the opposite role (Fig. 2). The dynamics of the environment and the behavior of the enemy change depending on the task.



(a) Predator task (b) Prey task

Fig. 2: Predator/Prey game. Both the Predator and Prey tasks look the same. (a) The movement path of the enemy in the Predator task: It tries to escape through the nearest gap between two NPCs. (b) The enemy path in the Prey task: It pursues the nearest NPC prey in front of it. Both situations look the same to the NPCs, but because the environmental dynamics and enemy behavior are different, different behavior is needed to succeed.

Predator/prey scenarios have long been of interest in reinforcement learning, multiagent systems, and evolutionary computation [27], [28], [29]. Even the Pac-Man game is a predator/prey scenario. What distinguishes both Pac-Man and the Predator/Prey game of this paper from other predator/prey scenarios is that agents are expected to succeed as both predators *and* prey, rather than just one or the other.

In the Predator task, NPCs are predators and the enemy is prey (Fig. 2a). The enemy tries to escape by moving through a gap between two predators. When a predator hits the enemy, the enemy sustains damage and becomes temporarily invulnerable while being flung away from its attacker, as in FBR. Also as in FBR, NPCs do not physically interact with each other. All agents move at the same speed, which means predators must avoid crowding the enemy, since hitting it can knock it so far away that it is impossible to catch. Therefore, evaluation ends prematurely if the enemy is no longer surrounded. This task is the same as the "Flight" task in [6], but the enemy's escaping behavior is more intelligent because it explicitly seeks gaps through which it can escape.

The Prey task reverses the dynamics of the Predator task, such that the enemy deals damage to NPCs, who are now the prey (Fig. 2b). This task is fairly simple since NPCs can avoid the enemy by just running away. The enemy's behavior consists of moving forward towards the closest NPC. 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 NPCs as in FBR. The Prey task has two objectives: minimize damage received, and maximize time alive. As in FBR, each amount is averaged across team members to get the team score.

As in FBR, the damage dealt per hit is 10 hit points, and all agents have 50 hit points. If the enemy dies (which is only possible in the Predator task), then it respawns surrounded by NPCs, as in FBR. The next section explains the evolutionary methods used to learn NPC behavior in these games.

IV. Evolutionary Methods

Evolutionary multiobjective optimization was used to learn behavior that satisfies the many objectives across tasks. The

evolved individuals were neural networks, and special methods were used to evolve them for multitask games.

A. Evolutionary Multiobjective Optimization

Multitask games are by their very nature multiobjective, since at least one objective is needed in each task. The above domains have multiple objectives *per* task, which makes evolving in them even more challenging. Therefore a principled way of dealing with multiple objectives is needed. Such an approach allows one to avoid the design pitfalls inherent in aggregating objectives (e.g. how to weight objectives), and also has theoretical benefits with respect to the types of solutions that are attainable [30]. In practice, a multiobjective approach can find better overall performance than simply optimizing a single combined objective [2]. The concepts of Pareto dominance and optimality provide the framework for multiobjective optimization[1]:

Pareto Dominance: Vector $\vec{v} = (v_1, \ldots, v_n)$ dominates $\vec{u} = (u_1, \ldots, u_n)$, i.e. $\vec{v} \succ \vec{u}$, iff

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

Pareto Optimality: A set of points $\mathcal{A} \subseteq \mathcal{F}$ is Pareto optimal iff it contains all points in $\mathcal{F}$ such that $\forall \vec{x} \in \mathcal{A}$: $\neg\exists \vec{y} \in \mathcal{F}$ such that $\vec{y} \succ \vec{x}$. The points in $\mathcal{A}$ 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; In this paper this goal is accomplished using the Non-Dominated Sorting Genetic Algorithm II (NSGA-II [31]).

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 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.

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.

Applying NSGA-II to a problem results in an approximation to the true Pareto front. This approximation set potentially contains multiple solutions, which must be analyzed in order to determine which solutions fulfill the needs of the user. However, NSGA-II is indifferent as to how these solutions are represented. For all domains in this paper, NSGA-II was used to evolve artificial neural networks to control the NPCs. The process of evolving these networks is called neuroevolution.

B. Neuroevolution

Neuroevolution is the application of evolution to neural networks. All evolved behavior in this paper was learned via constructive neuroevolution, meaning that 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. Furthermore, these networks are sparsely connected in a style similar to Feature Selective Neuro-Evolution of Augmenting Topologies (FS-NEAT [32]). Initializing the networks in this way allows them to ignore any inputs that are not, or at least not *yet*, useful. It is beneficial to ignore certain inputs early in evolution, when establishing a baseline policy is more important than refining the policy.

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. As mentioned in Section II-C, recurrent connections are particularly useful in partially observable domains, like those of this paper.

The mutation operators are similar to those in NEAT [33]. However, since NEAT was designed to use only a single fitness function, it turned out simpler to reimplement these features of NEAT in NSGA-II than to modify NEAT to use a Pareto-based multiobjective approach. One feature of NEAT that was *not* used in this paper is crossover, because preliminary experiments showed that it often had no effect, and in some cases even decreased performance in the domains of this paper.

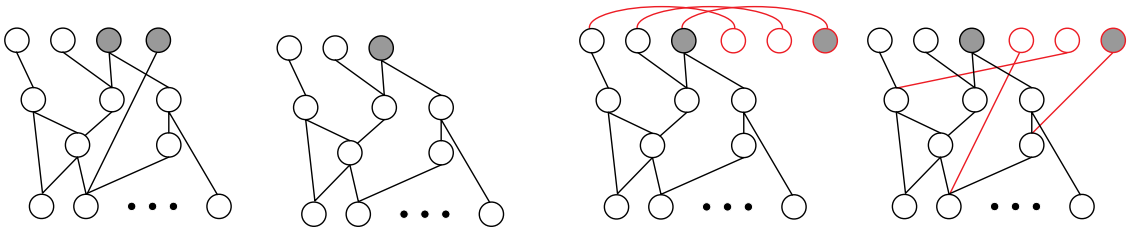
The form of neuroevolution described so far has been used to solve many challenging problems [33], [34], [35], but this approach does not directly target multitask domains. However, this approach can be used to implement the Multinetwork approach, where separate component controllers are evolved for each task of a multitask game.

In contrast, the next section describes two enhancements to neuroevolution for dealing with multitask domains using a single neural network.

C. Multitask Evolution

Two methods for evolving multimodal networks are described: Multitask Learning, which translates work by Caruana [5] into a neuroevolution framework, and Mode Mutation,

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



(a) Multitask network (b) One-mode network (c) Network after MM(P) (d) Network after MM(R)

Fig. 3: Networks for playing multitask games. (a) Multitask network with two modes, each consisting of two outputs. During execution, the simulator knows which of the two tasks the network is performing, and picks the appropriate outputs from the multitask network accordingly. (b) Network with only one output mode containing a grey preference neuron. (c) How the one-mode network would be modified by MM(P) to create a network whose new output mode receives inputs from the previous mode. The new lateral connections all have weights of 1.0 to assure similarity to the previous mode as a starting point for further evolution. (d) How the one-mode network would be modified by MM(R). In this case, the new mode is connected by randomly weighted synapses to random nodes in the hidden and input layers, thus making the new mode likely to be very different from pre-existing modes. For both types of Mode Mutation, further mutations can change the behavior of these new modes, and add modes beyond the two shown. These multimodal approaches are compared against two unshown approaches: (1) One-mode networks like the one in (b), but lacking the preference neuron (since they will only ever have one mode), and (2) Multinetwork, which involves training one-mode networks in separate tasks and combining them. The benefit of multimodal approaches over one-mode networks is that they allow networks to have multiple policies for different tasks/situations, which is useful in complex games.

which was introduced in our previous work [6], but is enhanced in this paper.

1) Multitask Learning: Multitask Learning assumes that evolving agents are always aware of the task they currently face. Each network is equipped with a complete set of output neurons per task (Fig. 3a). Therefore, if two outputs are required to define the behavior of an NPC, and the NPC must solve two tasks, then the networks would have two outputs for each task, for a total of four outputs. When performing a given task, the NPC bases its behavior on the outputs corresponding to the current task, and ignores the other outputs.

2) Mode Mutation: Mode Mutation does not provide NPCs with knowledge of the current task. It is a mutation operator that adds a new output mode to a network. As a result, networks can have many different output modes, often even exceeding the number of tasks in the domain.

There is no mode-to-task mapping, therefore a way of choosing a mode to define NPC behavior for each time step is needed. Mode arbitration depends on output neurons called preference neurons (similar to selector units [12]). Each mode has one preference neuron in addition to several policy neurons, i.e. neurons that define the agent's behavior. Every time step, the output mode whose preference neuron value is highest is selected. So if two neurons are needed to define agent behavior, Mode Mutation adds three neurons to the output layer: two policy neurons and one preference neuron.

Two methods of Mode Mutation are evaluated in this paper. The first is Mode Mutation Previous (MM(P); Fig. 3c), i.e. the original version [6]. Neurons for new modes start with one input synapse each. Each input comes from the corresponding neuron of the previous output mode. 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 mode is not identical to the previous mode because the $\tanh$ activation function is used on each neuron (which is common in neural networks), and acts as a squashing function. Therefore, the new mode is a similar but slightly diminished (in terms of activation) version of the previous mode. Future mutations can further differentiate the new mode from its source mode such that both modes exhibit distinct behavior.

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

The second method of Mode Mutation, new in this paper, addresses this problem. With Mode Mutation Random (MM(R); Fig. 3d), each neuron in a new mode receives one input with a random weight from a random source in either the hidden or input layer. This approach is risky since a new mode could cause fitness scores to plummet, but it has the advantage of more quickly introducing *distinct* modes of behavior.

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

However, modes can be safely deleted in MM(R) networks without modes becoming disconnected. In fact, preliminary experiments indicated that the ability to delete modes is very important to the success of MM(R). Therefore, whenever using MM(R), a mode-deletion mutation is also used.

Throughout evaluation, the number of times each mode is used is tracked. If a mode-deletion mutation occurs on a network with multiple modes, then this data is used to choose for deletion the output mode that was used the least in the previous evaluation. If multiple modes are tied for least usage (usually meaning they were not used at all), then the oldest of

these modes is deleted. This procedure removes unimportant, dead-end modes and allows the other mutation operators to focus on refining the remaining useful modes.

V. Experiments

This section describes how the methods discussed are evaluated. Following the experimental setup, methods used for evaluating the results are discussed.

A. Experimental Setup

The approaches in Section IV to solving multitask games were applied to the games in Section III. Experiments in both games were run in a similar manner. All experiments used constructive neuroevolution with a weight-mutation rate of 0.4, link-mutation rate of 0.2, and node-mutation rate of 0.1. These and other parameters are similar to those used in previous work [2], [6], [35].

In the results below, `Control` represents networks with one mode used in both tasks of each game. `Multitask` represents networks with one mode for each task in a game. These networks always knew which task they were facing, and used the appropriate mode accordingly. Both Mode Mutation methods, `MM(P)` and `MM(R)`, had initial populations containing networks with only one mode. New modes could be added by the appropriate Mode Mutation, whose rate was 0.1 in each case. Additionally, `MM(R)` used a mutation to delete the least-used output mode at a rate of 0.1. The fifth method, `Multinetwork`, involves evolving networks for each task individually, and then combining the resulting controllers so that the appropriate network is used in the task for which it was evolved. Each "run" of `Multinetwork` actually consists of a pair of runs: one in each task of a multitask game. 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 multitask game, representing the result of a single `Multinetwork` run.

NPCs were evolved 20 times for 500 generations for each method in each game. NSGA-II was used with a population size of $\mu = \lambda = 52$ to evolve neural network controllers (52 is a historical value; other values work as well). Each controller earned scores by being evaluated in multitask games. For each task, a network was copied into each of the four members of a team of NPCs. Such homogeneous teams tend to be better at teamwork because the altruistic behavior of individuals is not punished if it contributes to greater team scores [36]. We have confirmed this advantage in our own work [2], [6].

Because the enemy faces a random direction when it is spawned (Section III-A), evaluations are noisy. A common approach used to deal with noisy evaluations is to average fitness scores across multiple evaluations [8], [37], [7]. Therefore, every network was evaluated three times in each task, and the final scores in each objective were the averages. The maximum evaluation time for each task was 600 time steps, which was chosen through trial and error to balance the need for enough time to exhibit interesting behavior against the desire to reduce the overall duration of each experiment.

On each time step of the simulation, the enemy acts according to scripted behavior (described in Section III), and the evolving agents act according to their neural networks. On each time step, the NPCs' sensors provide inputs to the network, which are then processed to produce outputs, which then define the behavior of the NPC for the given time step.

The inputs to the NPCs' neural networks are described in Table I. 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 NPC'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 NPC's history of senses and actions. The recurrent states of all NPCs are reset whenever the enemy respawns. Even though there are many network inputs, recall that a feature-selective approach [32] is used to evolve the networks. This approach 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 mode 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.

B. Assessing Multiobjective Performance

A run of NSGA-II creates an approximation to the true Pareto front, i.e. an approximation set. Multiobjective performance metrics compare approximation sets from different runs. 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 metric being calculated, as is further explained below. The role of normalization in interpreting the results of each metric is also explained below. The normalized objective scores are used to calculate two types of metrics: hypervolume [38] and unary epsilon indicator values [39].

1) Hypervolume: Hypervolume (HV) is the primary performance measure of this paper. 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

TABLE I
DESCRIPTION OF INPUT SENSORS FOR NPCS.

Name	#	Range	Description
Bias	1	$\{1\}$	Constant
NPC/Enemy Heading Diff.	1	$(-\pi, \pi]$	Shortest amount the NPC would have to turn to have the same heading as the enemy.
NPC Heading/Enemy Loc. Diff.	1	$(-\pi, \pi]$	Shortest amount the NPC would have to turn to be directly facing the enemy.
NPC Dealt Damage	1	$\{0, 1\}$	1 if NPC dealt damage to enemy on previous time step, 0 otherwise.
NPC Received Damage	1	$\{0, 1\}$	1 if NPC received damage from enemy on previous time step, 0 otherwise.
Any NPC Dealt Damage	1	$\{0, 1\}$	1 if *any* NPC dealt damage to enemy on previous time step, 0 otherwise.
Any NPC Received Damage	1	$\{0, 1\}$	1 if *any* NPC received damage from enemy on previous time step, 0 otherwise.
Enemy Knockback	1	$\{0, 1\}$	1 if enemy is temporarily invulnerable because it is being knocked back, 0 otherwise.
In Front of Enemy	1	$\{0, 1\}$	1 if magnitude of shortest turn the enemy would need to make to face the NPC is less than or equal to $\pi/2$, 0 otherwise.
NPC/Teammate Heading Diff.	4	$(-\pi, \pi]$	For each slot x within the team of NPCs, sensor returns the shortest amount the sensing NPC would have to turn to have the same heading as the teammate in slot x. The difference in heading from an NPC to itself is always 0.
NPC Heading/Teammate Loc. Diff.	4	$(-\pi, \pi]$	For each slot x within the team of NPCs, sensor returns the shortest amount the sensing NPC would have to turn to be directly facing teammate x. 0 is returned by the sensor corresponding to the sensing NPC.
Teammate Dealt Damage	4	$\{0, 1\}$	For each slot x within the team of NPCs, sensor returns 1 if teammate x dealt damage to enemy on previous time step, 0 otherwise.
Enemy Ray Traces	5	$\{0, 1\}$	Each sensor returns a 1 if it is currently intersecting space occupied by the enemy, and 0 otherwise.
Teammate Ray Traces	5	$\{0, 1\}$	Each sensor returns a 1 if it is currently intersecting space occupied by any teammate, and 0 otherwise.

Each row stands for a different sensor or group of sensors, with number indicated by the "#" column. Some groups of sensors refer to team member slots. These groups consist of four sensors each, where each sensor corresponds to a given NPC, determined by its *starting* position with respect to the enemy (north, south, east or west). Given NPC x, its sensors that correspond to team slot x will refer to itself. Those same sensors in a different NPC y will refer to NPC x as well. Note that the values for the "NPC/Teammate Heading Diff." and "NPC Heading/Teammate Loc. Diff." sensors will be different for NPCs x and y, because the values depend on relative NPC positions and headings. Another type of grouped sensors are ray traces. Each of these sensor groups consists of an array of 5 sensors that are 3.5 times the length of an agent, and positioned around the NPC relative to its heading at the angles of $-\pi/4$, $-\pi/8$, 0, $\pi/8$ and $\pi/4$ radians. The 'Range' column lists the set/interval of possible sensor values for each sensor type. Whenever turning is referred to, a negative value corresponds to a left turn and a positive value corresponds to a right turn. This set of 31 inputs is sufficient for the evolving NPCs to develop complex and interesting behavior in the domains of this paper.

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.

When analyzing how hypervolume changes across generations, the reference points used for each game were their corresponding zero vectors, containing the minimum scores for each objective: (0,−50,0) for PP and (0,0,−50,−50,0,0) for FBR, 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 (250,0,600) for PP and (310,210,0,0,600,600) for FBR. The maximum point for PP indicates that the best damage score in the Predator task was 250, and some NPC teams survived the full 600 time steps of the Prey task sustaining no damage. The maximum point for FBR indicates that the best damage scores in Front and Back Ramming were 310 and 210, respectively. NPC teams in each of these tasks also managed to survive the entire 600 time steps sustaining no damage. The normalization scheme for hypervolume ranges from the absolute minimum possible scores to the maximum achieved because hypervolume scores are presented from generation 0, when scores are very small, all the way to generation 500, where the maximums occur.

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 [38], 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 [40]. 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 unary epsilon indicator.

2) Epsilon Indicators: Like hypervolume, both epsilon indicators are Pareto-compliant [39]. 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. [39], the reference set R for each game was the super Pareto front (Pareto front of several Pareto fronts) of all fronts for which epsilon values were being calculated across all methods.

Due to the extra complication of the reference set, epsilon indicator values are *only* calculated for the final generation of each run. However, since final performance is really all that matters, it is appropriate to focus on the final generation, especially since hypervolume values are calculated at every generation, 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, (0,−50,315.333333) is the minimum point for PP, and the lowest time alive score is now 315.333333. For FBR, the minimum point is (10,10,−50,−50,503.25,264.5), 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 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 Pareto fronts discovered for each multitask game cover the space of all objectives. For each of these metrics, a better score does not guarantee a superior Pareto front, but superior scores in *all* metrics gives confidence that a given Pareto front actually is better.

However, the individual tasks of each multitask game are only concerned with certain dimensions within objective space. It therefore makes sense to also compare performance within the individual tasks, as described in the next section.

C. Multitask Performance Metrics

Extra care must be taken to characterize performance properly in multitask games. In such games, NPCs 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. However, because tasks are independent in multitask games, there are no inherent trade-offs between objectives from separate tasks. When such trade-offs are observed in 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 Pareto fronts for each individual task, and compare these results to those for the full game. If one method is superior to another in the full game, but equal in the component tasks, then the superiority in the full game is exclusively a result of individuals that 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 paper is concerned with intelligent NPC behavior, it is at least possible to say that an NPC is only successful if it surpasses certain minimum expectations, i.e. obtains adequate goal scores in *each* objective. In other words, extreme trade-offs are considered undesirable. 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 V-B1). 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".

Note that this performance metric cannot be properly calculated for the `Multinetwork` approach because each `Multinetwork` run consists of two runs with different

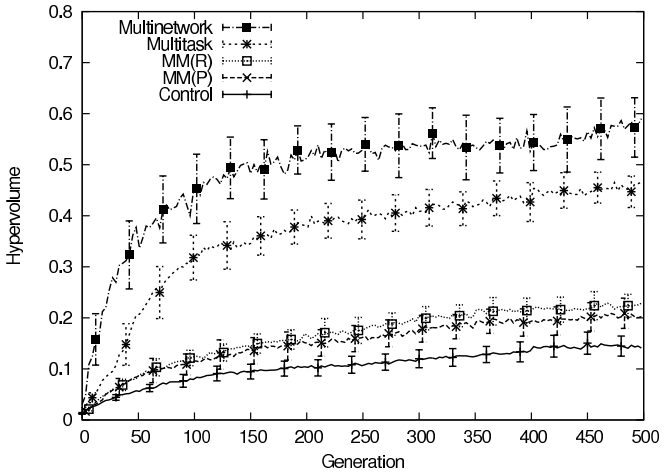


Fig. 4: Average hypervolumes in the Front/Back Ramming game. For each method, average normalized hypervolumes across 20 runs are shown by generation with 95% confidence intervals. The figure indicates that multimodal approaches are superior to the unimodal approach, represented by `Control`. Both `MM(P)` 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 Mode 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.

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 any success plots.

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

VI. Results

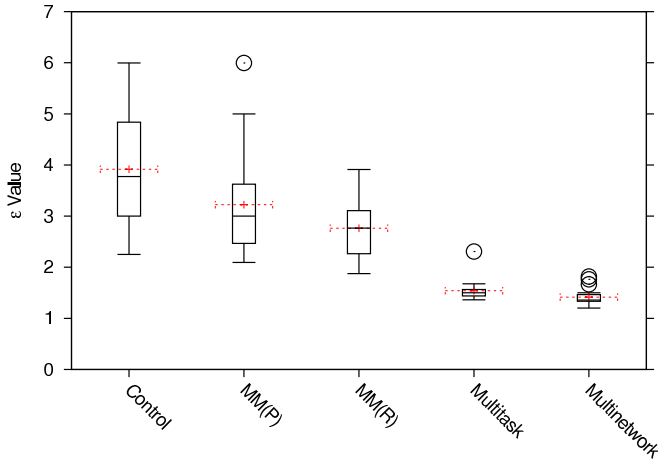
The results for FBR and PP are presented in this section in terms of the metrics described above. FBR is described first because its results are more straight-forward, followed by PP, for which the results were more surprising.

A. Front/Back Ramming Results

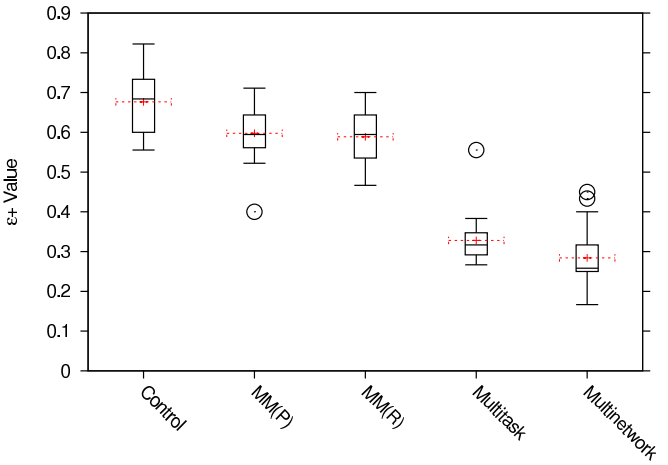
The results for FBR conform to expectations of how the different methods should perform: `Control` performed the worst, both `MM(P)` and `MM(R)` are better, `Multitask` is better still, with `Multinetwork` emerging as the best. The hypervolume learning curves (Fig. 4) indicate that this ordering is established early and maintained throughout evolution.

When considering the final generation only, epsilon indicator values corroborate these results (Fig. 5). Furthermore, Mann-Whitney U tests confirm that there are significant differences in the final generation between adjacent methods in order of increasing performance: `Control`, `MM(P)`/`MM(R)`, `Multitask`, then `Multinetwork`. The two Mode Mutation methods are lumped together because there is no significant difference between them (Table II).

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` (Fig. 6, Table III). 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.



(a) I^1_ϵ Values.



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

Fig. 5: 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^1_ϵ values and (b) $I^1_{\epsilon+}$ 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 $1.5IQR$ (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 Mode 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 Fig. 4

This point is seen in the average success counts as well (Fig. 7), 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

TABLE II
TWO-TAILED MANN-WHITNEY U TEST VALUES FOR THE FINAL GENERATION OF FRONT/BACK RAMMING.

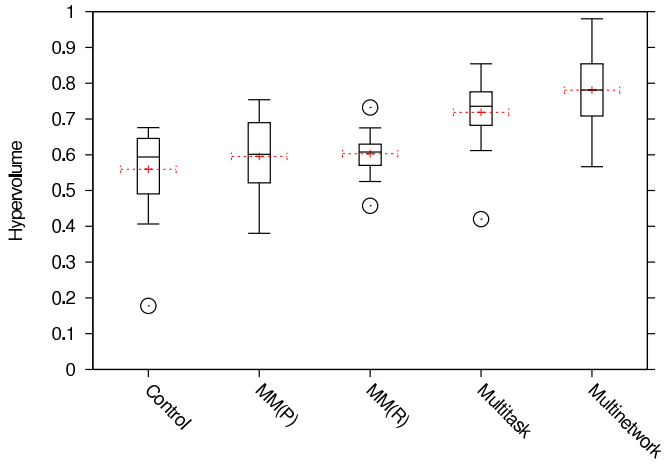
Comparison	HV	I^1_ϵ	$I^1_{\epsilon+}$
`Control` vs. `MM(P)`	**88**	**87.5**	127.5
`Control` vs. `MM(R)`	**36**	**81.5**	**76**
`MM(P)` vs. `MM(R)`	148	183.5	150.5
`MM(P)` vs. `Multitask`	**2**	**4.5**	**3**
`MM(R)` vs. `Multitask`	**4**	**7**	**6**
`Multitask` vs. `Multinetwork`	**59**	*111*	**93**

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 $I^1_{\epsilon+}$ comparison between `Control` and `MM(P)` are significantly different, and all but the I^1_ϵ comparison between `Multitask` and `Multinetwork` are different at the $p < 0.01$ level.

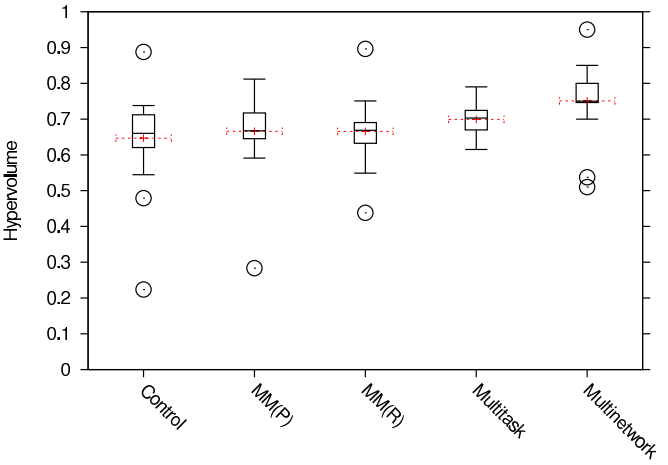
TABLE III
TWO-TAILED MANN-WHITNEY U TEST VALUES COMPARING HYPERVOLUMES FOR THE ISOLATED FRONT AND BACK RAMMING TASKS, I.E. IGNORING OBJECTIVES FROM THE OTHER TASK.

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**

`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.



(a) Hypervolumes for Front Ramming.



(b) Hypervolumes for Back Ramming.

Fig. 6: 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 Mode 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.

scores achieved by agents in the game. 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 Fig. 8, 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 V-C, 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 both tasks. The scores of these individuals are shown in Table IV.

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 NPC/Enemy Heading Diff. and the NPC Heading/Enemy Loc. Diff. sensors were commonly included. NPC teams also tended to have sensors for the NPC Heading/Teammate Loc. Diff. and Teammate Dealt Damage of at least one teammate, though the exact teammates varied across runs. Use of Enemy 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 NPCs 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

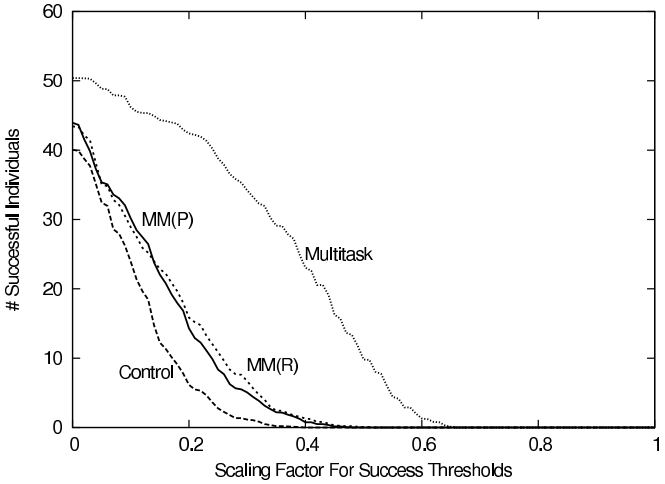


Fig. 7: Average success counts in Front/Back Ramming. This plot shows the average number of individuals across the final size 52 populations of each single-network method that are considered successful, in that their objective scores pass a threshold for all objectives, indicating the ability to perform 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 Mode Mutation curves intersect each other, emphasizing that they are not very different in FBR.

TABLE IV
THE MOST SUCCESSFUL INDIVIDUAL OF EACH METHOD IN FRONT/BACK RAMMING.

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

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 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.

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 Fig. 9. The specific scores achieved by the team in this figure are $(220, 120, -17.5, 0, 600, 600)$.

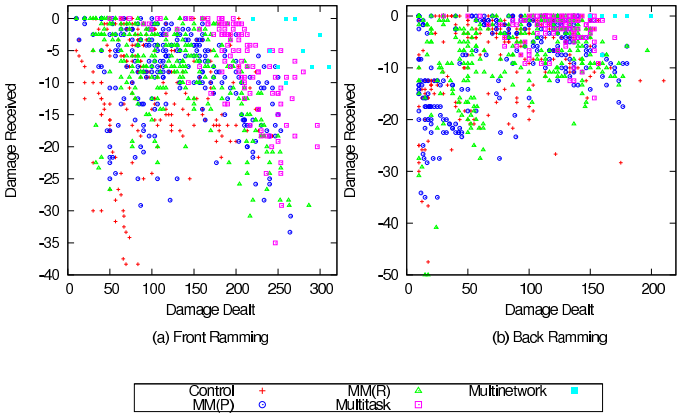
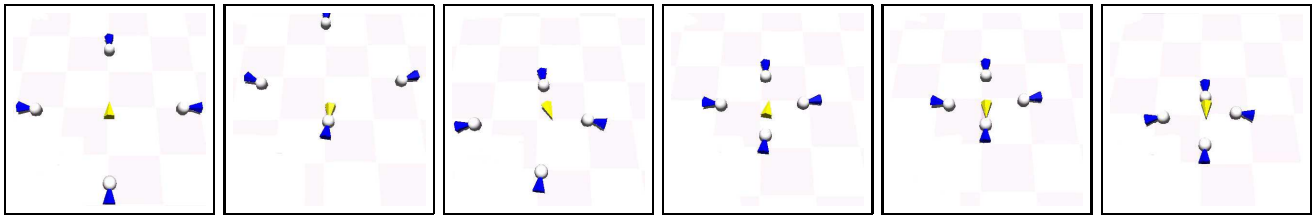


Fig. 8: Slices of Super-Pareto fronts from Front/Back Ramming. Damage received vs. damage dealt in Front Ramming, and damage received vs. damage dealt in Back Ramming are shown for all members of each Super-Pareto front across 20 runs of each method. These plots illustrate the non-normalized domain performance 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.

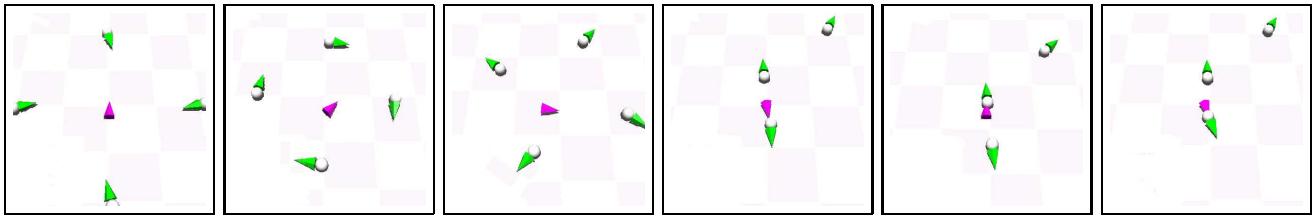
Such behaviors are easy for `Multitask` to learn since the networks have completely different policies for each task. In the Front Ramming task NPCs rush forward to ram the enemy, and in Back Ramming the same NPCs immediately turn around at the start of the trial so that they can attack the enemy with 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.

Mode 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 Mode 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 Fig. 10. The network had nine modes. 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 mode was used by the NPC team during evaluation in a particular task. Therefore, three of the nine modes were not used at all, two were not used in Front Ramming and only used sparingly in Back Ramming, and another mode was only used sparingly in both tasks. The

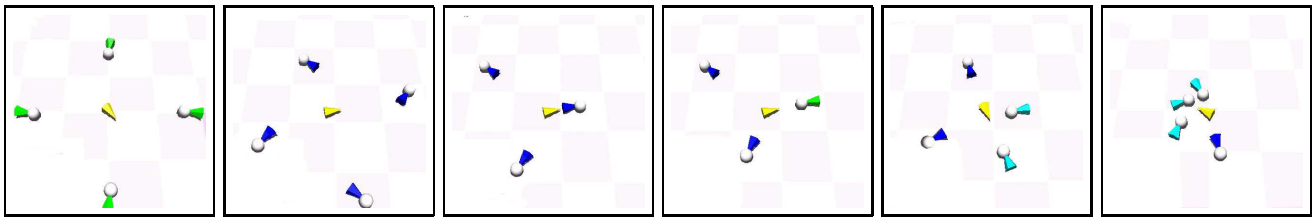


(a) Front Ramming

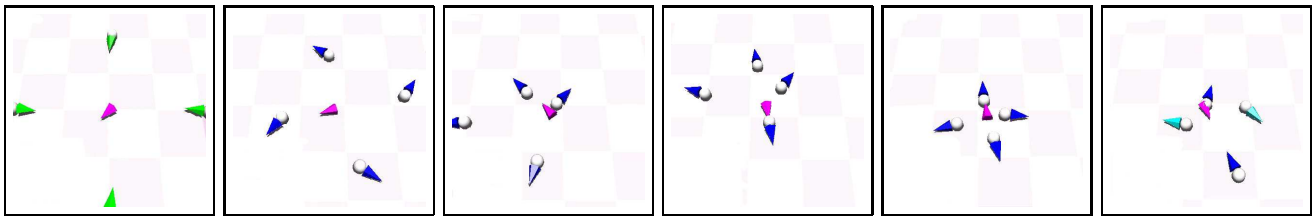


(b) Back Ramming

Fig. 9: 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 NPC behavior is distinctive in each task, since different output modes are dedicated to each one. `Multitask` networks immediately take advantage of their knowledge of the current task: In Front Ramming they attack immediately (column 2), and in Back Ramming they turn immediately around (column 2) and then start attacking. No time is wasted figuring out what task is being faced, as Mode Mutation networks must do (Fig. 10).



(a) Front Ramming



(b) Back Ramming

Fig. 10: Illustration of intelligent behavior learned by Mode 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 NPCs perform the same maneuver, since they do not know yet which task they are performing. They start by turning their backs towards the enemy. In the Back Ramming task, this strategy is immediately effective, as illustrated by the NPCs confining the enemy while knocking it around with their rams (columns 3–6). In the Front Ramming task, this behavior causes the NPCs to be hit (column 3), but this hit does two things: (1) the attacking NPC is flung backwards with its front ram facing the enemy (column 4), and (2) NPCs sense being hit, and as a result switch network modes so they now attack with their front rams (columns 5–6). Preference for the new attack mode is then maintained by internal recurrent state. This multimodal behavior is a good example of how Mode Mutation networks can learn to overcome the challenges of a multitask game with a combination of recurrent connections and structural modularity.

remaining three modes were primarily in charge of controlling the team. In particular, NPCs made use of mode 1 more often in Back Ramming than in Front Ramming, and less use of mode 3 in Back Ramming than in Front Ramming. Mode 2 is used equally often in both tasks.

For a single NPC in the team generated from this network, Fig. 11 shows how the activation of each preference neuron fluctuates during a single Front Ramming evaluation. Some modes mostly maintain constant activation while others exhibit wild thrashing behavior, in essence merging two modes. Others behave like digital signal waves, and some gradually rise and fall like sin waves. The modes that are most often chosen by this network tend to control the NPC for prolonged periods, making it easy for a human observer to associate particular modes with particular behaviors. Other networks (not shown) sometimes utilize the other types of modes instead. `MM(R)` thus uses its modes 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 modes, because they could not delete modes. For example, an `MM(P)` network with scores of (117.78, 152.22,

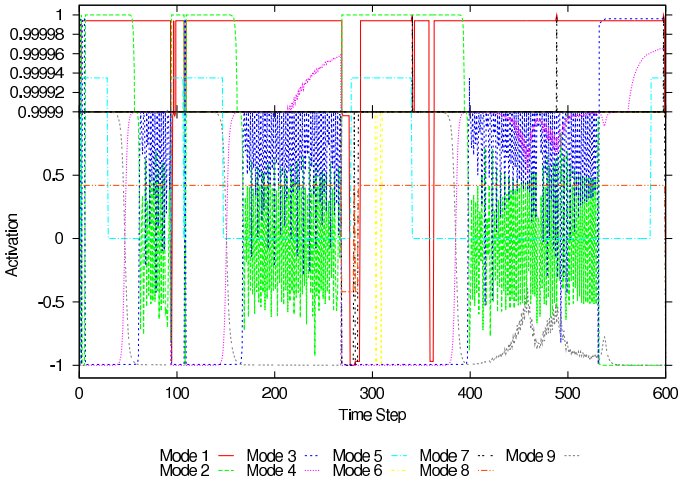


Fig. 11: Preference neuron activations of MM(R) network in Front Ramming. These activations correspond to a single member of the team whose behavior is illustrated in Fig. 10. At any given time step, the mode whose preference neuron has the highest activation is chosen. Neuron activations are restricted to the range $[-1, 1]$ from using the tanh activation function, but in order to better tell which mode has the highest activation on each time step, the range from 0.9999 and up is magnified in the figure. Mode 1 was used most because it almost always maintains high activation. This mode was used when the NPC was attacking the enemy, after it realized which task it faced. Mode 2 has activations even higher than mode 1 for three prolonged periods. Each of these periods begins immediately after the enemy spawns. This mode is second most used because whenever the enemy respawns, the recurrent states of NPCs are flushed, so that the NPCs lose the knowledge they had previously gained from interacting with the enemy; mode 2 gets used after each enemy respawn until the NPC is reminded that it is in the Front Ramming task. Mode 3 is used third most because it controls the NPC for a prolonged period near the end of the evaluation. This unusual usage occurs because the NPC team's plan goes awry after the last respawn, which is why the NPCs come up short of killing the enemy a third time. Mode 3 gets used because the NPC is trying to escape taking damage; a situation it does not have to deal with during the first two successful enemy spawns. The other modes were either not used at all, or used only for single isolated time steps at points when their activation spiked. For example, see how mode 7 spikes around generations 340, 490 and 600. This figure shows how MM(R) can evolve a network that makes use of multiple, separate modes of behavior in a multitask game.

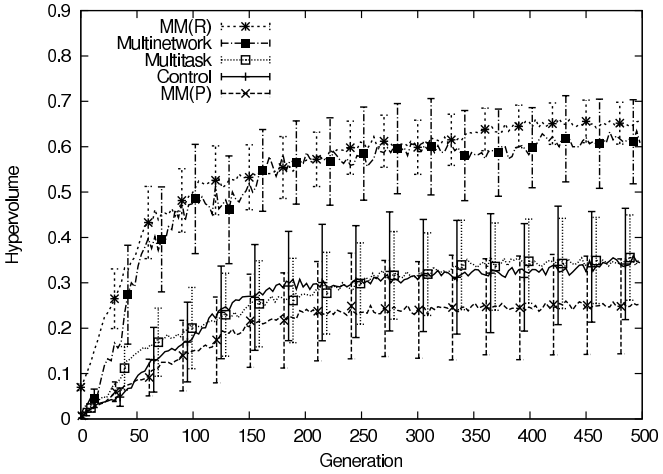


Fig. 12: Average hypervolumes in the Predator/Prey game. For each method, average hypervolumes across 20 runs are shown by generation with 95% confidence intervals. In contrast to the FBR game, MM(R) outperforms Control, MM(P), and Multitask significantly. None of these three methods with lower performance 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 modes automatically.

-7.22, -8.61, 600, 600) had 22 modes, but only five were used. The usage profile of these five modes 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 modes, only five were used, and in Front Ramming only three of those modes were used. Even in Back Ramming, these two extra modes are used sparingly. In each task, NPC teams are controlled a majority of the time by the same two modes. However, these two modes combined with minor contributions from the remaining three to define an effective multimodal strategy for the NPCs.

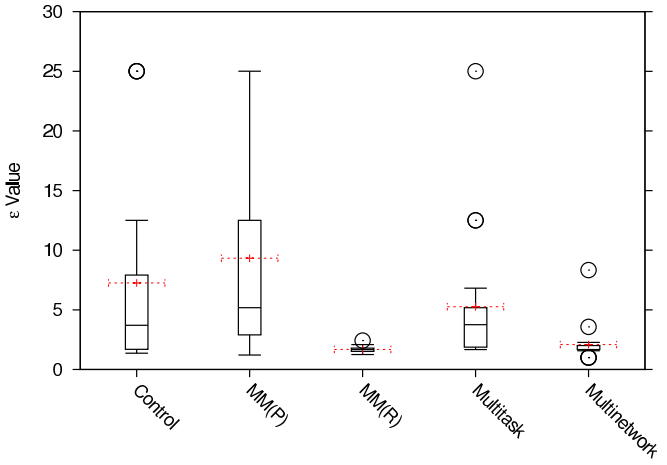
Though in terms of performance there is no significant difference between MM(P) and MM(R), each method's pattern of correspondence between chosen modes and exhibited behaviors indicates that MM(R) networks tend to associate particular modes more clearly with particular behaviors. That is, MM(R) behaviors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

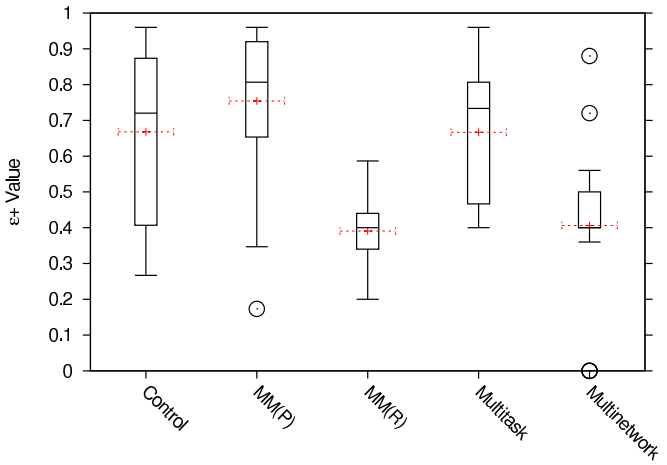
B. Predator/Prey Results

The results in PP are unexpected, in that neither Multitask nor MM(P) performs better than Control, but MM(R) and Multinetwork greatly outperform all of these methods, and achieve roughly equal performance. The hypervolume learning curves (Fig. 12) show MM(R) and Multinetwork quickly improving and remaining better than all the other methods. The epsilon indicator values in the final generation confirm these results (Fig. 13), and the Mann-Whitney U tests (Table V) confirm that the relevant differences are significant.

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 NPCs simply need to run away from the enemy 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.



(a) I^1_ϵ Values.



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

Fig. 13: 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^1_ϵ values, and (b) $I^1_{\epsilon+}$ values are calculated, and presented in the same manner as in Fig. 5. `MM(R)` and `Multinetwork` are superior to all other methods. `MM(R)`'s performance is unexpected since `Multitask`, like `Multinetwork`, always has knowledge of the current task, and has a specific policy for each task. There is little difference between `Control`, `Multitask`, and `MM(P)`. In terms of I^1_ϵ values, `Control` and `Multitask` seem better than `MM(P)`, but Table V indicates that these differences are not significant. The slight difference between `MM(R)` and `Multinetwork` is not significant either.

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 (Fig. 14): The Mann-Whitney U test values for comparing these scores are identical to those for I^1_ϵ and $I^1_{\epsilon+}$ in the full PP game (Table V). 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 (Fig. 15). However, pressure to do well in both the Predator

TABLE V
TWO-TAILED MANN-WHITNEY U TEST VALUES FOR THE FINAL GENERATION OF PREDATOR/PREY.

Comparison	HV	I^1_ϵ	$I^1_{\epsilon+}$
`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

`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.

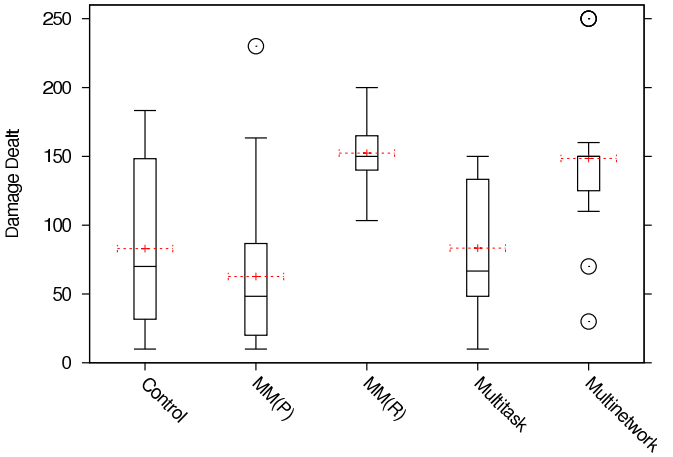


Fig. 14: Best damage-dealt scores in the Predator task of Predator/Prey. Isolating the Pareto fronts for the Predator task reduces it to a single-objective task. Because most individuals in the Pareto fronts for the full task had perfect damage-received and time-alive scores, these damage-dealt scores 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 clearly 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.

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 performance 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

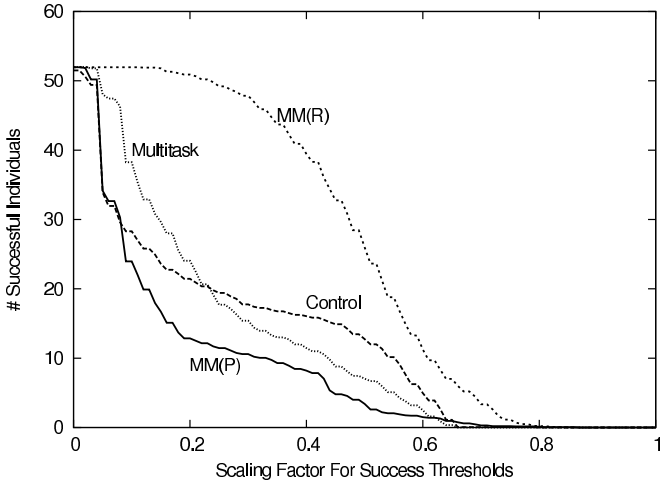


Fig. 15: Average success counts in Predator/Prey. The single-network method that has the most high-performing individuals for the highest success thresholds is `MM(R)`, whose curve dominates all others. At low success thresholds, the next best method is `Multitask`, but at higher success thresholds it is overcome by `Control` because `Multitask` networks have relatively low damage scores in the Predator task.

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 an NPC run from the enemy. 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 NPC/Enemy Heading Diff. and NPC Heading/Enemy 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 NPCs (animations can be seen 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 NPCs 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 NPCs 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 (Fig. 14).

A possible explanation is that giving equal attention to each task, as the `Multitask` architecture requires, is unnecessary and even detrimental in this game, 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 Mode Mutation evolution is free to choose how many modes 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 mode that is extensively used in both the Predator and Prey tasks. For example, an `MM(R)` network scoring $(146.67, 0, 600)$ had 19 modes, 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 mode is thus primarily responsible for controlling the team in both tasks. Furthermore, the fourth mode, which is second 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 mode. However, the behavior is still *functionally* modular.

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. Success with `MM(P)` seems to depend on luck in this domain. When `MM(P)` succeeds, it also tends to use few of its modes. For example, an `MM(P)` network scoring $(155.67, 0, 600)$ had 10 modes, but only used three of them. In fact, only *one* mode was used in the Prey task. This same mode was the most-used mode in the Predator task, where the usage profile of these three modes was 88.73%, 6.99%, 4.28%. It seems that because `MM(P)`'s output modes are so interconnected and similar, it is difficult for networks to specialize modes for either task, so success for `MM(P)` mainly happens when multiple modes are ignored.

Since the few quality `MM(P)` networks and the many quality `MM(R)` networks tend to favor only one mode, perhaps one mode is the ideal number for this game. Then why does `MM(R)` do so well? The mode-deletion mutation is likely the key. If a single quality mode is all that is necessary, then `MM(R)` can, in addition to creating new, novel modes via mode mutation, delete pointless, unused modes via mode deletion. In other words, `MM(R)` helps evolution find the right one mode for this game. In fact, modes found early on can serve as crutches until better modes are found, at which point the old modes can be deleted. Switching behavior in this way is easier than incrementally changing the behavior of existing modes, as in the `Control` and `Multitask` methods.

VII. Discussion and Future Work

Interestingly, although Multitask Learning and Mode Mutation each work well in at least one of the multitask games of this paper, the Multinetwork approach worked well in both. However, these domains are stepping stones towards

more difficult domains where tasks are not independent, and Multinetwork may not be as easy to apply. In order to best exploit these methods in more complex games, 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 the ideal choice for multitask games 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 counterexample to the Multitask Learning hypothesis; 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 `Multinetwork`. The reason could simply be that the Multitask Learning hypothesis does not apply in the context of evolutionary computation, or at least not in the narrow context of multiobjective constructive neuroevolution. Further studies using different domains and evolutionary approaches are needed to test this hypothesis.

The limited success of Multitask Learning only applied in FBR, where the task division *properly* split the challenges of the game. As was shown by the PP game, when tasks were not equally difficult, `Multitask`'s separate dedicated modes 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 largely by using only one output mode. 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, it may also work well in games where the task division is dynamic or overlapping. For instance, `MM(R)` should work well even when the agents choose which task they perform, as in the Unreal Tournament example discussed in Section III.

Because the Mode Mutation methods never knew which task they were facing, their multimodal/modular behavior was likely established by recurrent connections. For instance, the behavior demonstrated by `MM(R)` in Fig. 10 involves a mode switch after being hit by the enemy, but the hit itself is sensed for only a single time step. Therefore, this knowledge must have been maintained by recurrent connections, which means the behavior is diachronic. Since memory of past states determines how multiple modes are used, this behavior is an example of how diachronically influenced modularity (i.e. functional modularity based on recurrency) can work in concert with structural modularity.

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, rather than designating modes/networks as in `Multitask`/`Multinetwork`, networks could be informed of which task they face via an input sensor. This option could be used in combination with Mode Mutation, thus giving it awareness of the task division in addition to access to multiple modes of behavior.

`MM(R)` specifically could be further improved by controlling bloat more intelligently, while still allowing new modes to take hold in the network. In this paper, a Mode Mutation rate equal to the mode-deletion rate was used, which may not do a good enough job of pruning seldom-used modes. Furthermore, new modes may need some protection from deletion for a certain number of generations after being created. Exploring these and other ways of improving Mode Mutation is a promising direction for future work.

Another issue for future work is how to select an agent or agents from a Pareto front to use in a game. As was suggested in Section V-C, goal thresholds for each objective could be used for this purpose. However, such an approach ends up throwing away potentially useful extreme solutions on the trade-off surface. One way to make use of the whole Pareto front in a single agent is to form an ensemble of members of the Pareto front [41]. Such an approach can thus take advantage of individuals from all parts of the trade-off surface. The results for `Multinetwork` already demonstrate how useful multiple networks in a single agent can be, so it is possible that ensembles can be even more successful.

VIII. Conclusion

Two multitask games, Front/Back Ramming and Predator/Prey, were used to evaluate two methods of evolving multimodal networks: Multitask Learning and Mode Mutation. These approaches were compared against a control involving networks with a single mode, and a method of combining separately evolved networks called Multinetwork.

In the Front/Back Ramming game, where the task division is both obvious *and* balanced, Multitask Learning is very effective, but not as good as Multinetwork. Mode Mutation methods come in third, but still ahead of networks with just one mode. In Predator/Prey a form of Mode Mutation, named MM(R), tied with Multinetwork as the most effective method. MM(R) succeeded by efficiently searching the space of policies to find one mode of behavior that worked well with very little help from other modes.

Multinetwork, Multitask Learning, and Mode Mutation thus allow evolved agents to have multiple policies to fit different situations. Such an ability should prove useful in developing intelligent behaviors for challenging games consisting of multiple tasks.

Acknowledgment

This research was supported in part by the NSF under grants DBI-0939454, IIS-0757479, and IIS-0915038.

References

[1] D. Andre and A. Teller, "Evolving Team Darwin United," in *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, 1999, pp. 346–351.

[2] J. Schrum and R. Miikkulainen, "Constructing Complex NPC Behavior via Multi-Objective Neuroevolution," in *Artificial Intelligence and Interactive Digital Entertainment*, 2008. [Online]. Available: http://nn.cs.utexas.edu/?schrum:aiide08

[3] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber, "Super Mario Evolution," in *Computational Intelligence and Games*, 2009.

[4] G. N. Yannakakis and J. Hallam, "Interactive opponents generate interesting games," in *Computer Games: Artificial Intelligence, Design and Education*, 2004, pp. 240–247.

[5] R. A. Caruana, "Multitask learning," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA 15213, 1997.

[6] J. Schrum and R. Miikkulainen, "Evolving Multi-modal Behavior in NPCs," in *Computational Intelligence and Games*, 2009. [Online]. Available: http://nn.cs.utexas.edu/?schrum:cig09

[7] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving Competitive Car Controllers for Racing Games with Neuroevolution." in *Genetic and Evolutionary Computation Conference*, 2009, pp. 1179–1186.

[8] N. van Hoorn, J. Togelius, and J. Schmidhuber, "Hierarchical Controller Learning in a First-Person Shooter," in *Computational Intelligence and Games*, 2009.

[9] J. Togelius, "Evolution of a Subsumption Architecture Neurocontroller," *Journal of Intelligent and Fuzzy Systems*, pp. 15–20, 2004.

[10] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 10, 1986.

[11] T. Thompson, F. Milne, A. Andrew, and J. Levine, "Improving Control Through Subsumption in the EvoTanks Domain," in *Computational Intelligence and Games*, 2009, pp. 363–370.

[12] R. Calabretta, S. Nolfi, D. Parisi, and G. Wagner, "Duplication of Modules Facilitates the Evolution of Functional Specialization," *Artificial Life*, vol. 6, no. 1, pp. 69–84, 2000.

[13] F. Gruau, "Automatic definition of modular neural networks," *Adaptive Behavior*, vol. 3, no. 2, pp. 151–183, 1994.

[14] J. Kodjabachian and J.-A. Meyer, "Evolution and Development of Modular Control Architectures for 1D Locomotion in Six-legged Animats," *Connection Science*, vol. 10, no. 3–4, pp. 211–237, 1998.

[15] P. Verbancsics and K. O. Stanley, "Constraining Connectivity to Encourage Modularity in HyperNEAT," in *Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2011, pp. 1483–1490.

[16] V. Khare, X. Yao, B. Sendhoff, Y. Jin, and H. Wersing, "Co-evolutionary Modular Neural Networks for Automatic Problem Decomposition," in *Congress on Evolutionary Computation*, vol. 3, September 2005, pp. 2691–2698.

[17] H. H. Dam, H. A. Abbass, and C. Lokan, "Neural-based learning classifier systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 1, pp. 26–39, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4358957

[18] T. G. Dietterich, "The MAXQ Method for Hierarchical Reinforcement Learning," in *International Conference on Machine Learning*, 1998.

[19] B. Hengst, "Discovering Hierarchy in Reinforcement Learning with HEXQ," in *International Conference on Machine Learning*. Morgan Kaufmann, 2002, pp. 243–250.

[20] N. Sprague and D. Ballard, "Multiple-Goal Reinforcement Learning with Modular Sarsa(0)," in *International Joint Conference on Artificial Intelligence*, 2003.

[21] F. Tanaka and M. Yamamura, "Multitask Reinforcement Learning on the Distribution of MDPs," in *Computational Intelligence in Robotics and Automation*, vol. 3. IEEE, July 2003, pp. 1108–1113.

[22] A. Wilson, A. Fern, S. Ray, and P. Tadepalli, "Multi-Task Reinforcement Learning: a Hierarchical Bayesian Approach," in *International Conference on Machine Learning*. New York, NY, USA: ACM, 2007, pp. 1015–1022.

[23] T. Ziemke, "On 'Parts' and 'Wholes' of Adaptive Behavior: Functional Modularity and Diachronic Structure in Recurrent Neural Robot Controllers," in *Simulation of Adaptive Behavior*, 2000.

[24] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Evolving Adaptive Neural Networks with and Without Adaptive Synapses," in *Congress on Evolutionary Computation*. Piscataway, NJ: IEEE, 2003. [Online]. Available: http://nn.cs.utexas.edu/?stanley:cec03

[25] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998. [Online]. Available: http://www.cs.ualberta.ca/ sutton/book/the-book.html

[26] J. Klein, "BREVE: A 3D Environment for the Simulation of Decentralized Systems and Artificial Life," *Artificial Life*, pp. 329–334, 2003.

[27] M. Tan, "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents," in *International Conference on Machine Learning*. Morgan Kaufmann, 1993, pp. 330–337.

[28] P. Stone and M. Veloso, "Multiagent Systems: A Survey from a Machine Learning Perspective," *Autonomous Robotics*, vol. 8, no. 3, 2000.

[29] P. Rajagopalan, A. Rawal, R. Miikkulainen, M. A. Wiseman, and K. E. Holekamp, "The Role of Reward Structure, Coordination Mechanism and Net Return in the Evolution of Cooperation," in *Computational Intelligence and Games*, Seoul, South Korea, 2011. [Online]. Available: http://nn.cs.utexas.edu/?rajagopalan:cig11

[30] C. M. Fonseca and P. J. Fleming, "An Overview of Evolutionary Algorithms in Multiobjective Optimization," *Evolutionary Computation*, vol. 3, pp. 1–16, March 1995.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *Evolutionary Computation*, vol. 6, pp. 182–197, 2002.

[32] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl, "Automatic Feature Selection in Neuroevolution," in *Genetic and Evolutionary Computation Conference*, 2005. [Online]. Available: http://nn.cs.utexas.edu/?whiteson:gecco05

[33] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks Through Augmenting Topologies," *Evolutionary Computation*, vol. 10, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/keyword?stanley:ec02

[34] N. Kohl and R. Miikkulainen, "Evolving Neural Networks for Strategic Decision-Making Problems," *Neural Networks, Special issue on Goal-Directed Neural Systems*, 2009.

[35] J. Schrum and R. Miikkulainen, "Evolving Agent Behavior In Multiobjective Domains Using Fitness-Based Shaping," in *Genetic and Evolutionary Computation Conference*, July 2010. [Online]. Available: http://nn.cs.utexas.edu/?schrum:gecco10

[36] M. Waibel, L. Keller, and D. Floreano, "Genetic Team Composition and Level of Selection in the Evolution of Multi-Agent Systems," *Evolutionary Computation*, vol. 13, no. 3, 2009.

[37] B. D. Bryant and R. Miikkulainen, "Evolving Stochastic Controller Networks for Intelligent Game Agents," in *Congress on Evolutionary Computation*. Piscataway, NJ: IEEE, 2006. [Online]. Available: http://nn.cs.utexas.edu/?bryant:cec06

[38] E. Zitzler, D. Brockhoff, and L. Thiele, "The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration," in *Evolutionary Multi-Criterion Optimization*, 2007.

[39] J. Knowles, L. Thiele, and E. Zitzler, "A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers," TIK, ETH Zurich, TIK Report 214, Feb. 2006.

[40] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca, "Performance Assessment of Multiobjective Optimizers: An Analysis and Review," TIK, ETH Zurich, TIK Report 139, 2002.

[41] H. Abbass, "Pareto Neuro-evolution: Constructing Ensemble of Neural Networks Using Multi-objective Optimization," in *Congress on Evolutionary Computation*, 2003, pp. 2074–2080.

Jacob Schrum received his B.S. degree in 2006 from Southwestern University in Georgetown, Texas, where he triple-majored in Computer Science, Math and German, graduating with honors in both Computer Science and German. He received his M.S. degree in Computer Science from the University of Texas in 2009. He is currently a Ph.D. candidate at the University of Texas at Austin, where his research focuses on automatic discovery of complex multi-modal behavior, particularly in the domain of video games.

Risto Miikkulainen is a Professor of Computer Sciences at the University of Texas at Austin. He received an M.S. in Engineering from the Helsinki University of Technology, Finland, in 1986, and a Ph.D. in Computer Science from UCLA in 1990. His current research focuses on methods and applications of neuroevolution, as well as models of natural language processing, and self-organization of the visual cortex; he is an author of over 250 articles in these research areas. He is currently on the Board of Governors of the Neural Network Society, and an action editor of IEEE Transactions on Autonomous Mental Development, IEEE Transactions on Computational Intelligence and AI in Games, the Machine Learning Journal, Journal of Cognitive Systems Research, and Neural Networks.