Chapter 6
Temporal Memory Sharing in Visual Reinforcement Learning

Stephen Kelly and Wolfgang Banzhaf

Abstract Video games provide a well-defined study ground for the development of behavioural agents that learn through trial-and-error interaction with their environment, or reinforcement learning (RL). They cover a diverse range of environments that are designed to be challenging for humans, all through a high-dimensional visual interface. Tangled Program Graphs (TPG) is a recently proposed genetic programming algorithm that emphasizes emergent modularity (i.e. automatic construction of multi-agent organisms) in order to build successful RL agents more efficiently than state-of-the-art solutions from other sub-fields of artificial intelligence, e.g. deep neural networks. However, TPG organisms represent a direct mapping from input to output with no mechanism to integrate past experience (previous inputs). This is a limitation in environments with partial observability. For example, TPG performed poorly in video games that explicitly require the player to predict the trajectory of a moving object. In order to make these calculations, players must identify, store, and reuse important parts of past experience. In this work, we describe an approach to supporting this type of short-term temporal memory in TPG, and show that shared memory among subsets of agents within the same organism seems particularly important. In addition, we introduce heterogeneous TPG organisms composed of agents with distinct types of representation that collaborate through shared memory. In this study, heterogeneous organisms provide a parsimonious approach to supporting agents with task-specific functionality, image processing capabilities in the case of this work. Taken together, these extensions allow TPG to discover high-scoring behaviours for the Atari game Breakout, which is an environment it failed to make significant progress on previously.

Stephen Kelly
Michigan State University, e-mail: kellys27@msu.edu

Wolfgang Banzhaf
Michigan State University, e-mail: banzhafw@msu.edu

6.1 Introduction

Reinforcement learning (RL) is an area of machine learning that models the way living organisms adapt through interaction with their environment. RL can be characterized as learning how to map situations to actions in the pursuit of a pre-defined objective [34]. A solution, or *policy* in RL is represented by an agent that learns through episodic interaction with the problem environment. Each episode begins in an initial state defined by the environment. Over a series of discrete timesteps, the agent observes the environment (via sensory inputs), takes an action based on the observation, and receives feedback in the form of a reward signal. The agent's actions potentially change the state of the environment and impact the reward received. The agent's goal is to select actions that maximize the long-term cumulative reward. Most real-world decision-making and prediction problems can be characterized as this type of environmental interaction.

Animal and human intelligence is partially a consequence of the physical richness of our environment, and thus scaling RL to complex, real-world environments is a critical step toward sophisticated artificial intelligence. In real-world applications of RL, the agent is likely to observe the environment through a high-dimensional, visual sensory interface (e.g. a video camera). However, scaling to high-dimensional input presents a significant challenge for machine learning, and RL in particular. As the complexity of the agent's sensory interface increases, there is a significant increase in the number of environmental observations required for the agent to gain the breadth of experience necessary to build a strong decision-making policy. This is known as the curse of dimensionality (Section 1.4 of [6]). The temporal nature of RL introduces additional challenges. In particular, complete information about the environment is not always available from a single observation (i.e the environment is partially observable) and delayed rewards are common, so the agent must make thousands of decisions before receiving enough feedback to assess the quality of its behaviour [22]. Finally, real-world environments are dynamic and non-stationary [9], [26]. Agents are therefore required to adapt to changing environments without 'forgetting' useful modes of behaviour that are intermittently important over time.

Video games provide a well-defined test domain for scalable RL. They cover a diverse range of environments that are designed to be challenging for humans, all through a common high-dimensional visual interface, namely the game screen [5]. Furthermore, video games are subject to partial observability and explicitly non-stationary. For example, many games require the player to predict the trajectory of a moving object. These calculations cannot be made from observing a single screen capture. To make such predictions, players must identify, store, and reuse important parts of past experience. As the player improves, new levels of play are unlocked which may contain completely new visual and physical dynamics. As such, video games represent a rich combination of challenges for RL, where the objective for artificial agents is to play the game with a degree of sophistication comparable to that of a human player [27], [4]. The potential real-world applications for artificial agents with these capabilities is enormous.

6.2 Background

Tangled Program Graphs (TPG) are a representation for Genetic Programming (GP) with particular emphasis on *emergent* modularity through compositional evolution: the evolution of hierarchical organisms that combine multiple agents which were previously adapted independently [38], Figure 6.1.

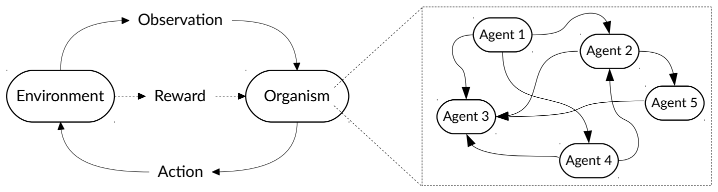


Fig. 6.1: A multi-agent organism developed through compositional evolution.

This approach leads to automatic division of labour within the organism and, over time, a collective decision-making policy emerges that is greater than the sum of its parts. The system has three critical attributes:

1. **Adaptive Complexity.** Solutions begin as single-agent organisms and then develop into multi-agent organisms through interaction with their environment. That is, the complexity of a solution is an adapted property.
2. **Input Selectivity.** Multi-agent organisms are capable of decomposing the input space such that they can ignore sensory inputs that are not important at the current point in time. This is more efficient than assuming that the complete sensory system is necessary for every decision.
3. **Modular Task Decomposition.** As multi-agent organisms develop they may subsume a variable number of stand-alone agents into a hierarchical decision-making policy. Importantly, hierarchies emerge incrementally over time, slowly growing and breaking apart through interaction with the environment. This property allows a TPG organism to adapt in non-stationary environments and avoid unlearning behaviours that were important in the past but are not currently relevant.

In the Atari video game environment, TPG matches the quality of solutions from a variety of state-of-the-art deep learning methods. More importantly, TPG is less computationally demanding, requiring far fewer calculations per decision than any of the other methods [21]. However, these TPG policies were purely reactive. They represent a direct mapping from observation to action with no mechanism to integrate past experience (prior state observations) into the decision-making process. This might be a limitation in environments with temporal properties. For example, there are Atari games that explicitly involve predicting the trajectory of a moving object (e.g. Pong) for which TPG performed poorly. A temporal memory mechanism would allow agents to make these calculations by integrating past and present environmental observations.

6.2.1 Temporal Memory

Temporal memory in sequential decision-making problems implies that behavioural agent has the ability to identify, store, and reuse important aspects of past experience when predicting the best action to take in the present. More generally, temporal memory is essential to any time series prediction problem, and has thus been investigated extensively in GP (see Agapitos et. al [2] for a broad review). In particular, an important distinction is made between *static* memory in which the same memory variables are accessed regardless of the state of the environment, and *dynamic* memory in which different environmental observations trigger access to different variables.

Dynamic memory in GP is associated with indexed memory, which requires the function set to include parameterized operations for reading and writing to specific memory addresses. Teller [35] showed that GP with memory indexing is Turing complete, i.e. theoretically capable of evolving any algorithm. Brave [8] emphasized the utility of dynamic memory access in GP applied to an agent planning problem, while Haynes [15] discuses the value of GP with dynamic memory in sequential decision-making environments that are themselves explicitly dynamic. Koza [23] proposed Automatically Defined Stores, a modular form of indexable memory for GP trees, and demonstrated its utility for solving the cart centering problem *without* velocity sensors, a classic control task that requires temporal memory.

Static memory access is naturally supported by the register machine representation in linear genetic programming [6]. For example, a register machine typically consists of a sequence of instructions that read and write from/to memory, e.g. $Register[x] = Register[x] + Register[y]$. In this case, the values contained in registers x and y may change depending on environmental input, but reference to the specific registers x and y is determined by the instruction's encoding and is not affected by input to the program. If the register content is cleared prior to each program execution, the program is said to be *stateless*. A simple form of temporal memory can be implemented by not clearing the register content prior to each execution of the program. In the context of sequential decision-making, the program retains/accumulates state information over multiple timesteps, i.e., the program is *stateful*. Alternatively, register content from timestep t may be fed back into the program's input at time $t+1$, enabling temporal memory through recurrent connections, e.g [10], [16].

Smith and Heywood [32] introduced the first memory model for TPG in particular. Their method involved a single global memory bank. TPG's underlying linear GP representation was augmented with a probabilistic write operation, enabling long and short-term memory memory stores. They also included a parameterized read operation for indexed (i.e. dynamic) reading of external memory. Furthermore, the external memory bank is ecologically global. That is, sharing is supported among the entire population such that organisms may integrate their own past experience *and* experience gained by other (independent) organisms. The utility of this memory model was demonstrated for navigation in a partially observable, visual RL environment.

In this work we propose that multi-agent TPG organisms can be extended to support dynamic temporal memory *without* the addition of specialized read/write operations. This is possible because TPG organisms naturally decompose the task both spatially *and* temporally. Specifically, each decision requires traversing one path through the graph of agents, in which only agents along the path are 'executed' (i.e. a subset of the agents in the organism). Each agent will have a unique complement of static environmental input and memory references. Since the decision path at time t is entirely dependent on the state of the environment, both state and memory access are naturally dynamic.

The intuition behind this work is that dynamic/temporal problem decomposition w.r.t input and memory access is particularly important in visual RL tasks because: 1) High-dimensional visual input potentially contains a large amount of information that is intermittently relevant to decision making over time. As such, it is advantageous if the model can parse out the most salient observational data for the current timestep and ignore the rest; and 2) RL environments with real-world complexity are likely to exhibit partial observability at multiple times scales. For example, predicting the trajectory of a moving object may require an agent to integrate a memory of the object's location at $t-1$ with the current observation at time t, or short-term memory. Conversely, there are many tasks that require integration over much longer periods of time, e.g. maze navigation [12]. These two points clearly illustrate the drawback of *autoregressive* models [2, 27], in which the issue of temporal memory is side-stepped by stacking a fixed number of the most recent environmental observations into a single 'sliding window' world view for the agent. This approach potentially increases the amount of redundant input information and limits temporal state integration to a window size fixed a priori.

6.2.2 Heterogeneous Policies and Modularity

Heterogeneous policies provide a mechanism through which multiple types of active device, entities which accept input, perform some computation, and produce output, may be combined within a stand-alone decision-making agent, eg. [17], [25]. In this work, we investigate how compositional evolution can be used to adaptively combine general-purpose and task-specific devices. Specifically, GP applied to visual problems can benefit from the inclusion of specialized image-progressing operators, e.g. [3], [24], [39]. Rather than augmenting the instruction set of *all* programs with additional operators, compositional evolution of heterogeneous policies provides an opportunity to integrate task-specific functionality in a modular fashion, where modularity is likely to improve the evolvability of the system [1], [38], [36].

6.3 Evolving Heterogeneous Tangled Program Graphs

The algorithm investigated in this work is an extension of Tangled Program Graphs [21] with additional support for temporal memory and heterogeneous policies. This section details the extended system with respect to three system components, each playing a distinct role within the emergent hierarchical structure of TPG organisms:

- A **Program** is the simplest active device, capable of processing input data and producing output, but *not* representing a stand-alone RL agent.
- A **Team of Programs** is the smallest independent decision-making organism, or agent, capable of observing the environment and taking *actions*.
- A **Policy Graph** adaptively combines multiple teams (agents) into a single hierarchical organism through open-ended compositional evolution. In this context, *open-ended* refers to the fact that hierarchical transitions are not planned a priori. Instead, the hierarchical complexity of each policy graph is a property that emerges through interaction with the problem environment.

6.3.1 Programs and Shared Temporal Memory

In this work, all programs are linear register machines [6]. Two types of program are supported: Action-value programs and Image processors.

Action-value programs have a pointer to one action (e.g. a joystick position from the video game domain) and produce one scalar *bidding* output, which is interpreted as the program's confidence that its action is appropriate given the current state of the environment. As such, the role of an action-value program is to define environmental *context* for one action, Algorithm 3. In order to support shared temporal memory, action-value programs have two register banks; one *stateless* bank that is reset prior to each program execution, and a pointer to one *stateful* bank that is only reset at the start of each episode (i.e. game start). Stateless register banks are private to each program, while stateful banks are stored in a dedicated *memory* population and may be shared among multiple programs (This relationship is illustrated in the lower-left of Figure 17.2). Shared memory allows multiple programs to communicate within a single timestep or integrate information across multiple timesteps. In effect, shared memory implies that each program has a parameterized number of *context* outputs (see $Registers_{shared}$ in Table 6.2).

Image processing programs have *context* outputs only. They perform matrix manipulations on the raw pixel input and store the result in shared memory accessible to all other programs, Algorithm 4. In addition to private and shared register memory, image processors have access to a task-specific type of shared memory in the form of a single, global image buffer matrix with the same dimensionality as the environment's visual interface. The buffer matrix is stateful, reset only at the start of each episode. This allows image processors to accumulate full-screen manipulations over the entire episode. Note that image-processor programs have no bidding

Algorithm 3 Example **action-value program**. Each program contains one *private stateless* register bank, R_p, and a pointer to one *shareable stateful* register bank, R_s. R_p is reset prior to each execution, while R_s is reset (by an external process) at the start of each episode. Programs may include two-argument instructions of the form $R[i] \leftarrow R[j] \circ R[k]$ in which $\circ \in \{+,-,x,\div\}$; single-argument instructions of the form $R[i] \leftarrow \circ(R[k])$ in which $\circ \in \{cos, ln, exp\}$; and a conditional statement of the the form IF $(R[i] < R[k])$ THEN $R[i] \leftarrow -R[i]$. The second source variable, $R[k]$, may reference either memory bank or a state variable (pixel), while the target and first source variables ($R[i]$ and $R[j]$) may reference either the stateless or stateful memory bank only. Action-value programs always return the value stored in $R_p[0]$ at the end of execution.

```
1: $R_p \leftarrow 0$                                # reset private memory bank $R_p$

2: $R_p[0] \leftarrow R_s[0] - Input[3]$
3: $R_s[1] \leftarrow R_p[0] \div R_s[7]$
4: $R_s[2] \leftarrow Log(R_s[1])$
5: **if then**$(R_p[0] < R_s[2])$
6:     $R_p[0] \leftarrow -R_p[0]$
7: **end if**
8: **return** $R_p[0]$
```

output because they do not contribute directly to action selection. Their role is to preprocess input data for action-value programs, and this contribution is communicated to action-value programs through shared register memory (This relationship is illustrated in the lower-left of Figure 17.2).

Memory sharing implies that a much larger proportion of program code is now *effective*, since an effective instruction is one that effects the final value in the bidding output ($R_p[0]$) or any of the shared registers, R_s. As a result, sharing memory incurs a significant computational cost relative to programs without shared memory, since fewer ineffective instructions, or *introns*, can be removed prior to program execution. In addition, shared memory implies that the order of program execution within a team now potentially impacts bidding outputs. In this work, the order of program execution within a team remains fixed, but future work could investigate mutation operators that modify execution order. Program variation operators are listed in Table 6.2, providing an overview of how evolutionary search is focused on particular aspects of program structure. In short, program length and content, as well as the degree of memory sharing, are all adapted properties.

Algorithm 4 Example **image-processor program**. As with action-value programs, image-processor programs contain one *private stateless* register bank, R_p, and a pointer to one *shareable stateful* register bank, R_s. In addition, all image processors have access to a global buffer matrix, S, which is reset (by an external process) at the start of each episode. Image-processor programs accept either the raw pixel screen or the shared buffer as input. Depending on the operation, the result is stored in R_p, R_s, or back into the shared buffer S. Unlike the operations available to action-value programs, some image processing operations are parameterized by values stored in register memory or sampled directly form input. This opens a wide range of possibilities for image processing instructions, a few of which are illustrated in this algorithm. Table 6.1 provides a complete list of image processing operations used in this work.

```
1: $R_p \leftarrow 0$                                        # reset private memory bank $R_p$

2: $S \leftarrow AddC(Screen, R_s[0])$                       # Add an amount to each image pixel
3: $S \leftarrow Div(Screen, S)$                             # Divide the pixel values of two images
4: $S \leftarrow Sqrt(S)$                                    # Take the square root of each pixel
5: $R_p[2] \leftarrow MaxW(S, R_s[0], R_s[5], R_s[7])$       # Store max of parameterized window
6: $R_s[2] \leftarrow MeanW(S, R_s[3], R_s[1], R_p[2])$      # Store mean of parameterized window
```

6.3.2 *Cooperative Decision-Making with Teams of Programs*

Individual action-value programs have only one action pointer, and can therefore never represent a complete solution independently. A team of programs represents a complete solutions by grouping together programs that collectively map environmental observations (e.g. the game screen) to *atomic* actions (e.g. joystick positions). This is achieved through a bidding mechanism. In each timestep, every program in the team will execute, and the team then takes the action pointed to by the action-value program with the highest output. This process repeats at each timestep from the initial episode condition to the end of an episode. When the episode ends, due to a *GameOver* signal from the environment or an episode time constraint is reached, the team as a whole is assigned a fitness score from the environment (i.e. the final game score). Since decision-making in TPG is a strictly collective process, programs have no individual concept of fitness. Team variation operators may add, remove, or modify programs in the team, with parameters listed in Table 6.2.

In this work, new algorithmic extensions at the team level are twofold: 1) Teams are heterogeneous, containing action-value programs *and* image processing programs; and 2) All programs have access to shareable stateful memory. Figure 17.2 illustrates these extensions.

Table 6.1: Operations available to image-processor programs. These operations were selected based on their previous application in GP applied to image classification tasks [3]. See Algorithm 4 for an example program using a subset of these operations.

Operation	Parameters	Description
Add	Image, Image	Add pixel values of two images
Sub	Image, Image	Subtract pixel values of two images
Div	Image, Image	Divide pixel values of two images
Mul	Image, Image	Multiply pixel values of two images
Max2	Image, Image	Pixel-by-pixel max of two images
Min2	Image, Image	Pixel-by-pixel min of two images
AddC	Image, x	Add integer x to each pixel
SubC	Image, x	Subtract integer x from each pixel
DivC	Image, x	Divide each pixel by integer x
MulC	Image, x	Multiply each pixel by integer x
Sqrt	Image	Take the square root of each pixel
Ln	Image	Take the natural log of each pixel
Mean	Image, x	Uses a sliding window of size x and replaces the centre pixel of the window with the mean of the window
Max	Image, x	As Mean, but takes the maximum value
Min	Image, x	As Mean, but takes the minimum value
Med	Image, x	As Mean, but takes the median value
MeanA	Image, 3 Int (x, y, size)	Returns the mean value of the pixels contained in a window of *size*, centred at x, y in the image
StDevA	Image, 3 Int (x, y, size)	Returns standard deviation
MaxA	Image, 3 Int (x, y, size)	Returns maximum value
MinA	Image, 3 Int (x, y, size)	Returns minimum value

6.3.3 Compositional Evolution of Tangled Program Graphs

This section details how teams and programs are coevolved, paying particular attention to emergent hierarchical transitions. Parameters are listed in Table 6.2.

Evolution begins with a population of R_{size} teams, each containing at least one program of each type, and a max of ω programs in total. Programs are created in pairs with one shared memory bank between them (See left-hand-side of Figure 17.2). Program actions are initially limited to task-specific (atomic) actions, Figure 17.2. Throughout evolution, program variation operators are allowed to introduce actions that index other teams within the team population. To do so, when a program's action is modified, it may reference either a different atomic action or any team created in a previous generation. Specifically, the action set from which new program actions are sampled will correspond to the set of atomic actions, A,

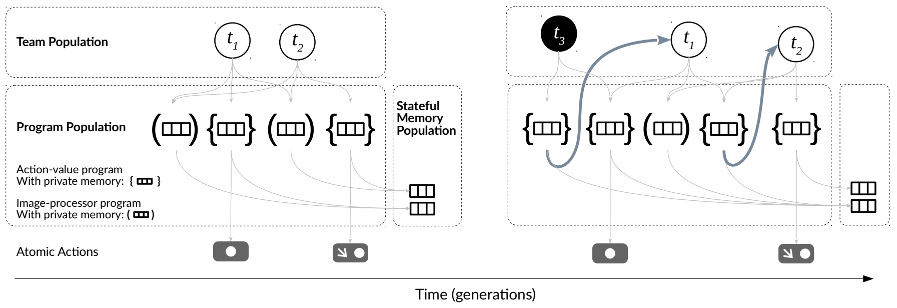

Team Population
Program Population
Action-value program With private memory: { }
Image-processor program With private memory: ( )
Stateful Memory Population
Atomic Actions
Time (generations)

Fig. 6.2: Illustration of the relationship between teams, programs, and shared memory in heterogeneous TPG.

with probability p_{atomic}, and will otherwise correspond to the set of teams present from any previous generation. In effect, action pointer mutations are the primary mechanism by which TPG supports compositional evolution, adaptively recombining multiple (previously independent) teams into variably deep/wide directed graph structures, or *policy graphs*, right-hand-side of Figure 17.2. The hierarchical interdependency between teams is established entirely through interaction with the task environment. Thus, more complex structures can emerge as soon as they perform better than simpler solutions. The utility of compositional evolution is empirically demonstrated in Section 6.4.2.

Table 6.2: Parameterization of Team and Program populations. For the team population, p_{mx} denotes a mutation operator in which: $x \in \{d,a\}$ are the prob. of deleting or adding a program respectively; $x \in \{m,n\}$ are the prob. of creating a new program, changing the program action pointer, and changing the program shared memory pointer respectively. ω is the max initial team size. For the program population, p_x denotes a mutation operator in which $x \in \{delete, add, mutate, swap\}$ are the prob. for deleting, adding, mutating, or reordering instructions within a program. p_{atomic} is the probability of a modified action pointer referencing an atomic action.

Team population			
Parameter	Value	Parameter	Value
R_{size}	1000	R_{gap}	50% of *Root* Teams
p_{md}, p_{ma}	0.7	ω	60
p_{mm}	0.2	p_{mn}, p_{ms}	0.1

Program population			
Parameter	Value	Parameter	Value
$Registers_{private}$	8	$maxProgSize$	100
$Registers_{shared}$	8	p_{atomic}	0.99
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

Decision-making in a policy graph begins at the root team (e.g. t_3 in Figure 17.2), where each program in the team will produce one bid relative to the current state observation, $\vec{s}(t)$. Graph traversal then follows the program with the largest bid, repeating the bidding process for the same state, $\vec{s}(t)$, at every team along the path until an atomic action is reached. Thus, in sequential decision-making tasks, the policy graph computes one path from root to atomic action at every time step, where only a subset of programs in the graph (i.e those in teams along the path) require execution.

As hierarchical structures emerge, only root teams (i.e. teams that are not referenced as any program's action) are subject to modification by the variation operators. As such, rather than pre-specify the desired team population size, only the number of root teams to maintain in the population, or R_{size}, requires prior specification. Evolution is driven by a generational GA such that the worst performing root teams (50% of the root population, or R_{gap}) are deleted in each generation and replaced by offspring of the surviving roots. After team deletion, programs that are not part of any team are also deleted. As such, selection is driven by a symbiotic relationship between programs and teams: teams will survive as long as they define a complementary group of programs, while individual programs will survive as long as they collaborate successfully within a team. The process for generating team offspring uniformly samples and clones a root team, then applies mutation-based variation operators to the cloned team, as listed in Table 6.2. Complete details on TPG are available in [21] and [19].

6.4 Empirical Study

The objective of this study is to evaluate heterogeneous TPG with shared temporal memory for object tracking in visual RL. This problem explicitly requires an agent to develop short-term memory capabilities. For an evaluation of TPG in visual RL with longer-term memory requirements see [32].

6.4.1 Problem Environments

In order to compare with previous results, we consider the Atari video game Breakout, for which the initial version of TPG failed to learn a successful policy [21]. Breakout is a vertical tennis-inspired game in which a single ball is released near the top of the screen and descends diagonally in either left or right direction, Figure 3(a). The agent observes this environment through direct screen capture, a 64×84 pixel matrix[1] in which each pixel has a colour value between 0 and 128. The player

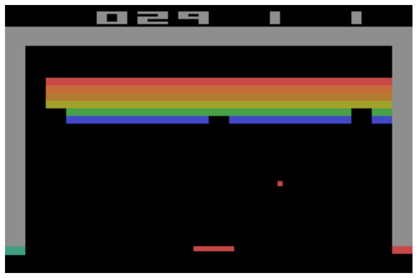
[1] This screen resolution corresponds to 40% of the raw 210 Atari screen resolution. TPG has previously been shown to operate under the full Atari screen resolution [21]. The focus of this study is temporal memory, and the down sampling is used here to speed up empirical evaluations.

controls the horizontal movement of a paddle at the bottom of the screen. Selecting form 4 *atomic* actions in each timestep, $A \in \{Serve, Left, Right, NoAction\}$, the goal is to maneuver the paddle such that it makes contact with the falling ball, causing it to ricochet up towards the brick ceiling and clear bricks one at a time. If the paddle misses the falling ball, the player looses a turn. The player has three turns to clear two layers of brick ceiling. At the end of each episode, the game returns a reward signal which increases relative to the number of bricks eliminated. The primary skill in breakout is simple: the agent must integrate the location of the ball over multiple timesteps in order to predict its trajectory and move the paddle to the correct horizontal position. However, the task is dynamic and non-trivial because, as the game progresses, the ball's speed increases, its angle varies more widely, and the width of the paddle shrinks. Furthermore, *sticky actions* [27] are utilized such that agents stochastically skip screen frames with probability $p = 0.25$, with the previous action being repeated on skipped frames. Sticky actions have a dual purpose in the ALE: 1) artificial agents are limited to roughly the same reaction time as a human player; and 2) stochasticity is present throughout the entire episode of gameplay.

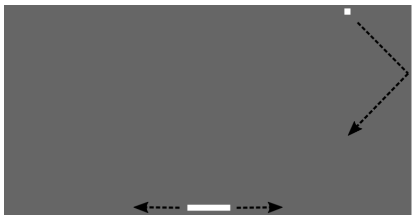
The Atari simulator used in this work, The Arcade Learning Environment (ALE) [5], is computational demanding. As such, we conduct an initial study in a custom environment that models only the ball tracking task in breakout. This "ball catching" task is played on a 64×32 grid (i.e. representing roughly the bottom 3/4 of the Breakout game screen) in which each tile, or pixel, can be one of two colours represented by the values 0 (no entity present) and 255 (indicating either the ball or paddle is present at this pixel location), Figure, 3(b). The ball is one pixel large and is stochastically initialized in one of the 64 top-row positions at the start of each episode. The paddle is 3 pixels wide and is initialized in the centre of the bottom row. The ball will either fall straight down (probability = 0.33) or diagonally, moving one pixel down and one pixel to the left or right (chosen with equal probability at time $t = 1$) in each timestep. If a diagonally-falling ball hits either wall, its horizontal direction is reversed. The agent's objective is to select one of 3 paddle movements in each timestep, $A \in \{Left, Right, NoAction\}$, such that the paddle makes contact with the falling ball. The paddle moves twice as fast as the ball, i.e. 2 pixels at a time in either direction. An episode ends when the ball reaches the bottom row, at which point the game returns a reward signal of 1.0 if the ball and paddle overlap, and 0 otherwise. As in Breakout, success in this task requires the agent to predict the trajectory of the falling ball *and* correlate this trajectory with the current position of the paddle in order to select appropriate actions.

6.4.2 Ball Catching: Training Performance

Four empirical comparisons are considered in the ball catching environment, with 10 independent runs performed for each experimental case. As discussed in Section 11.1, visual RL policies require a breadth of experience interacting with the problem environment before their fitness can be estimated with a sufficient degree of gener-



(a) Breakout



(b) Ball Catching

Fig. 6.3: Screenshots of the two video game environments utilized in this work.

ality. As such, in each generation we evaluate every TPG policy in 40 episodes and let the mean episode score represent their fitness. Curves in Figure 6.4 represent the fitness of the champion individual over 2000 generations, which is equivalent to roughly 12 hours of wall-clock time. Each line is the median over 10 independent runs.

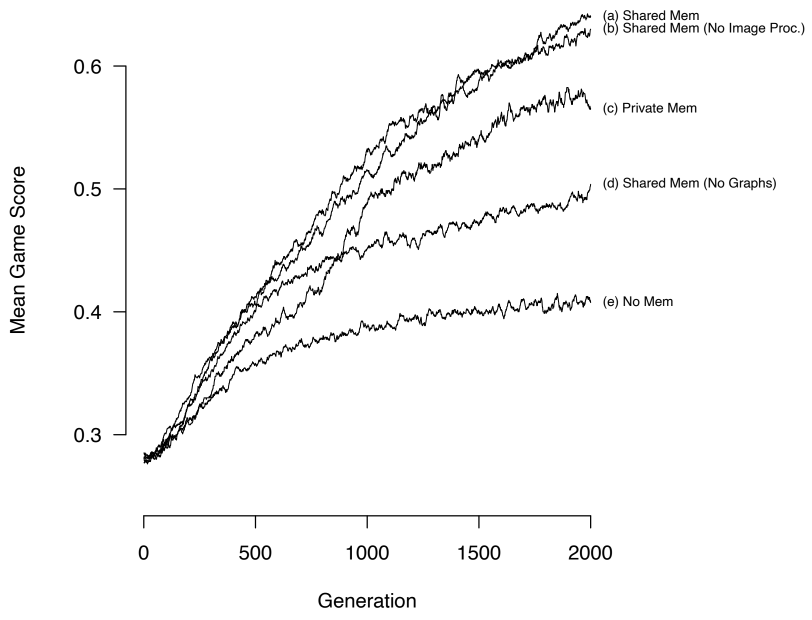


Fig. 6.4: Fraction of successful outcomes (Mean Game Score) in 40 episodes for the champion individual at each generation. Each line is the median over 10 independent runs for experimental cases (a) - (e). See Section 6.4.2 text for comparative details.

Figure 6.4 (a) is the training curve for heterogeneous TPG with the capacity for shared stateful memory as described in Section 6.3. For reference, a mean game score of 0.65 indicates that the champion policy successfully maneuvered the paddle to meet the ball 65% of the time over 40 episodes.

Figure 6.4 (b) is the training curve for TPG with shared memory but *without* image-processor programs. While the results are not significantly different than case (a), heterogeneous TPG did not hinder progress in any way. Furthermore, the single best policy in either (a) or (b) was heterogeneous and *did* make use of image-processor programs, indicating that the method has potential.

Figure 6.4 (c) is the training curve for heterogeneous TPG without the capacity for *shared* memory. In this case, each program is initialized with a pointer to one *private* stateful register bank. Mutation operators are not permitted to modify memory pointers ($p_{ms} = 0$). The case *with* memory sharing exhibits significantly better median performance after very few generations (≈ 100).

Figure 6.4 (d) shows the training curve for heterogeneous TPG without the capacity to build policy graphs. In this case, team hierarchies can never emerge ($p_{atomic} = 1.0$). Instead, adaptive complexity is supported by allowing root teams to acquire an unbounded number of programs ($\omega = \infty$). The weak result clearly illustrates the advantage of emergent hierarchical transitions for this task. As discussed in Section 6.2.1, one possible explanation for this is the ability of TPG policy graphs to decompose the task spatially by defining an appropriate subset of inputs (pixels) to consider in each timestep, and to decompose the task temporally by identifying, storing, and reusing subsets of past experience through dynamic memory access. Without the ability to build policy graphs, input and memory indexing would be static, i.e. the same set of inputs and memory registers would be accessed in every timestep regardless of environmental state.

Figure 6.4 (e) shows the training curve for heterogeneous TPG without the capacity for stateful memory. In this case, both private and shared memory registers are cleared prior to each program execution. This is equivalent to equipping programs with 16 *stateless* registers. The case with shared temporal memory achieves significantly better policies after generation ≈ 500, and continues to gradually discover increasingly high scoring policies up until the runs terminate at generation 2000. This comparison clearly illustrates the advantage that temporal memory provides for TPG organisms in this domain. Without the ability to integrate observations over multiple timesteps, even the champion policies are only slightly better than random play.

6.4.3 Ball Catching: Solution analysis

Section 6.4.2 established the effectiveness of heterogeneous TPG with shared temporal memory in a visual object tracking task (i.e. ball-catching). The following analysis confirms that champion policies rely on shared temporal memory to succeed at this task under test conditions. Box plots in Figure 6.5 summarize the mean game score (over 30 episodes) for the single champion policy from 20 runs[2]. The distribution labeled ’Shared Mem’ indicates that the median success rate for these 20 champions is $\approx 76\%$. The box labeled ’Mem (No Sharing)’ summarizes scores for the *same* 20 policies when their ability to share memory is suppressed. The decrease in performance indicates that policies are indeed exploiting shared temporal memory in solving this task. The box labeled ’No Mem’ provides test scores for these policies when all memory registers are stateless (i.e. reset prior to each program execution), again confirming that temporal memory plays a crucial role in the behaviour of these policies. Without temporal memory, the champion policies are often no better than a policy that simply selects actions at random, or ’Rand’ in Figure 6.5.

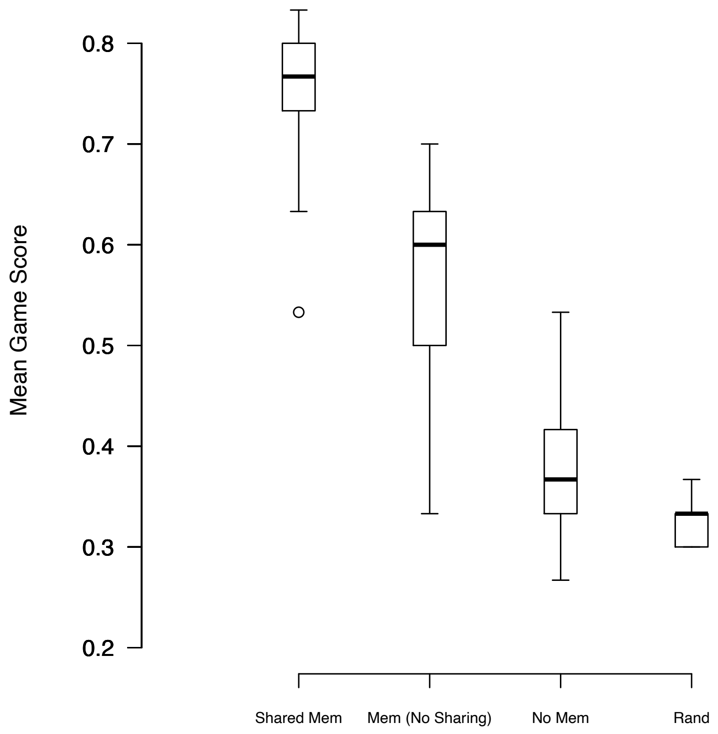


Fig. 6.5: Fraction of successful outcomes (Mean Game Score) in 30 test episodes for the champion policies from the case of heterogeneous TPG with shared temporal memory, Figure 6.4 (a). Box plots summarize the results from 20 independent runs. See Section 6.4.3 text for details on each distribution.

[2] An additional 10 runs were conducted for this analysis relative to the 10 runs summarized in Figure 6.4 (a).

6.4.4 Atari Breakout

In this Section, the most promising TPG configuration identified under the ball-catching task, or heterogeneous TPG with shared temporal memory, is evaluated in the Atari game Breakout (Section 6.4.1). The computational cost of game simulation in the ALE precludes evaluating each individual policy in 40 episodes per generation during training. In Breakout, each policy is evaluated in only 5 episodes per generation. This limits the generality of fitness estimation but is sufficient for a proof-of-concept test of our methodology in a challenging and popular visual RL benchmark. Figure 6.6 provides the training curves for 10 independent Breakout runs. In order to score any points, policies must learn to serve the ball (i.e. select the *Serve* action) whenever the ball does not appear on screen. This skill appears relatively quickly in most of the runs in Figure 6.6. Next, static paddle locations (e.g. moving the paddle to the far right after serving the ball and leaving it there) can occasionally lead to $\approx 11-15$ points. In order to score $\approx 20-50$ points, policies must surpass this somewhat degenerate local optima by discovering a truly responsive strategy in which the paddle and ball make contact several times at multiple horizontal positions. Finally, policies that learn to consistently connect the ball and paddle will create a hole in the brick wall. When this is achieved, the ball can pass through all layers of brick and become trapped in the upper region of the world where it will bounce around clearing bricks from the top down and accumulating scores above 100.

Table 6.3 lists Breakout test scores for several recent visual RL algorithms. Previous methods either employed stateless models that failed to achieve a high score (TPG, CGP, HyperNeat) or side-stepped the requirement for temporal memory by using autoregressive, sliding window state representations. Heterogeneous TPG with shared memory (HTPG-M) is the highest scoring algorithm that operates directly from screen capture *without* an autoregressive state, and roughly matches the test scores from 3 of the 4 deep learning approaches that *do* rely on autoregressize state.

Table 6.3: Comparison of Breakout test scores (mean game score over 30 episodes) for state-of-the art methods that operate directly from screen capture. Heterogeneous TPG with shared memory (HTPG-M) is the highest scoring algorithm that operates *without* an autoregressive, sliding window state representation. 'Human' is the score achieved by a "professional" video game tester reported in [27]. Scores for comparator algorithms are from the literature: Double [13], Dueling [37], Prioritized [31], A3C FF [28], A3C LSTM [28], TPG [21], HyperNeat [14], CGP [39].

Human	Double	Dueling	Prioritized	A3C LSTM	TPG	HTPG-M	HyperNeat	CGP
31.8	368.9	411.6	371.6	766.8	12.8	374.2	2.8	13.2

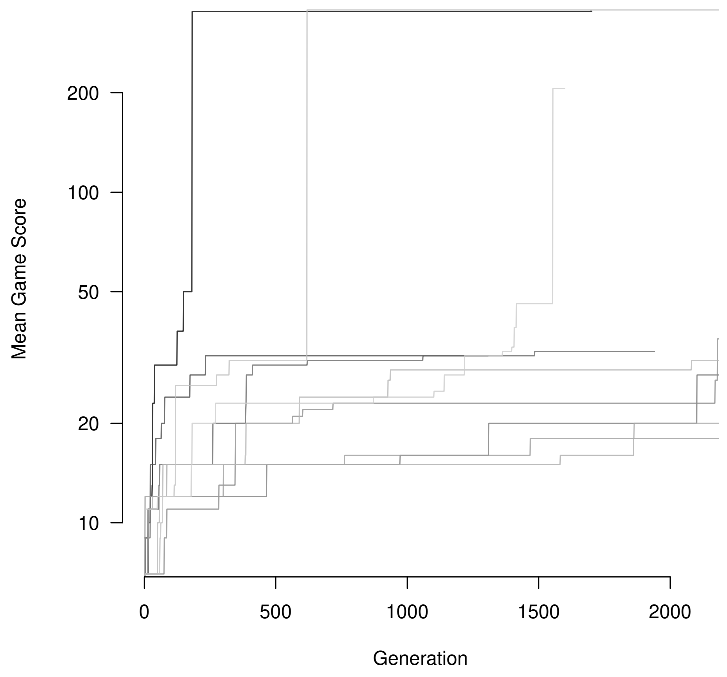


Fig. 6.6: Fitness (mean game score) for the single champion policy in each of 10 independent Breakout runs. Scores are averaged over 5 episodes. For clarity, line plots show the fitness of the best policy discovered up to each generation.

6.5 Conclusions and Future Work

We have proposed a framework for shared temporal memory in TPG which significantly improves the performance of agents in a partially observable, visual RL problem with short-term memory requirements. This study confirms the significance of private temporal memory for individual programs as well as the added benefit of memory sharing among multiple programs in a single organism, or policy graph. No specialized program instructions are required to support dynamic memory access. The nature of temporal memory access and the degree of memory sharing among programs are both emergent properties of an open-ended evolutionary process. Future work will investigate this framework in environments with long-term partial observability. Multi-task RL is a specific example of this [21], where the agent must build short-term memory mechanisms *and* integrate experiences from multiple ongoing tasks. Essentially, the goal will be to construct shared temporal memory at multiple time scales, e.g. [17]. This will most likely require a mechanism to trigger the erosion of non-salient memories based on environmental stimulus, or active forgetting [11].

Supporting multiple program representations within a single heterogeneous organism is proposed here as an efficient way to incorporate domain knowledge in

TPG. In this study, the inclusion of domain-specific image processing operators was not crucial to building strong policies, but it did not hinder performance in any way. Given the success of shared memory as a means of communication within TPG organisms, future work will continue to investigate how heterogeneous policies might leverage specialized capabilities from a wider of variety bio-inspired virtual machines. Image processing devices that model visual attention are of particular interest, e.g. [33], [29].

Acknowledgements Stephen Kelly gratefully acknowledges support from the NSERC Postdoctoral Fellowship program. Computational resources for this research were provided by Michigan State University through the Institute for Cyber-Enabled Research (icer.msu.edu) and Compute Canada (computecanada.ca).

References

1. A. Simon, H.: The architecture of complexity. Proceedings of the American Philosophical Society **106**, 467–482 (1962)
2. Agapitos, A., Brabazon, A., O'Neill, M.: Genetic programming with memory for financial trading. In: G. Squillero, P. Burelli (eds.) Applications of Evolutionary Computation, pp. 19–34. Springer International Publishing (2016)
3. Atkins, D., Neshatian, K., Zhang, M.: A domain independent genetic programming approach to automatic feature extraction for image classification. In: 2011 IEEE Congress of Evolutionary Computation (CEC), pp. 238–245 (2011)
4. Beattie, C., Leibo, J.Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., Petersen, S.: Deepmind lab. arXiv preprint arXiv:1612.03801 (2016)
5. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research **47**, 253–279 (2013)
6. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag (2006)
7. Brameier, M., Banzhaf, W.: Linear Genetic Programming, 1st edn. Springer (2007)
8. Brave, S.: The evolution of memory and mental models using genetic programming. In: Proceedings of the 1st Annual Conference on Genetic Programming, pp. 261–266. MIT Press (1996)
9. Choi, S.P.M., Yeung, D.Y., Zhang, N.L.: An environment model for nonstationary reinforcement learning. In: S.A. Solla, T.K. Leen, K. Müller (eds.) Advances in Neural Information Processing Systems 12, pp. 987–993. MIT Press (2000)
10. Conrads, M., Nordin, P., Banzhaf, W.: Speech sound discrimination with genetic programming. In: W. Banzhaf, R. Poli, M. Schoenauer, T.C. Fogarty (eds.) Genetic Programming, pp. 113–129. Springer Berlin Heidelberg (1998)
11. Davis, R.L., Zhong, Y.: The Biology of ForgettingA Perspective. Neuron **95**(3), 490–503 (2017)
12. Greve, R.B., Jacobsen, E.J., Risi, S.: Evolving neural turing machines for reward-based learning. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, pp. 117–124. ACM (2016)
13. Hasselt, H.v., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16, pp. 2094–2100. AAAI Press (2016)

14. Hausknecht, M., Lehman, J., Miikkulainen, R., Stone, P.: A neuroevolution approach to general Atari game playing. IEEE Transactions on Computational Intelligence and AI in Games **6**(4), 355–366 (2014)
15. Haynes, T.D., Wainwright, R.L.: A simulation of adaptive agents in a hostile environment. In: Proceedings of the 1995 ACM Symposium on Applied Computing, SAC '95, pp. 318–323. ACM (1995)
16. Hintze, A., Edlund, J.A., Olson, R.S., Knoester, D.B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P.D., Sheneman, L., Goldsby, H., Bohm, C., Adami, C.: Markov brains: A technical introduction. arXiv preprint 1709.05601 (2017)
17. Hintze, A., Schossau, J., Bohm, C.: The evolutionary buffet method. In: W. Banzhaf, L. Spector, L. Sheneman (eds.) Genetic Programming Theory and Practice XVI, Geenetic and Evolutionary Computation Series, pp. 17–36. Springer (2018)
18. Jaderberg, M., Czarnecki, W.M., Dunning, I., Marris, L., Lever, G., Castaeda, A.G., Beattie, C., Rabinowitz, N.C., Morcos, A.S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J.Z., Silver, D., Hassabis, D., Kavukcuoglu, K., Graepel, T.: Human-level performance in 3d multiplayer games with population-based reinforcement learning. Science **364**(6443), 859–865 (2019)
19. Kelly, S.: Scaling genetic programming to challenging reinforcement tasks through emergent modularity. Ph.D. thesis, Faculty of Computer Science, Dalhousie University (2018)
20. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. Evolutionary Computation **26**(3), 347–380 (2018)
21. Kelly, S., Smith, R.J., Heywood, M.I.: Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial, pp. 37–57. Springer International Publishing (2019)
22. Kober, J., Peters, J.: Reinforcement learning in robotics: A survey. In: M. Wiering, M. van Otterio (eds.) Reinforcement Learning, pp. 579–610. Springer (2012)
23. Koza, J.R., Andre, D., Bennett, F.H., Keane, M.A.: Genetic Programming III: Darwinian Invention & Problem Solving, 1st edn. Morgan Kaufmann Publishers Inc. (1999)
24. Krawiec, K., Bhanu, B.: Visual learning by coevolutionary feature synthesis. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **35**(3), 409–425 (2005)
25. Lalejini, A., Ofria, C.: What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP. In: W. Banzhaf, L. Spector, L. Sheneman (eds.) Genetic Programming Theory and Practice XVI, pp. 103–121. Springer International Publishing (2019)
26. Lughofer, E., Sayed-Mouchaweh, M.: Adaptive and on-line learning in non-stationary environments. Evolving Systems **6**(2), 75–77 (2015)
27. Machado, M.C., Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M., Bowling, M.: Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. J. Artif. Int. Res. **61**(1), 523–562 (2018)
28. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: M.F. Balcan, K.Q. Weinberger (eds.) Proceedings of The 33rd International Conference on Machine Learning, *Proceedings of Machine Learning Research*, vol. 48, pp. 1928–1937. PMLR (2016)
29. Mnih, V., Heess, N., Graves, A., Kavukcuoglu, K.: Recurrent models of visual attention. In: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pp. 2204–2212. MIT Press (2014)
30. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
31. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. In: International Conference on Learning Representations (2016)
32. Smith, R.J., Heywood, M.I.: A model of external memory for navigation in partially observable visual reinforcement learning tasks. In: L. Sekanina, T. Hu, N. Lourenço, H. Richter, P. García-Sánchez (eds.) Genetic Programming, pp. 162–177. Springer International Publishing (2019)

33. Stanley, K.O., Miikkulainen, R.: Evolving a Roving Eye for Go. In: T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, K. Deb (eds.) Genetic and Evolutionary Computation GECCO 2004, vol. 3103, pp. 1226–1238. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
34. Sutton, R.R., Barto, A.G.: Reinforcement Learning: An introduction. MIT Press (1998)
35. Teller, A.: Turing completeness in the language of genetic programming with indexed memory. In: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, vol. 1, pp. 136–141 (1994)
36. Wagner, G.P., Altenberg, L.: Perspective: Complex adaptations and the evolution of evolvability. Evolution **50**(3), 967–976 (1996)
37. Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., De Freitas, N.: Dueling network architectures for deep reinforcement learning. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ICML'16, pp. 1995–2003. JMLR.org (2016)
38. Watson, R.A., Pollack, J.B.: Modular interdependency in complex dynamical systems. Artificial Life **11**(4), 445–457 (2005)
39. Wilson, D.G., Cussat-Blanc, S., Luga, H., Miller, J.F.: Evolving simple programs for playing atari games. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18, pp. 229–236. ACM (2018)