Emergent Policy Discovery for Visual Reinforcement Learning through Tangled Program Graphs: A Tutorial

Stephen Kelly[1], Robert J. Smith[1], and Malcolm I. Heywood[1]

[1]*Faculty of Computer Science, Dalhousie University, Halifax, NS. Canada*

Article appears at GPTP XVI under Springer copyright 2019.[*]
`https://link.springer.com/chapter/10.1007/978-3-030-04735-1_3`

Abstract

Tangled Program Graphs (TPG) represents a framework by which multiple programs can be organized to cooperate and decompose a task with minimal a priori information. TPG agents begin with least complexity and incrementally coevolve to discover a complexity befitting the nature of the task. Previous research has demonstrated the TPG framework under visual reinforcement learning tasks from the Atari Learning Environment and VizDoom first person shooter game that are competitive with those from Deep Learning. However, unlike Deep Learning the emergent constructive properties of TPG results in solutions that are orders of magnitude simpler, thus execution never needs hardware support. In this work, our goal is to provide a tutorial overview demonstrating how the emergent properties of TPG have been achieved as well as providing specific examples of decompositions discovered under the VizDoom task.

1 Introduction

Visual reinforcement learning represents the direct application of reinforcement learning algorithms to frame (pixel) data from camera or video sources. The learning agent is therefore able to interact with the environment more directly than previously possible, i.e. there are no a priori decisions made regarding what features are useful/ important, potentially reducing sources of bias. To date, such approaches have been dominated by results from deep learning that have successfully minimized the amount of pre-processing necessary to the source images (typically no more than image cropping

[*]This version of the article includes colour illustrations/screen captures that could not be included in the book chapter published by Springer.

with sequential image averaging) while demonstrating the ability to better the performance of humans [27].

Some of the rationale for the ability of deep learning to provide state-of-the-art performance in visual reinforcement learning tasks has been attributed to the explicit manner in which the initially high-dimensional sensory input (pixels) is encoded into an efficient low(er) dimensional representation. Having found an appropriate encoding though a highly modular hierarchical (deep learning) architecture, a decision making component is simultaneously trained to provide the agent's policy (typically a multi-layer perceptron). Also of importance was achieving these results with a common topology and hyperparameters regardless of task (e.g. 49 Atari gaming titles). Most research since the initial pioneering report by Mnih et al. [27] has concentrated on either improving on the initial formulation of reinforcement learning employed with deep learning (of which there are many, see for example [23]) or suggesting approaches for evolving different parts of the deep learning architecture. The latter would imply that the complexity of the deep learning architecture would more closely reflect the underlying complexity of the task, e.g. [28].

In this work, we review a recent result in which a completely different approach is taken, one in which program modularity plays a central role; hereafter Tangled Program Graphs (TPG) [12, 13, 30]. Thus, instead of learning to encode a lower dimensional representation, we learn how to connect together multiple modules (teams of programs). Each module individually attempts to make decisions based on a low dimensional policy, thus modules decompose the task based on very small subsets of pixel information. In this way the composition of modules and the connectivity between modules is all emergent. Finally, only a fraction of an agent's complement of modules are actually utilized per decision. Thus, as an agent reacts to different states, different subsets of the modules respond.[1] All of this results in an exceptionally efficient framework for decision making under visual reinforcement learning tasks.

In the following, we review related work (Section 2) and characterize visual reinforcement learning (Section 3). A tutorial overview to the TPG framework is presented in Section 4, before a case study is presented on decision making using TPG in the Atari video game and VizDoom environments (Sections 5 and 6 respectively). A concluding discussion is made in Section 7.

2 Related Work

TPG emphasizes the development of interrelationships between programs, ultimately resulting in the emergence of a (tangled) graph. However, this came about as a generalization of earlier research into teaming metaphors in genetic programming. With this in mind, we provide a brief survey of related work from these two perspectives. Naturally, there are may other interesting developments involving graphs and programs, for example Cartesian GP [26], gene regulatory networks [2], and graph programming [1]. However, these developments have tended to focus on using graphs to express the

[1]In contrast, even once trained, the convolutional operation central to deep learning results in orders of magnitude higher computational cost.

interrelation between individual instructions, whereas the contribution of TPG lies in providing for the emergent organization of the interrelation *between* programs.

2.1 Evolving graphs

Fogal pioneered the evolution of Finite State Machines for sequence learning tasks [8]. However, such models required transition rules for all combinations of states and inputs, limiting their scalability.

PADO appeared in 1996 and expressed solutions as a graph of programs [34]. The graph had a start node and finish node. Each node also had a program that manipulated its own stack, as well as supporting the indexing of global memory. The path through the graph was determined by a single conditional instruction associated with each node. Execution began with the program at the start node and continued until a time out or the finish node was encountered. Given that the input remains unchanged,[2] it appears that the state of programs local to each node is retained. It is not clear how much of the graph was developed as an emergent property (e.g. not clear if the number of nodes predefined / constant throughout evolution), but it did appear that most if not all the nodes of the graph are visited during each execution.

More recently Genetic Network Programming (GNP) has been proposed in which a fixed number of graph nodes are declared a priori and each node has to be one of two 'types': conditional or action [22, 17]. There is a fixed number of nodes (constant across evolution and common to all individuals), and a finite set of (application specific) conditional operators and actions. GNP is initialized at a specific start node, and then allowed to execute up to a minimum of 5 'ticks'. Each node type has a specific tick cost (1 for conditionals and 5 for actions), where this limits how much of the graph is visited per state from the environment. In effect, a GA is used to define the connectivity between nodes and node type relative to a predefined library of conditionals or actions. Evaluation was performed in a tile world task and with a Khepera robot wall following task.

Neural Evolution of Augmented Topologies (NEAT) represents a framework for developing neural networks with arbitrary topology, beginning with a population of perceptrons (each initially fully connected to the inputs). As such, the genotype expresses a graph of connections between different types of node (input, output and neurone). Moreover, NEAT introduced a genotypic marking scheme for connections in order to establish context for crossover [32]. The same marking scheme forms the basis for a genotypic diversity measure that maintains a fixed number of niches in the population during evolution. The framework is capable of describing recurrent connections, as well as defining weight values. The NEAT framework has been widely adopted, and even benchmarked under the Atari visual reinforcement learning domain [9]. One interesting artifact of the NEAT framework is that new architectural innovations are always inserted 'under' the initial perceptron, where such innovations are always of a very sparse connectivity, whereas the initial neurone is always fully connected. Later developments, such as HyperNEAT, concentrated on expressing very large arrays of neurones in a par-

[2]Evaluation was limited to some small image data sets consisting of about 100 exemplars.

ticularly compact genotype, but resulted in all members of the population retaining a fixed number of neurones [9].

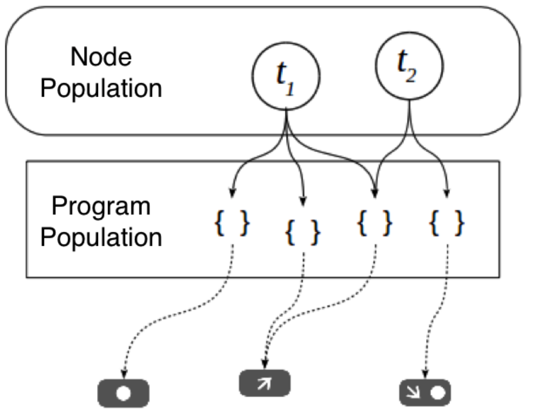
Several other frameworks for evolving neural networks (and therefore graphs) have since been proposed, including neuroevolution through Cartesian GP [36] and the use of linear genetic encoding for expressing graphs [24]. To date, such approaches have not been applied to visual reinforcement learning tasks. More recently, the DeepNEAT framework was proposed [25], in which each node of the genome represents a *layer* as opposed to a single neuron (as in NEAT). Specifically, each genome is a table of parameters used to characterize layers of the deep learning architecture and edges of the genome express connectivity between layers. Evaluation performed with image classification and language modelling tasks.

2.2 Evolution of multiple programs without graphs

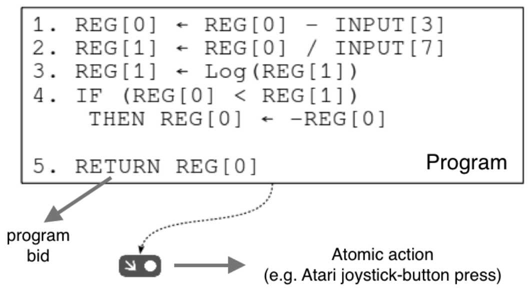
Teaming metaphors in genetic programming have previously been proposed and define cooperation in terms of an ensemble of programs that all operate simultaneously. Early work assumed a fixed length genome, thus the number of programs was always declared a priori and never varied by the evolutionary process [5]. In other cases a variable length genome was assumed, but fitness had to be specified at the the 'level' of programs ***and*** teams. This limited the base of applications that such an approach could be applied to [35, 38].

In order to avoid these issues, a symbiotic framework was previously assumed in which one population conducts a search for useful team members and a second provides the population of programs [21]. In addition, a trick from learning classifier systems was adopted in which the decision of when to act and what action to take were explicitly separated, or bid-based GP (Figure 1) [20]. Hence, given a team of bid-based GP individuals, the program for each is executed given the current state, $\vec{s}(t)$, of the task. Which ever individual from the team has the maximum output is said to have 'won' the right to suggest its action. The action is just a scalar, a, taken from the set of scalar task specific (atomic) actions, $\mathcal{A}$. Thus, for a three class classification problem $\mathcal{A} = \{0, 1, 2\}$. Each bid-based GP individual can only ever have a single action, implying that the program context evolves against a static action, potentially clarifying credit assignment. Moreover, team complement incrementally evolves in an emergent way until the relevant task decomposition is achieved, i.e. there might be multiple programs with the same action within the same team. Application examples to date include, multi-class classification [21, 18], decomposition of very large attribute spaces into very simple classification rules [18, 7], and operation under non-stationary streams [37, 16].

A second development of the above approach introduced the ability to have (bid-based GP) programs learn the context for deploying other (bid-based GP) programs [19, 6, 14]. This was particularly useful in reinforcement learning tasks in which the ultimate policy is constructed hierarchically from earlier policies (as in task transfer) [11, 29]. A limitation of the approach is that all aspects of each task need encountering at the first 'layer' of development (a form of diversity maintenance), as all later layers (of teams) express their policy 'through' some subset of earlier policies. Naturally, there are many reinforcement learning tasks for which an agent only incrementally uncovers



(a) Symbiotic relation between Nodes (Teams) and Programs



```
REG[0] ← REG[0] - INPUT[3]
REG[1] ← REG[0] / INPUT[7]
REG[1] ← Log(REG[1])
IF (REG[0] < REG[1])
  THEN REG[0] ← -REG[0]

RETURN REG[0]
```

(b) Example (bid-based GP) program

Figure 1: Symbiotic coevolutionary relation between population of nodes and programs (a). Each program expresses a bid-value and scalar action (b), as per bid-based GP. Programs assume a linear representation (b) as this enables us to support other algorithmic speedups such as intron instruction skipping [4]. Instruction set is limited to arithmetic operators, 3 non-linear operators (cosine, natural log, exponential), and a conditional (IF $R[x] < R[y]$ THEN $R[x] = -R[x]$)

properties of the task as the agent interacts with the environment, e.g. game playing agents or evolutionary robotics. Conversely, the TPG approach explicitly addresses this limitation by providing much more flexibility in how hierarchical relationships between teams of programs develop.

3 Visual Reinforcement Learning

In the following we will assume visual reinforcement learning tasks with discrete actions. For example, the Atari (arcade) Learning Environment (ALE) [3] defines state in terms of the frame buffer at time t, with $t = \{0, 1, 2, ..., n\}$ denoting a sequence of $n + 1$ consecutive frames.[3] The goal of the decision making agent is to suggest an action for each frame such that game score is maximized, i.e. reward is delayed until some end criterion is encountered, such as the maximum number of agent to game interactions $(t = t_{max})$ or an end of game state. Under the ALE the set of available (atomic) actions, $\mathcal{A}$, are defined in terms of up to 18 atomic actions[4] corresponding to the enumeration of the 8 discrete directions of the joystick, both with and without a button 'press',[5] plus a button press alone and 'no action' (NA).

[3]In practice the sequence of frames as experienced by the agent might represent a stochastic sampling of the actual true frame sequence [23].

[4]Game titles might not use all atomic actions.

[5]The action of a button press is game dependent and might make the avatar 'jump' in some games and 'fire' in others.

The interest therefore lies in finding a single machine learning framework that is capable of playing multiple game titles directly from the visual state information. No attempt is made to a priori identify appropriate input features or decompose the task into a sequence of incrementally more difficult training scenarios. The learning agent therefore has to identify an appropriate policy for playing the game directly from the same information that a human player would perceive (albeit without any sound information).

To date, such visual reinforcement learning tasks have been dominated by developments from Deep Learning (see the review in [23]). Evolutionary approaches have been proposed that required prior information to preprocess the original frame information into separate 'channels' for each object[6] [9] or limited to optimizing the parameters of a prior Deep Learning architecture [28, 33]. A GP formulation has also been proposed in which GP defines a sequence of image processing operators. The resulting GP processed image then requires interpretation through a set of heuristics in order to determine the action [10]. Such an approach is therefore limited to subsets of games for which appropriate heuristics can be designed.

Conversely, TPG was previously demonstrated to be particularly effective at the 20 ALE game titles that Deep Learning was poor at playing [12]. TPG was also demonstrated to be capable of developing single policies that played *multiple* game titles [13]. Most recently TPG was demonstrated under the VizDoom first person shooter environment [30], where this indicates that TPG may also scale to much larger state spaces than under ALE, as well as operating under much higher levels of partial observability than typically present in ALE.

4 Tangled Program Graphs

TPG represents a framework for organizing *multiple* programs into structures such that they solve some larger task in a highly modular way. Our starting point is a symbiotic evolutionary framework (Figure 1(a)) consisting of: 1) a Node population that defines nodes in the TPG graph and 2) a Program population. Nodes define which programs will cooperate in order to make a decision at that particular node. The program population defines each individual in terms of program, p, and atomic action $a \in \mathcal{A}$ (Figure 1(b)). The two populations coevolve under a symbiotic relationship in which the Node population conducts a search for 'good' modules (teams of programs) and the Program population concentrates on sourcing 'useful' programs.

In the initial population all nodes consist of between 2 and ω (bid-based) programs, initialized randomly, with action, a, assigned with uniform probability from the set of atomic actions, $\mathcal{A}$, under the constraint that:

- there must be at least two different actions present in the set of programs associated with the same node.
- each node must have a unique complement of programs.

[6]Implying that prior knowledge is necessary regarding the number of objects appearing each game title.

Such a starting point implies that: 1) all graphs initially only possess a single node (Figure 2(a)), and 2) the same program can appear in multiple Nodes (Figure 1(a)).

After fitness evaluation the agents are ranked (Step 2d, Algorithm 1) and the worst performing *Gap* deleted (Step 2e). Variation operators will sample *Gap* parents from the surviving population content in order to produce offspring (Section 4.2). There is no special significance to the adoption of a breeder model, other than it is elitist.

At this point it is worth noting that:

- Although a two population framework is assumed, fitness is only explicitly defined for nodes in the Node population.[7]

- Members of the program population are tested after the *Gap* worst performing agents are deleted. If any program is not associated with a surviving individual from the Node population, it is deleted (Step 2f). This implies that the size of the Program population actually fluctuates as a function of the selection and variation operators.

- Nodes in the Node population might be root nodes or nodes that are internal to a graph. As will become apparent, the number of agents is equal to the number of root nodes. It is therefore generally the case that the number of agents is less than the size of the Node population, the precise composition also being an emergent property. With this in mind, a test is introduced to ensure that a minimal number of agents, R_{size}, is maintained at each generation (Step 2j).

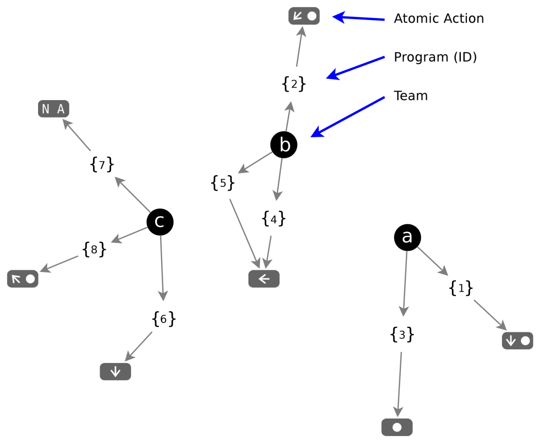
Before introducing further details of the TPG algorithm Section 4.1 provides some intuition as to how the symbiotic representation develops solutions into a tangled graph from teams of programs (each node is a unique team). Section 4.2 will then define the variation operators and relate their function to the overall TPG algorithm (Algorithm 1). Finally, Section 4.3 provides a walk through for how a TPG agent is evaluated, given a frame buffer input.

4.1 Developmental cycle

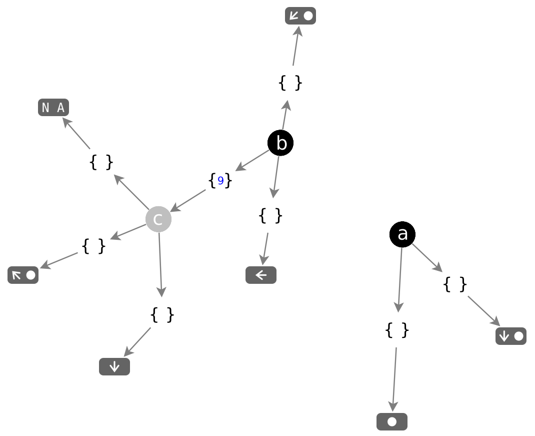
Figure 1(a) established that at initialization each node (of the Node population) identifies a unique subset of programs from the program population. We can make the connection to graphs more obvious by ignoring the relationship to the two populations (a genotypic property) and instead just concentrate on the representation of agents (initially each node is an agent), a phenotypic property.

Figure 2(a) illustrates the case of an initial population consisting of three agents a, b, c, each of which only consist of a single node. Agents b and c consist of three programs and agent a only 2. Taking agent a as an example, it comprises of two programs, 1 and 3. Arc direction always follows from the node at which programs have team membership and ends at the corresponding action. Given that this is the initial population, actions can only ever be atomic actions, Figure 2(a), and all members of the Node population are

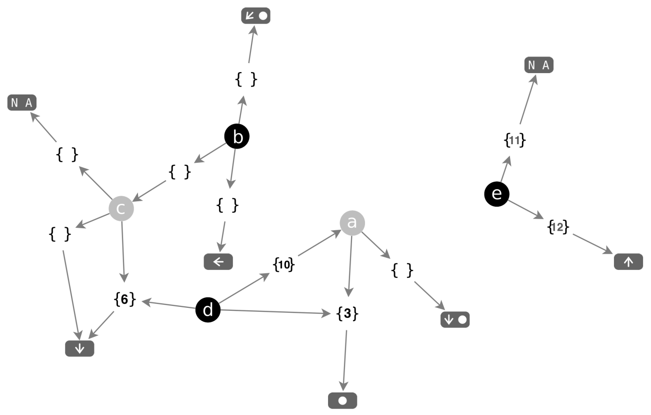
[7]Actually as nodes are subsumed into graphs, it will be come apparent that only a subset of nodes require explicit fitness evaluation (Section 4.1).



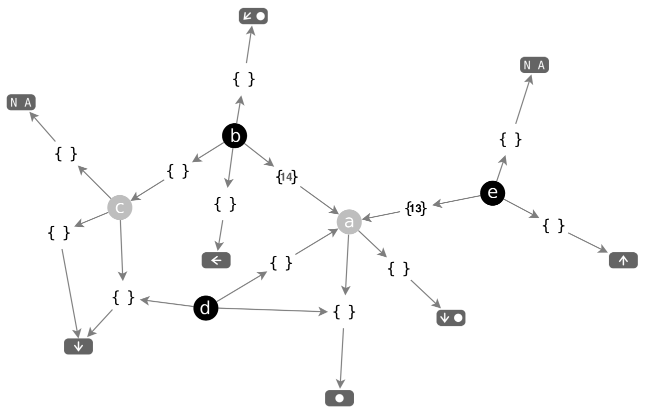
(a) Initial (single node) agents



(b) Transition from a single node to a bi-node agent



(c) Two bi-node agents emerge



(d) An agent with 4 nodes emerges

Figure 2: Snapshot of emergent process of TPG graph construction using a hypothetical illustrative example. Black circles represent root nodes, and each root node constitutes the starting node for evaluating an agent. Grey nodes represent members of the Node population that have became subsumed into another agent through the action of the program variation operator pair (p_{act}, p_{typ}).

therefore root nodes. Each root node represents the node at which evaluation/execution begins (Section 4.3), thus each agent may only have a single root node.

Figure 2(b) illustrates the effect of a modification to the action of program '5'. Action modifications can either result in a different atomic action being selected ($a \in \mathcal{A}$), or the program pointing to another node ($a \in N(g)$). As program '5' was modified, its identifier changes (to '9') to reflect the fact that the original program '5' might still exist elsewhere, i.e. as used by a different node (hence the use of cloning before the application of variation operators, Section 4.2). Node 'c' now no longer represents a root node, as it is a child of node 'b'. Hence the number of agents has actually decreased.

In Figure 2(c) node 'a' from Figure 2(b) was sampled twice as the parent and cloned in both cases resulting in nodes 'd' and 'e'. Node 'e' inherited the programs of node 'a', but had its atomic actions changed. Conversely, Node 'd' retained one program unchanged ('3') but also gained program '6' and a new program '10'. As the action of program '10' is a pointer to node 'a', node 'a' is no longer a root node. At this point we have three agents: $\langle b : c \rangle$, $\langle d : a \rangle$, $\langle e : \emptyset \rangle$ where $\langle x : y, z \rangle$ denotes agent 'x' with root node 'x' and non-root nodes 'y' and 'z'.

Figure 2(d) illustrates the point at which an agent with three nodes emerges. In this case Node 'b' gained a 4th program ('14') with action identifying Node 'a'. In addition, Node 'e' also had a program added ('13') which also in this case happened to use Node 'a' as its action. We now have one three node agent, $\langle b : a, c \rangle$, and two agents with 2 nodes, $\langle d : a \rangle$, $\langle e : a \rangle$. Note, that for clarity we have focused the above commentary on the offspring agents introduced as a function of selection and variation. The parents would be retained and compete with the offspring for the right to survive.

4.2 Variation

Following the identification of parents (Step 2g, Algorithm 1), each parent is cloned. More specifically, only the *root node* of each parent is subject to cloning and variation (at initialization all nodes are root nodes). This implies that competition between the simpler parent and generally more complex offspring is enforced. Thus, in order to survive the more complex offspring have to perform better than the (simpler) parent.

The first set of variation operators operate on the cloned (root) nodes (Step 2i), and take the form:

- add or delete a reference to a program (p_a, p_d) or,
- modify a program currently in the node (p_m).[8]

The second set of variation operators operate on programs associated with a root node offspring (only called upon when p_m test true, Step 2(i)iii), in which case the effected program is first cloned before applying variation operators to the cloned program:

- delete or add an instruction (p_{del}, p_{add}),

[8]The variation operators are actually applied multiplicatively, possibly resulting in any single operator being applied several times, see [18].

- mutate an instruction (p_{mut}),
- swap two instructions within the same program (p_{swp}), or
- modify the action of a program within the team (p_{act}).

Should p_{act} test true, then an additional test is applied, p_{typ}, which establishes the *type* of action change (Step 2(i)iiiD). Thus, for p_{typ} false, the cloned program's action, a_i is selected from the set of atomic actions $a_i \in \mathcal{A}$, whereas for p_{typ} true, the new action is a pointer to any node in the Node population of generation g, or $a_i \in N(g)$.

One constraint is enforced during action variation. When the program is inserted in its corresponding node, there must be at least one atomic action present across the subset of programs local to that node. This property will later be employed to guarantee that infinite loops do not result during evaluation, i.e. all states will always result in an atomic action (Section 4.3).

Finally, given the high cost of fitness evaluation we assess the uniqueness of programs subject to program, as opposed to action, modification (Step 2(i)iv). To do so, a collection of the 50 last state observations as executed under fitness evaluation are retained. Each new program is then tested to determine whether the bid values for the new program are at least τ different from any of the other programs in $Prog(g)$. If not, the variation operators for program modification are reapplied. In effect this represents a test on the neutrality of the variation operators using a minimum threshold of behaviour.

4.3 Agent Evaluation

In the following we will assume that state information, $\vec{s}(t)$, corresponds to the current content of the frame buffer.[9] Evaluation of the agent always commences from the root node. Thereafter, the path through the agent's graph is dynamic, generally resulting in only a fraction of the agent's program graph being evaluated before an action is identified. This makes TPG exceptionally efficient to evaluate as compared to current Deep Learning or neuro-evolutionary approaches (both of which evaluate the contribution from all of the topology for every decision).

Evaluation of a node is a two step process in which only the subset of programs associated with the node are executed. Out of this subset, only the program with the largest bid 'wins' the right to suggest its corresponding action (providing it is not 'marked', see below). Thus, evaluating a node identifies a specific arc. There are then two scenarios, either the arc references an atomic action or it references another node in the graph. We process these cases as follows:

Action is atomic this represents the decision of this agent. The state of the world would then be updated and the process of evaluation repeated.

[9]In practice a down sampling step is often employed in order to keep the computational cost plausible. In the case of the Atari ALE environment the down sampling was by a factor of five [12], whereas no down sampling was employed in the case of VizDoom [30].

Algorithm 1 TPG algorithm. The term ‘agent’ denotes the subset of nodes in the Node population that represent root nodes (Section 4.1). Although an agent typically consists of multiple nodes, only the root node is subject to variation (Section 4.2).

```
• Initialize $Node(0)$
• Initialize $Prog(0)$
• For $(g = 0; !\text{EndOfEvolution}; g = g + 1)$                          ▷ Generation loop
    1. $AgentList = \emptyset$
    2. For all (node $\in Node(g)$) AND (node = root)                  ▷ Identify valid agent
        (a) agent = node;
        (b) update($AgentList$, node)
        (c) For all (evaluations)                                  ▷ Evaluation loop (§4.3)
            i. Evaluate(agent)
            ii. update(agent.Fitness)
        (d) Rank(agents $\in AgentList$)                                   ▷ Select parents
        (e) Prune worst ranked $Gap$ agents from $AgentList$ and corresponding nodes
            from $N(g)$
        (f) Prune all $prog \in Prog(g)$ without a node
        (g) Select (Parents $\in AgentList$)
        (h) Clone ($NewAgents$, Parents)
        (i) For all (agent $\in NewAgents$)                          ▷ Create offspring (§4.2)
            i. newRoot = DeleteProgFromNode(root $\in$ agent, $p_d$)   ▷ Node variation
            ii. newRoot = AddProgToNode(root $\in$ agent, $p_a$)
            iii. IF (ModifyProgram($p_m$) THEN                          ▷ Program variation
                A. Select(prog $\in$ newRoot)
                B. Clone (newProg, prog)
                C. ModInstr(newProg, $p_{del}, p_{add}, p_{mut}, p_{swp}$)
                D. ModAction(newProg, $p_{act}, p_{typ}$)                 ▷ Action variation
            iv. IF (!unique(newProg)) THEN repeat ‘ModifyProgram’      ▷ Neutrality
                test
            v. $Prog(g) = Prog(g) \cup$ newProg                  ▷ Update Program population
            vi. $Node(g) = Node(g) \cup$ newRoot                    ▷ Update Node population
        (j) IF $|AgentList| < R_{size}$ THEN goto Step 2i                ▷ Case of too few agents
```

Action is non-atomic the arc is marked and the node pointed to is subject to evaluation, as above. The environmental state, $\vec{s}(t)$, is unchanged.

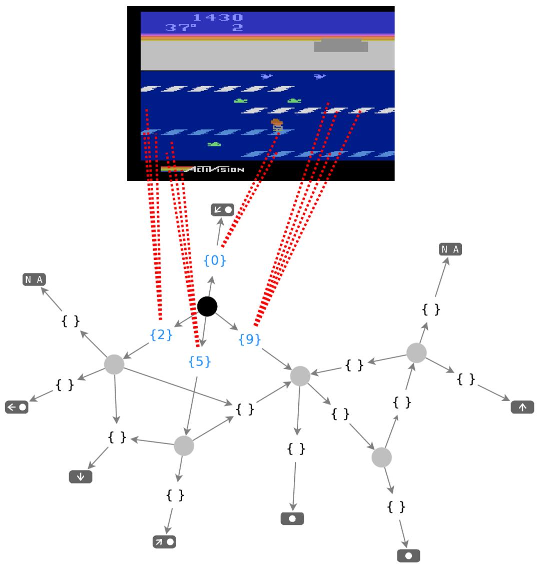
In the special case of a *marked* arc 'winning' a Node evaluation, this implies that the node in question has been previously visited and a loop detected. The loop is broken by removing this arc from the set of candidate programs at this node evaluation and then returning the arc with the winning bid. If this is also marked, the process of removing the marked arc and selecting the next available winning bid at this node repeats. Because every node must have at least one atomic action, there is always a way to break out of a loop, i.e. each time the same node is visited, it has to 'exit' using a different arc.

Figure 3 illustrates the process of evaluation for a hypothetical TPG agent and state. Subplot 3(a) represents the execution of programs associated with the agent's root node (programs 0, 2, 5, 9). Note that each program is free to index any part of the state space (frame), adding to the ability to decompose the task. Program 9 had the 'winning' bid and identifies the next node for evaluation (this arc is also marked), Subplot 3(b). This node only consists of two (outgoing) arcs, corresponding to programs 3 and 7. Execution of these two programs potentially implies that completely different state information is utilized, Subplot 3(c). The winning program is identified as program 7, with an atomic action, so evaluation is complete, Subplot 3(d). This action would result in the game state changing in response to the avatar assuming this action, thus game state is advanced one step, $\vec{s}(t) \rightarrow \vec{s}(t+1)$. Naturally, if $\vec{s}(t+1)$ corresponds to an end-of-game condition this would complete the fitness evaluation for the agent. Otherwise all marked arcs are reset and agent evaluation again commences from the root node.

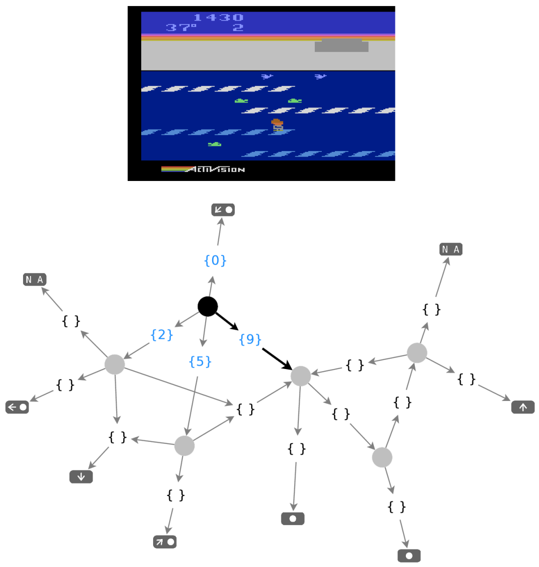
5 Case study: Atari Learning Environment

Section 3 provided background to the Atari Learning Environment (ALE) [3, 23], which represents one of the most widely employed benchmarks for visual reinforcement learning. The basic goal is to return an agent capable of playing different gaming titles from the ALE library under a common parameterization. The only information provided to the agent is the visual information from the frame buffer, and the subset of joystick actions specific to the title in question. We also note that the ALE provides various sources of uncertainty including: random sampling of the frames provided to the agent, inter frame variation in sprites described, and depending on the game title, partial observability, i.e. the first person perspective (see [23] for a discussion of these properties).

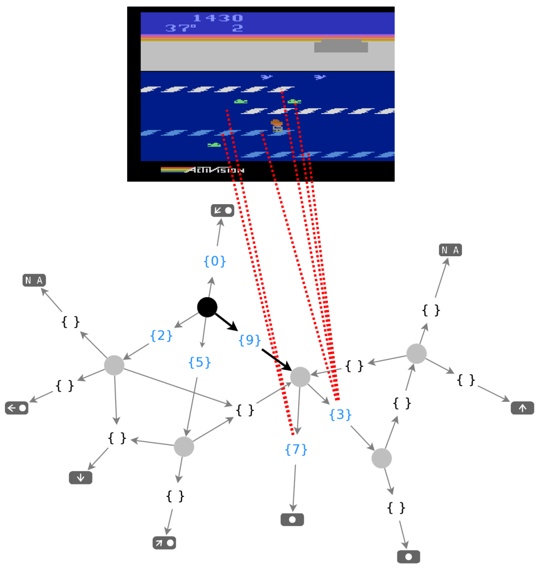
The approach taken by deep learning to reduce the impact of the stochastic components has been to adopt 'frame stacking' [27], thus the content of multiple frames is incorporated into the input to the deep learning agent in order provide visual input that removes the stochastic aspects of the task. For the basis of this comparison, we will assume 12 titles common to recent evaluations of evolutionary computation approaches to visual reinforcement learning under ALE. Specifically Salimans et al. describe an approach based on Evolutionary Strategies for identifying weights of an a priori specified deep learning architecture [28]. Such et al., use a genetic algorithm to provide a very compact description of a deep learning architecture consisting of 4 million weights [33].



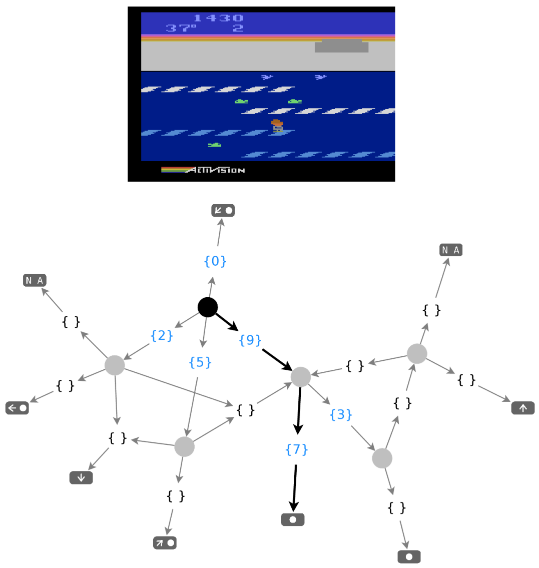
(a) Execution begins at the root node



(b) Winning program identified from root node



(c) Execution passes to node, evaluate programs at node



(d) Winning program associated with an atomic action

Figure 3: Example of a TPG agent during evaluation. Execution always commences relative to the agent's root node (Subplot (a)). Winning program identifies next hop through the TPG individual (Subplot (b)). Process of node evaluation repeats (Subplot (c)), until an atomic action is identified (Subplot (d)).

In both cases, the authors emphasize that the evolutionary computation approach is faster to train (with extensive GPU support) than the original deep learning architecture, DQN [27], while providing competitive agent policies.

Initial results for TPG in the ALE employed a frame preprocessing procedure in which each pixel was limited to 8 potential colour values (i.e. the SECAM colour encoding, [3]) and each frame was quantized by a factor of 5, resulting in an input space of 1,344 decimal state variables in the range 0-255 [12, 13]. In the results reported here, *no attempt* was made to reduce the dimensionality of the initial state information. This implies that the screen as perceived by TPG agents includes all $210 \times 160 = 33,600$ pixels with 128 possible colour values for each pixel (i.e. the NTSC colour encoding, [3]). The instruction set adopted is unchanged from earlier research with team GP applied to classification tasks [18], i.e. no application/image specific operators are employed. Figure 1(b) illustrates a typical program employing this instruction set. No attempt is made to filter out stochastic properties in the sequence of frames provided by ALE.

Table 1 summarizes the average game score for each agent on the 12 ALE titles (averaged over 200 evaluations with the no-op game initialization [27]). It is clear that specific algorithms do particularly well on different subsets of games. Also apparent is that DQN has both the most number of games at rank 1 (best) and most number of games at rank 4 (worst). Conversely, TPG appeared to be the most consistent across this set of titles, with only one worst ranked title (Atlantis) and the best average rank across all titles. In short, TPG is competitive in terms of the quality of the agent policies identified. Moreover, TPG is providing these results while operating at the original ALE resolution and no frame staking. The GA employs the approach to down-sampling/ cropping/ frame skipping adopted by DQN which reduces the original representation to 84×84 pixels and reduces the partial observability of the task (no information is given regarding the set up for the ES).

Table 2 summarizes the complexity of TPG solutions, where for comparison the GA individuals describe a deep learning architecture with 4 million weights [33]. Moreover, the number of computations actually performed *per decision* is significantly more than this on account of the convolution operation present in deep learning architectures. It is readily apparent that the worst case cost of decision making in TPG ($\approx$1,200 instructions) is at least 3 orders of magnitude less than that experienced in deep learning (and can be up to 5 orders of magnitude less). In short, TPG can evolve solutions to visual reinforcement learning tasks *without* specialized hardware support such as GPUs because it explicitly breaks a task down through a coevolutionary process of divide and conquer.

In general, TPG policies developed an exceedingly sparse coverage of the available frame pixels, typically less that 5%, while each decision required less than 2% of the frame pixels, Table 2. This is a reflection of the fact that Atari video games (and visual information in general) have a lot of redundant information. In particular, a large portion of the screen content is designed for entertainment value rather that being important for decision-making. Furthermore, while important game entities are most often larger than a single pixel, the agent may only need to percieve a very small number of pixels in order to detect and respond to such entities. The capacity to automatically scale to these properties of the environment constributes to the overall efficiency of the resulting policies.

Table 1: Game scores for 12 Atari titles under no-op game initialization. Integer in parenthesis denotes the rank of the algorithm under each game title. Last row details Average rank of each algorithm across the 12 titles.

Game Title	TPG	GA [33]	ES [28]	DQN [33]
Amidar	365 (3)	377 (2)	112 (4)	978 (1)
Assault	2,027 (2)	814 (4)	1,674 (3)	4,280 (1)
Asterix	2,967 (2)	2,255 (3)	1,440 (4)	4,359 (1)
Asteroids	2,575 (2)	2,700 (1)	1,562 (3)	1,365 (4)
Atlantis	110,247 (4)	127,167 (3)	1,267,410 (1)	279,987 (2)
Enduro	108 (2)	80 (4)	95 (3)	727 (1)
Frostbite	6,059 (2)	6,220 (1)	370 (4)	797 (3)
Gravitar	1,068 (1)	764 (3)	805 (2)	473 (4)
Kangaroo	14,026 (1)	11,254 (2)	11,200 (3)	7,259 (4)
Sequest	1,083 (3)	850 (4)	1,390 (2)	5,861 (1)
Venture	740 (3)	1,422 (1)	760 (2)	163 (4)
Zaxxon	6,523 (2)	7,864 (1)	6,380 (3)	5,363 (4)
Avg. rank	2.25	2.417	2.833	2.5

Table 2: Cost of decision making in TPG agent post training. #Nodes is the total number of nodes in the agent; Av. Nodes is the average number of nodes actually evaluated in order to make a decision; Av. #Instr. is the average number of instructions executed per decision; %Pixels is the percent pixels indexed by the entire TPG graph; Av.%Pixels is the average percent of pixels indexed per decision.

Game Title	#Nodes	Av. Nodes	Av. #Instr.	%Pixels	Av.%Pixels
Amidar	65	4	680	6.36	0.9
Assault	53	4	768	4.03	0.88
Asterix	24	2	643	2.73	0.75
Asteroids	51	4	1259	4.86	1.38
Atlantis	53	6	1201	3.84	1.36
Enduro	7	2	170	0.53	0.24
Frostbite	63	4	1063	4.22	1.08
Gravitar	48	4	1150	4.93	1.21
Kangaroo	109	7	1213	9.43	1.38
Seaquest	70	4	980	4.98	1.03
Venture	20	2	294	2.18	0.38
Zaxxon	35	3	414	2.23	0.48

6 Case study: VizDoom

In the following we provide a further illustration of some of the possible outcomes and process of decision making when using TPG, in this case under the first person shooter environment of VizDoom [15]. VizDoom represents an environment with three dimensional state information and (care of the first person perspective) a lot of partial observability. Smith and Heywood demonstrated TPG agents operating under the raw VizDoom frame resolution of $320 \times 240 = 76,800$ [30].

Figure 4(a) illustrates a typical view that a player might encounter under VizDoom when playing in single agent mode. That is to say, the goal is to navigate a cavern like environment, while annihilating a range of predefined opponents, before your own health reaches zero. Statistics characterizing the agent specific information is present at the bottom of the screen (e.g., available ammunition, current health), whereas current game state is captured in the remaining frame real-estate. Information that a decision making agent can employ must come from the frame buffer. There are no special inputs that express/summarize agent health or armour.

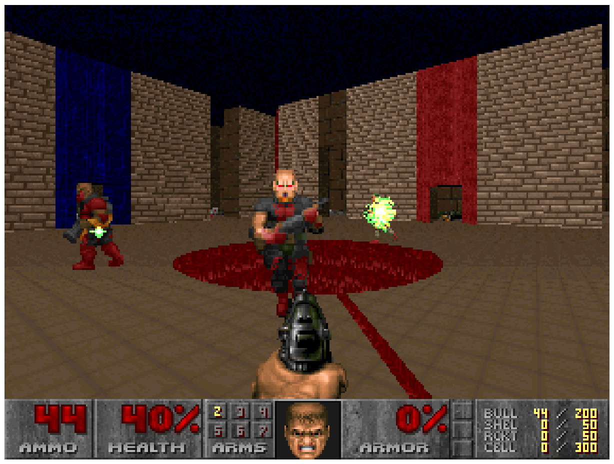
Figure 4(b) illustrates a TPG individual post training, and the nodes visited in order to make a decision on the specific game state of Figure 4(a).[10] We can observe what state information is actually indexed in order to make each decision at each TPG node, Figure 4. Note that the grey–red bars on the top and LHS summarize a projection of the pixels indexed by TPG programs from the x and y-axis respectively. The red (grey) regions represent the locations in which most (least) indexing by programs occurs.

Figure 5(a) characterizes the pixels indexed by the 7 programs at the root node. A broad sweep of pixels appear to be indexed across the entire visual field, with some what higher density present at the centre. The arc from Root to Node 1 represented the winning program, thus evaluation of the 5 programs at Node 1 resulted in the distribution of pixel indexing summarized by 5(b). What is interesting with this distribution is that the greatest density of pixel indexes falls in the information bar across the bottom of the frame. This seems to imply that programs at this node tend to use avatar statistics to resolve what arc to prioritize at Node 1.

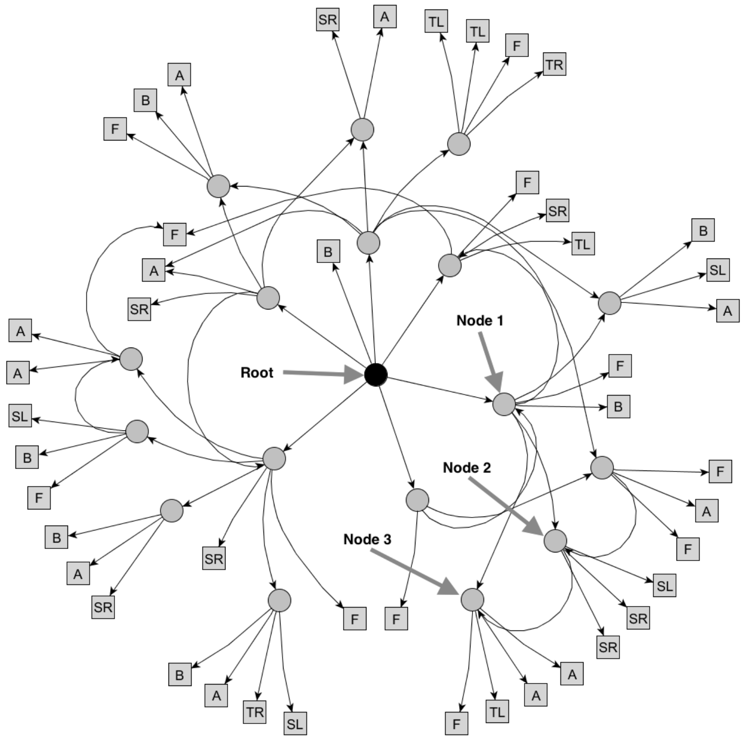
Node 2 was selected next, resulting in the execution of a further 4 programs to determine the 'winning' arc (Figure 5(c)). It now appears that pixels lying in the bottom third of the screen, from the centre and to the right are preferred. This seems to coincide with the weapon that the avatar is holding (along with additional statistics), possibly implying that the agent is checking what type of weapon is being held. The final node referenced before recommending an atomic action (Node 3) appears to index pixels from across the middle of the screen (Figure 5(d)) before choosing the 'A = Attack/ShoCumulot' atomic action as its recommendation.

In order to illustrate how much information in total is being used for this particular decision, Figure 6 provides an equivalent cumulative view of the pixels indexed as each node is visited. The increased indexing at the state bar across the bottom of the screen is apparent between Root node and Node 1. Likewise, more emphasis of the avatar's

[10]This TPG individual actually represents a policy able to operate under 10 different VizDoom tasks [29].

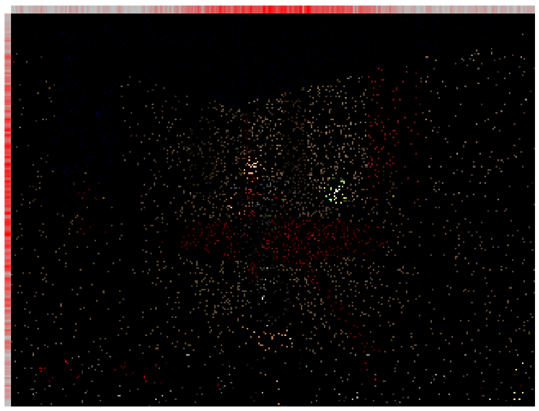


(a) Source frame from Defend the Circle task

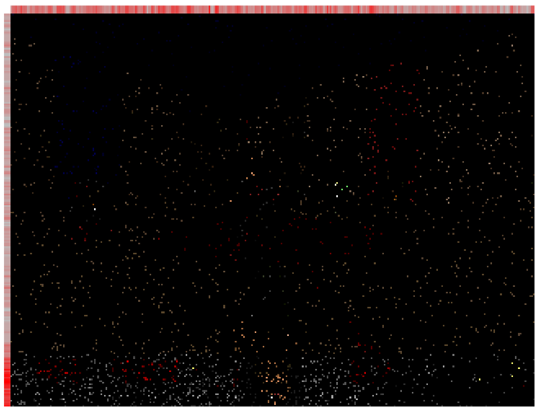


(b) TPG Agent

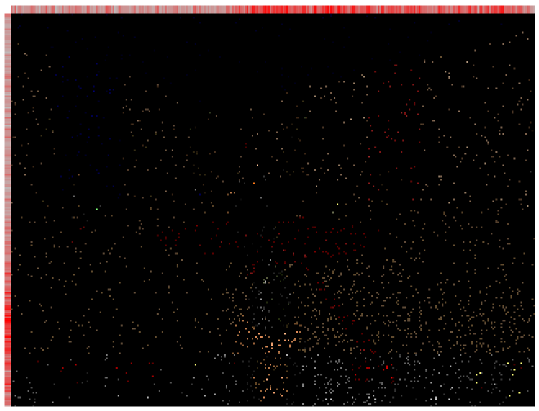
Figure 4: Example source frame and TPG agent. Sequence of node evaluations for this frame content: Root → Node 1 → Node 2 → Node 3. A total of 7 atomic actions exist: F/ B - move forward/ backward; TL/ TR - turn left/ right; SL/ SR - Strafe Left/ Right, A - Attack (shoot).



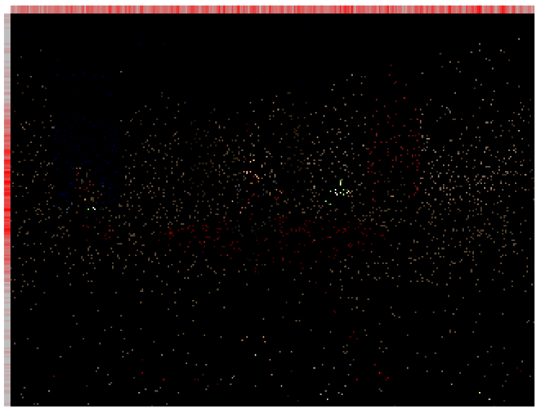
(a) Root node



(b) Node 1

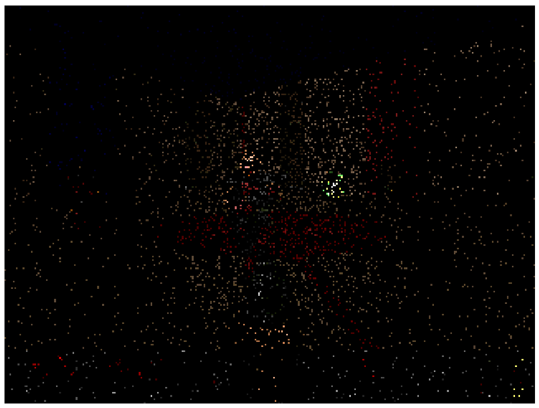


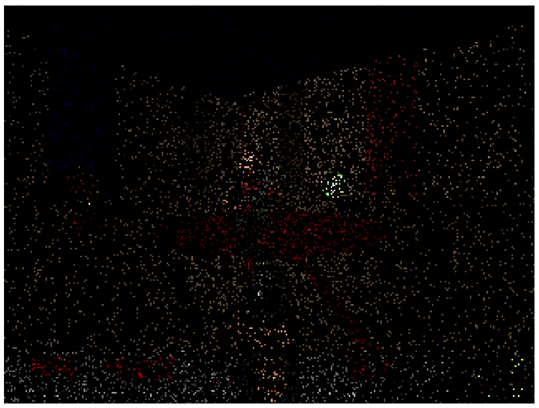
(c) Node 2

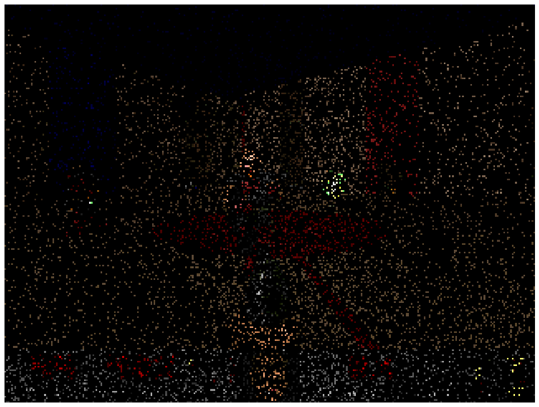


(d) Node 3

Figure 5: Case of decision making for a specific frame of VizDoom. Node specific view. Outer red-grey bar on top and LHS summarize the distribution of pixels indexed as projected on to the x and y-axis respectively (Figure best viewed in colour)


(a) Root node (cumulative)


(b) Node 1 (cumulative)


(c) Node 2 (cumulative)

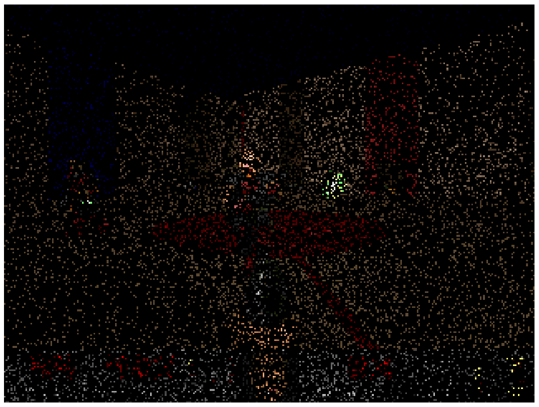

(d) Node 3 (cumulative)

Figure 6: Case of decision making for a specific frame of VizDoom. Cumulative view. (Figure best viewed in colour)

weapon occurs at Node 2, whereas little in the way of 'new' pixel indexing occurs at the last step.

7 Discussion

Few machine learning paradigms are able to auto-construct modular topologies at multiple levels of abstraction. Genetic programming represents a variable length representation, but is rarely deployed to evolve anything other than monolithic solutions, i.e. single programs in which all the instructions are executed.[11] Conversely, teaming metaphors organize programs into a single 'group', but might require (task specific) fitness assignment at the level of individual (program) and team or face issues regarding diversity maintenance.

TPG always adds innovations 'top down', i.e. new topologies are identified by an offspring pointing to something else in the population. A competition between simpler parents and more complex children is the norm. Evaluation is always initiated from the root node and limited to the programs explicitly associated with that node. There can only ever be a single 'winning' arc at each node evaluation, and loop detection resolves such that a different path must be taken. All of these properties help ensure that graph structure is emergent, execution efficient, and task decomposition explicit.

Naturally, this represents a snapshot of very early developments in the TPG framework and consequently a lot of unknowns exist. For example, can refinements also be introduced 'bottom up' during development as well as 'top down', or how might memory be introduced into geno/phenotypes that emerge dynamically? It is certainly clear that the deep learning methodology (in concentrating on finding appropriate encodings) scales to a wide range of tasks. Conversely, TPG learns to develop very compact decompositions of a task relative to the original high-dimensional state space. It is not clear what task preferences and/or strengths/ weaknesses might result as a function of this rather different approach to model building.

Acknowledgments. Stephen Kelly gratefully acknowledges support from the Nova Scotia Graduate Scholarship program. Malcolm Heywood gratefully acknowledges support from the NSERC Discovery and CRD programs (Canada).

References

[1] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs by graph programming. In *European Conference on Genetic Programming*, volume 10781 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2018.

[11] Part of this might be due to the types of tasks that researchers choose to deploy GP on. For example, 'expressive GP' demonstrates its more interesting properties under tasks such as software synthesis [31].

[2] Wolfgang Banzhaf. Artificial regulatory networks and genetic programming. In R. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*, pages 43–62. Springer, 2003.

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[4] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 1st edition, 2007.

[5] Markus Brameier and Wolfgang Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.

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

[7] John A. Doucette, Andrew R. McIntyre, Peter Lichodzijewski, and Malcolm I. Heywood. Symbiotic coevolutionary genetic programming: a benchmarking study under large attribute spaces. *Genetic Programming and Evolvable Machines*, 13(1):71–101, 2012.

[8] L.J. Fogal, A.J. Owens, and M.J. Walsh. Artificial intelligence through a simulation of evolution. In *Proceedings of the Cybernetic Sciences Symposium*, pages 131–155, 1965.

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

[10] Baozhu Jia and Marc Ebner. Evolving game state features from raw pixels. In *European Conference on Genetic Programming*, volume 10196 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2017.

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

[12] Stephen Kelly and Malcolm I. Heywood. Emergent tangled graph representations for atari game playing agents. In *European Conference on Genetic Programming*, volume 10196 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2017.

[13] Stephen Kelly and Malcolm I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *ACM Genetic and Evolutionary Computation Conference*, pages 195–202, 2017.

[14] Stephen Kelly, Peter Lichodzijewski, and Malcolm I. Heywood. On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[15] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2016.

[16] Sara Khanchi, Ali Vahdat, Malcolm I. Heywood, and A. Nur Zincir-Heywood. On botnet detection with genetic programming under streaming data label budgets and class imbalance. *Swarm and Evolutionary Computation*, 39:123–140, 2018.

[17] Xianneng Li, Shingo Mabu, and Kotaro Hirasawa. A novel graph-based estimation of the distribution algorithm and its extension using reinforcement learning. *IEEE Transactions on Evolutionary Computation*, 18(1):98–113, 2014.

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

[19] P. Lichodzijewski and M. I. Heywood. The Rubik's Cube and GP temporal sequence learning. In R. Riolo, T. McConaghy, and E. Vladislavleva, editors, *Genetic Programming Theory and Practice VIII*, pages 35–54. Springer, 2011.

[20] Peter Lichodzijewski and Malcolm I. Heywood. Coevolutionary bid-based genetic programming for problem decomposition in classification. *Genetic Programming and Evolvable Machines*, 9(4):331–365, 2008.

[21] Peter Lichodzijewski and Malcolm I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *ACM Genetic and Evolutionary Computation Conference*, pages 363–370, 2008.

[22] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A graph-based evolutionary algorithm: Genetic network programming and its extension using reinforcement learning. *Evolutionary Computation*, 15(3):369–398, 2007.

[23] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[24] J. H. Metzen, M. Edgington, Y. Kassahun, and F. Kirchner. Analysis of an evolutionary reinforcement learning method in multiagent domain. In *ACM International Conference on Autonomous Agents and Multiagent Systems*, pages 291–298, 2008.

[25] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[26] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2000.

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

[28] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017.

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

[30] Robert J. Smith and Malcolm I. Heywood. Scaling tangled program graphs to visual reinforcement learning in vizdoom. In *European Conference on Genetic Programming*, volume 10781 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2018.

[31] Lee Spector and Nicholas Freitag McPhee. Expressive genetic programming: concepts and applications. In *ACM Genetic and Evolutionary Computation Conference (Tutorial)*, 2016.

[32] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 2002.

[33] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2018.

[34] A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic visual learning*. Oxford University Press, 1996.

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

[36] Andrew James Turner and Julian Francis Miller. Neuroevolution: Evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, 7(3):135–154, 2014.

[37] Ali Vahdat, Jillian Morgan, Andrew R. McIntyre, Malcolm I. Heywood, and A. Nur Zincir-Heywood. Evolving GP classifiers for streaming data tasks with concept change and label budgets: A benchmarking study. In Amir Hossein Gandomi, Amir Hossein Alavi, and Conor Ryan, editors, *Handbook of Genetic Programming Applications*, pages 451–480. Springer, 2015.

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