Scaling Tangled Program Graphs to Visual Reinforcement Learning in ViZDoom

Robert J. Smith[1] and Malcolm I. Heywood[1]

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

Article originally appears at EuroGP'18 (LNCS 10781) under Springer copyright 2018
`https://link.springer.com/chapter/10.1007%2F978-3-319-77553-1_9`

Abstract

A tangled program graph framework (TPG) was recently proposed as an emergent process for decomposing tasks and simultaneously composing solutions by organizing code into graphs of teams of programs. The initial evaluation assessed the ability of TPG to discover agents capable of playing Atari game titles under the Arcade Learning Environment. This is an example of 'visual' reinforcement learning, i.e. agents are evolved directly from the frame buffer without recourse to hand designed features. TPG was able to evolve solutions competitive with state-of-the-art deep reinforcement learning solutions, but at a fraction of the complexity. One simplification assumed was that the visual input could be down sampled from a 210×160 resolution to 42×32. In this work, we consider the challenging 3D first person shooter environment of ViZDoom and require that agents be evolved at the original visual resolution of 320×240 pixels. In addition, we address issues for developing agents capable of operating in multi-task ViZDoom environments *simultaneously*. The resulting TPG solutions retain all the emergent properties of the original work as well as the computational efficiency. Moreover, solutions appear to generalize across multiple task scenarios, whereas equivalent solutions from deep reinforcement learning have focused on single task scenarios alone.

1 Introduction

State-of-the-art reinforcement learning (RL) algorithms increasingly emphasize the encoding of important properties from visual inputs, typically screen capture [17]. Several benchmarks have appeared including (but not limited to) arcade gaming titles [2], racing games [16] and first person shooter (FPS) environments [10]. Arcade formulations tend to emphasize the ability to play tens of titles using the same learning algorithm, but assume a relatively low resolution input and frequently supply complete information of game state. Conversely, the 'first person' perspective of FPS games provide a very rich environment – multiple rooms, opponents, objects for capture/ avoidance – in which to investigate learning algorithms under partial observability of state, much higher dimensionality and a corresponding increase in the variation of content.

In this work we are interested in assessing the utility of a recently proposed Tangled Program Graph (TPG) approach [8, 9] to the ViZDoom FPS environment. TPG was previously demonstrated to be capable of discovering emergent solutions to reinforcement learning problems based on visual input alone which were orders of magnitude simpler than solutions from deep learning [8, 9]. Our motivation in this work is to answer to what degree the previous TPG results were dependent on a relatively low resolution state capture[1] and assess the capacity to perform task transfer between different typically partially observable task scenarios described in the ViZDoom environment. This is of merit as: 1) visual task domains are increasingly assuming very high resolution inputs that will potentially result in a loss of objects when down sampled too much, and 2) transferring skills between different source training scenarios can potentially lead to stronger performance in a target scenario, such as ViZDoom 'deathmatches' (e.g. [13, 21]).

In this work we show that, by incorporating some additional task agnostic learning heuristics, TPG is able to simultaneously provide solutions to a total of ten custom task scenarios, eight of which come with the ViZDoom distribution and two are specifically introduced in this work (Section 4). We believe that this demonstrates that TPG is capable of scaling to much larger state spaces, while still retaining the capacity to discover solutions that are very efficient to execute without special hardware support.

[1]1344 pixels in the down sampled visual interface of [8, 9] versus 76800 pixels in the TPG deployment demonstrated for ViZDoom.

2 Related Research

Gaming environments are increasingly being employed as suitably difficult tasks for demonstrating artificial general intelligence [22]. In this work, we are particularly interested in scenarios that focus on the ability of the machine learning algorithm alone, as opposed to hybrid formulations that augment machine learning with search, e.g. deep learning with Monte Carlo Tree Search [4]. From this perspective, the result of Mnih et al. was particularly noteworthy in that it demonstrated an effective strategy for playing a cross section of games using state information defined by screen capture alone [17], i.e. without recourse to any features crafted by humans. Moreover, the resulting performance in some game titles was better than/competitive with that of a human player. Since then, multiple deep reinforcement learning frameworks have been proposed and benchmarked on multiple console game titles (e.g. [11, 12]), as have neuro-evolutionary approaches [5, 3] and schemes based on some form of GP [8, 9, 6].

More recently, researchers have began to focus on gaming tasks outside of arcade console environments, where the case of first person shooter (FPS) environments is of particular relevance to this work. Specifically, FPS environments introduce three dimensional state information and partial observability. This means that strategies need discovering for navigating multiple rooms while avoiding specific objects/opponents and engaging with other types of opponents/objects. In short, FPS environments represent a different set of challenges from that experienced in console style games.

The ViZDoom framework was developed in 2016 in order to explicitly facilitate the efficient evaluation of 3D vision based machine learning algorithms on a FPS environment [10]. Machine learning agents therefore view the game world through the frame buffer and learn how to play on the basis of visual stimuli alone. ViZDoom gives researchers control over the size of the screen buffer, access to the depth buffer, as well as the ability to customize the learning process using WAD ("Where's All the Data?") files. These files were originally defined by John Carmack while building the original version of Doom and are thus interchangeable, giving ViZDoom a breadth of options for providing AI agents with learning environments ranging from the official Doom levels to custom scenarios.

ViZDoom has been used by researchers in two generic settings, a competition setting in which the principle objective is to survive some form of 'deathmatch' with other agents. Particular examples of this include deploying the best deep learning RL approach from other general game playing challenges [19], or modifying a general deep learning RL approach to include properties specifically useful for VizDoom, e.g. some form of prior task decomposition [13, 21].

A second use of ViZDoom has been through the use of custom scenarios representing underlying tasks of value for developing better general purpose visual learning agents. For example, [1] use a health pack gathering scenario in which the objective is to avoid mines and collect health packs, whereas [18] assume a single scenario involving learning to aim against a static target. One interesting observation from Alvernaz and Togelius was that the deep learning element might produce a representation that failed to recognize hazards [1].

3 Tangled Program Graphs

The tangled program graph (TPG) represents a generalization of GP teaming into the *emergent* discovery of graphs of teams of programs [8, 9]. To do so, a definition of GP is assumed in which each program provides a single (real valued) scalar output, b_i, which is interpreted as the 'bid'. Each program also has a single discrete atomic action, a_i, selected from the set of task specific atomic actions $\mathcal{A}$. Two populations (P and H) representing programs and teams respectively are then coevolved in a symbiotic relationship, or symbolic bid-based GP (SBB) [15, 14]. Fitness is only defined explicitly at the team population. Each member of the team population identifies a unique subset of programs from the program population (the same program may appear in multiple teams). A variable length representation is assumed for members of each population, hence team complement evolves in the team population and program complement evolves in the program population.

SBB is independent of the specific representation assumed for a program. In this work we adopt a linear GP representation (as did the original work [15, 14]), thus instructions take the form of operations on registers of the general form: $R[x] = R[x] < op > R[y]$, where x and y are register references and $< op >$ is an 'opcode'. Likewise we also adopt the eight opcodes common to Lichodzijewski's work: $\{+, -, \times, \div, ln, cos, cos, IF\}$, where the conditional is interpreted as IF $(R[x] < R[y])$ THEN $(R[x] = -R[x])$. Each instruction also has a mode bit which enables $R[y]$ to either index a register or an input attribute (in this case the visual screen buffer). The $R[y]$ references are always to registers of which there are a total of 8 in this work.

The output of a team is established by executing each program (from the same team) on the current state of the environment, $\vec{s}(t)$, and identifying a single 'winning' bid, i.e. the program with maximum output [15, 14]. Let this be p^*. Program p^* has won the right to suggest its action, call this $a(t)$. Under a reinforcement learning task, action $a(t)$ defines the action of the agent at time step t. Such an action potentially changes the state of the environment and the process repeats until some end condition is encountered (e.g. terminal state of the game, maximum number of training interactions). Fitness of a team can then be assessed using an appropriate measure of game score (Section 4). The process repeating over all teams, or a training 'generation'.

After each generation the teams are ranked (Step 2(b)v, Algorithm 1) and the worst performing H_{gap} individuals deleted (Step 2(b)vi). Any programs from the program population lacking a reference from at least one team are also then deleted (Step 2(b)vii). H_{gap} parents are then identified, and an equivalent number of child teams produced (care of a set of variation operators) to replace the deleted teams (Step 2(b)ii). One set of variation operators operate at the level of teams [15, 14]: add or delete a reference to a program (P_a, P_d), modify a program currently in the team (P_m), or modify the action of a program within the team (P_a). Likewise a second set of variation operators operate at the level of programs [15, 14] (only called upon when P_m test true), 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}), mutate an instruction (P_{mut}) or swap two instructions within the same program (P_{swp}).

SBB provides a framework for evolving teams of arbitrary complement, i.e. without having to specify the number of programs per team [15, 14]. Moreover, only the size of the team population needs explicitly defining (H_{size}). The size of the program population floats as programs either fail to be referenced by a team (delete program from P) or a program is cloned (add program to P).

TPG assumes SBB as the starting point and introduces an additional variation operator, P_α, which establishes the *type* of action change when P_a tests true. Thus, for P_α false, the cloned program's action, a_i is selected from the set of task specific atomic actions $a_i \in \mathcal{A}$, whereas for P_α true, the new action is a pointer to any team in the team population of generation g, or $a_i \in H(g)$. Once this happens we have a graph in which each 'node' is a team and 'arcs' are programs. Evaluation of a graph follows the same process as per SBB, and commences with the root team (all individuals in the initial team population are root nodes), w.r.t. state $\vec{s}(t)$ of the environment. However, if the action of the winning program is a pointer to another team, then the new team is also evaluated on state, $\vec{s}(t)$. As teams in the graph are evaluated they are 'marked'. If following execution of a team the winning program's action selects a team that was previously visited (on state $\vec{s}(t)$), then the runner up program's bid is (recursively) assumed as the action until either an unvisited team or an atomic action is selected (each team must have at least one atomic action). As soon as the winning program's action takes the form of an atomic action, evaluation of the graph (for state $\vec{s}(t)$) is complete and any 'marked' teams are reset.

Algorithm 1 TPG algorithm (Section 3) with incremental record keeping (Section 4). The term 'agent' is used to denote a TPG individual. Note that only teams representing root nodes constitute an agent for which fitness evaluation is performed.

```
1. Initialize team population, $H(g = 0)$, and program population, $P(g)$.
2. For $(b = 0; b < \text{Bags}; b = b + 1, g = g + 1)$
    (a) $\mathcal{S}(b) = \mathcal{S}$                                        ▷ Initialize set of task scenarios
    (b) For all $(t \in \text{Scenarios})$
        i. $s(b) = rnd(\mathcal{S}(b))$                                     ▷ Select a task scenario
        ii. $H(g) = H(g) \cup create(H_{gap})$
        iii. For all $(agent \in H(g))$ and $(k \in \text{Episodes})$
            A. $task = rnd(s(b))$                                          ▷ Initialize task scenario instance
            B. evaluate$(agent, task)$                                     ▷ Eq. (1)
        iv. Update OverallFitness over scenarios $\{t, ..., t - R\}$       ▷ Eq. (2) and (3)
        v. Rank $(agents \in H(g))$
        vi. Prune lowest ranked $H_{gap}$ agents from $H(g)$
        vii. Prune all $p \in P(g)$ without an agent
        viii. $\mathcal{S}(b) = \mathcal{S}(b) - s(b)$                      ▷ Remove evaluated task scenario
```

The number of root teams varies during evolution. That is to say, if a root team receives a pointer from another team (care of the variation operators), then that graph has been incorporated as a subgraph into another root team's graph. Naturally, the application of P_a, P_α can also result in a subgraph being decoupled from its parent, resulting in the reappearance of a root team (a root has no incoming arcs). The resulting complexification of single team root nodes into graphs is therefore an emergent process in which the complexity of the relation between teams, team complement and program complexity are all driven by the underlying complexity of the task [8, 9].[2]

4 Scaling TPG to large visual state spaces

In the following we summarize the additional heuristics investigated to deploy TPG under the ViZDoom environment. Specifically, TPG was originally deployed under the Atari arcade learning environment, [2], with a down sampled visual state space, $\vec{s}(t)$, from the original 210×160 pixels to 42×32 pixels.[3] ViZDoom, on the other hand, represents a 3D environment capable of presenting a much richer range of encounters than under individual arcade console titles [10]. With this in mind we introduce five factors for specifically scaling TPG up to the challenge of evolving agents capable of solving challenges present in ViZDoom.

4.1 Representation of Features

The default structure for screen buffer data in ViZDoom is a series of nested arrays of integers, where the outer array represents the height of the buffer (240 rows of pixels) and the inner arrays represent the RGB integer values of pixels in a given row (320 pixels per row) [10]. In order to provide SBB with a 'flat' set of environment registers, the three RGB values were concatenated into a single 32-bit integer format, thus defining input as an array of 76,800 integers to represent each frame received from the screen buffer. This is distinct from deep learning approaches in which each of the RGB channels is subject to convolution filtering. Moreover, the computational cost of retaining independent channels with deep reinforcement learning might require some form of cropping or down sampling from the resolution of the original input.[4] In effect, GP 'sees' a single channel consisting of pixel values over a much larger dynamic range (due to the concatenation of the three RGB channels into a single integer value) as opposed to retaining the representation of each pixel in three colour channels over a lower dynamic range.

4.2 Task scenarios

As the complexity of task domains increases, it might be beneficial to learn from specific training scenarios or source tasks. Such approaches have proved effective in a wide range of settings including ViZDoom competitions [21] and the evolution of soccer skills [20].

The ViZDoom distribution has 8 example scenarios: *Basic*, *Deadly Corridor*, *Defend the Centre*, *Defend the Line*, *Health Gathering*, *My Way Home*, *Predict Position*, and *Take Cover*.[5] Naturally each scenario represents a different skill that might be beneficial for a general purpose policy capable of 'surviving' within the ViZDoom environment. In addition to the scenarios included with the package, two further scenarios were constructed in order to promote a particular type of movement called circle-strafing. Circle-strafing is when a player continuously focuses their aim on a target in the game while sidestepping (strafing). If an agent does both of these things, they will naturally walk in a circle around their target, potentially making it more difficult for the the target to hit the agent (because it is moving) while decreasing the health of the target.

Circling scenario 1 places the player on one side of a raised ring (6 o'clock). The ring itself causes a small amount of damage to the player periodically. The floor inside and outside of the ring causes a moderate amount of damage if a player stands on it for too long. The player is tasked with picking up a health kit on the other side of the ring (12 o'clock). Whenever a health kit is acquired, another health kit spawns on the opposite side of the ring. This scenario is meant to help the player specifically perform circular movements, though not necessarily with strafing involved. There are three atomic actions made available $\mathcal{A} = \{F, TL, TR\}$ (see Table 1) and the rewards are $+1$ per time unit alive, -100 on agent death, and there is a scenario time limit of 2100 time units.

Circling scenario 2 is a continuation of the first. The ring remains the same, except health kits spawn sequentially at every 90 degree section of the ring rather than on opposite sides. An enemy spawns in the middle of the ring. The player is tasked with moving to acquire the health kit, but has the additional requirement of shooting the enemy in the centre of the room in order to make the next health kit spawn. Note that the turning speed in ViZDoom is static when using a key (rather than a mouse) and will not allow the player to run to a health kit, turn,

[2]For an illustration of the incremental construction of TPG individuals see the presentation slides of Stephen Kelly from EuroGP'17 `http://stephenkelly.ca/research_files/skelly-mheywood-eurogp-2017.pdf`

[3]Conversely, the deep reinforcement learning approach of [17] cropped the original visual space to 84×84 pixels.

[4]For example, [1] assume a 120×160 visual input and [13] assume 60×108.

[5]`https://github.com/mwydmuch/ViZDoom/tree/master/scenarios`

Table 1: Parameterization of VizDoom environment and training scenarios. Total number of training encounters is Bags $\times$ Scenarios $\times$ Episodes

Parameter	Value
Dimension of visual input space	$320 \times 240 = 78,800$
Colour Settings	RGB24
Atomic action set ($\mathcal{A}$)	Move forward (F), Move backward (B), Turn Left (TL), Turn Right (TR), Strafe Left (SL) Strafe Right (SR), Shoot/Attack (A)
Record Keeping (R)	2
Bags	6
Scenarios	10
Episodes	20

shoot the enemy, and then return to moving. In order to be successful, the player will need to learn to circle-strafe around the ring in order to stay alive. There are six atomic actions made available $\mathcal{A} = \{F, TL, TR, SL, SR, A\}$ (see Table 1) and the rewards are $+1$ per time unit alive, -100 on agent death, and there is a scenario time limit of 2100 time units.

Note that in all cases, there is no special 'task scenario' flag provided as an additional input. The agent has to infer the nature of the task from the visual clues in the environment itself.

4.3 Graduated Action Space

Programs can potentially assume any atomic action, $a_i \in \mathcal{A}$. However, different learning scenarios might only explicitly require a subset of the atomic actions. Given that at initialization the programs are initialized randomly, limiting programs to a subset of the total set of atomic actions might be beneficial. Thus, instead of allowing all actions to be used from the beginning of the first generation, we gradually add new actions to the existing pool as they are required. Thus, the first scenario will determine the starting actions of the population $\mathcal{A}_0$. When the next scenario, k, is introduced, any additional actions, $\mathcal{A}'$, specific to the scenario are added, or $\mathcal{A}_k = \mathcal{A}_{k-1} + \mathcal{A}'$. Thus, new programs will be able to sample from an incrementally expanding set of atomic actions. As the additional scenarios are attempted, new actions are added until all scenarios have been encountered. This allows TPG to gradually integrate the new actions into the existing program graphs.

4.4 Random Bags

In order to reduce unintended sequencing biases into the order with which scenarios are introduced during evolution, we randomize their order. Thus, each scenario is presented for a set of episodes. The collection of scenarios is placed into a *bag*, where they are shuffled randomly to produce a scenario permutation (Step 2(b)i, Algorithm 1). Each agent experiences the task scenarios in the permutation order specific to the bag (Step 2(b)iii). Once a bag is empty (all of the tasks have been performed) a new bag (ordering of scenarios) is created.

4.5 Fitness through incremental record keeping

One of the most difficult aspects of learning multiple tasks from a single continuous run is forgetfulness, e.g. [11]. Forgetfulness implies that as agents experience different task scenarios, they specialize on their most recent task, and slowly forget the nuances of the tasks that were encountered earlier. In short, if agents are unable to discover how to leverage what they have previously learnt from earlier tasks into the new task, they risk loosing important properties.

Ideally, we would evaluate agent performance on all tasks at each generation. This might lead to a multi-objective evaluation through Pareto dominance, e.g. [7, 9]. However, in this case we have additional factors to consider, including the cost of maintaining an absolute measure of fitness over the 10 visual reinforcement task scenarios for the entire population at any point in time, i.e. each task scenario requires evaluation over multiple task initializations ('Episodes' parameter, Table 1).

In essence, we are attempting to strike a balance between an absolute performance function (as measured across all task scenarios) and measuring performance on the single current task scenario. Thus, an individual is not considered fit merely for their ability to solve the current task, but based on their ability to solve the most recently attempted R tasks. Once an agent's current task fitness is calculated, we then go back and retrieve fitness scores for the agent in the previous R tasks. Naturally, the H_{gap} agents created in the current task evaluation (Step 2(b)ii, Algorithm 1) lack appropriate fitness evaluations for the R previous tasks. In these cases, additional evaluations are performed to establish fitness on the past R tasks. Thus, at each generation up to $H_{size} - H_{gap}$ agents are evaluated on the current task, and up to H_{gap} agents on the current and R previous tasks.[6]

Fitness scores are calculated in the following way:

1. Estimate the raw fitness for each agent across a set of episodes for the current task scenario (Section 4.2). Thus, the raw fitness of agent i on task t is:

$$RawFitness(i,t) = \sum_{k=1}^{Episodes} score_k(i,t) \tag{1}$$

2. Normalize all fitness values by dividing every agent's by the maximum:[7]

$$NormFitness(i,t) = \frac{RawFitness(i,t)}{max_k(RawFitness(k,t))} \tag{2}$$

3. Estimate overall fitness of agent i in terms of the performance over the last R tasks

$$OverallFitness(i) = \sum_{k=0}^{R} NormFitness(i,t-k) \tag{3}$$

   Note that Eq. (3) is limited to the evaluated scenarios in the case of a 'code start' (first bag with less than R scenarios evaluated).

Once the overall fitness is calculated for all agents, they are ranked in descending order and the worst H_{gap} agents removed. In short, we are attempting to balance the cost of evaluation across all task scenarios versus reduced accuracy introduced by only estimating fitness over a subset of tasks. Moreover, in continuously shuffling the order of scenario evaluation we hope to mitigate the introduction of other biases. Unlike the previous instance of multi-task learning with TPG [9], we anticipate some continuity between the tasks encountered, but this same continuity also potentially makes it more difficult to resolve which task an agent is facing. Moreover, in this work we are facing a total of 10 tasks as opposed to a maximum of 3 tasks in [9].

5 Results

TPG is based on the symbiotic bid-based (SBB) teaming framework from Lichodzijewski et al. [15], for which there are several code bases. In this work, we assumed the open source Java code base of SBBJ (`https://web.cs.dal.ca/~rsmith/_sbb_fcube/`) and added additional functionality to support TPG as summarized in Section 3.

Training sessions were divided into permutations of scenarios, with each scenario appearing for a number of episodes, or a bag (Section 4.4), Table 1. In total, 6 bags (entire rounds through all the task scenarios) are conducted and results summarized for 12 independent runs. All runs were performed on desktop computing platforms without recourse to parallel hardware. Table 2 summarizes the TPG algorithm parameters assumed in this work.

After all the episodes from the same task scenario conclude Eqn. (1) can be estimated. This reward is normalized relative to the highest scoring agent in order to determine the fitness of agents, Eqn. (2). This normalization is important due to the mis-alignment of reward scales across the various scenarios (see for example the difference between column values in Table 3). Once all teams are given a fitness value for the current task scenario, then the normalized fitness values from the previous R completed scenarios are retrieved for each agent or Eqn. (3). For the purposes of this experiment, $R = 2$, (Table 1), implying that the three most recently attempted scenarios are considered in fitness calculation.

[6]Note that H_{size} is the number of teams present and reflects the number of agents (root teams) at initialization. However, as teams are subsumed into graphs, the number of agents (root teams) will decrease. Likewise, application of the variation operator could switch an action from a team pointer to an atomic action, breaking a graph into two smaller graphs, resulting in an increase in the number of agents.

[7]Any negative normalized fitness values are treated as 0, thus producing a number in the range $[0,1]$

Table 2: Parameterization of Tangled Program Graph algorithm.

Teams		Learners	
Parameter	Value	Parameter	Value
Team Population Size (H_{size})	450	Max. Instructions	1024
Team Gap (H_{gap})	50% of Root Teams	Prob. Delete Instr. (P_{del})	0.5
Max. Prog. per Team (ω)	9	Prob. Add Instr. (P_{add})	0.5
P_d,P_a	0.7	Prob. Mutate Instr. (P_{mut})	1.0
P_m	0.2	Prob. Swap Instr. (P_{swp})	1.0
P_n	0.1		

Table 3: Best multi-task agent and best single-task agent vs. Human. In all cases larger scores are better. Performance reflects game score over 200 test games for each scenario for each agent (1st, 2nd, 3rd quartile performance). Human score reflects a single best play by an experienced player. $\{^a, ^b, ^c, ^d\}$ identify each single-task agent and the subset of tasks on which they were best.

	Multi-task agent			Single-task agent			
Task Scenario	Q1	Median	Q3	Q1	Median	Q3	Human
Basic	59	63	68	74	76.57^a	79	76
Deadly Corridor	272.8	393.5	508.8	258.5	470.7^d	653.08	2280
Defend the Centre	9	11	12	8	10^b	12	16
Defend the Line	5	9	11	9	11^b	13	25
Health Gathering	595.75	642	778.25	1006	1021^c	1032	1065
My Way Home	0.81	0.86	0.88	0.813	0.834^d	0.869	0.982
Predict Position	-0.071	0.88	0.951	-0.055	0.892^a	0.919	0.965
Take Cover	125.75	172	215.25	147.25	180^a	216.25	1445
Health Circle	57	73	88	82	110.5^c	143.25	848
Circle Strafing	18	24	35	17.75	23^c	29	632

The overall fitness score then allows all agents (root teams) to be ranked according to their overall utility across multiple scenario types. Since the assortment of scenarios is randomly bagged, there is no predictability of previous records in practice. Thus, given enough generations, each agent should be evaluated across all permutations of scenarios bound by $R + 1$.

5.1 Human Results

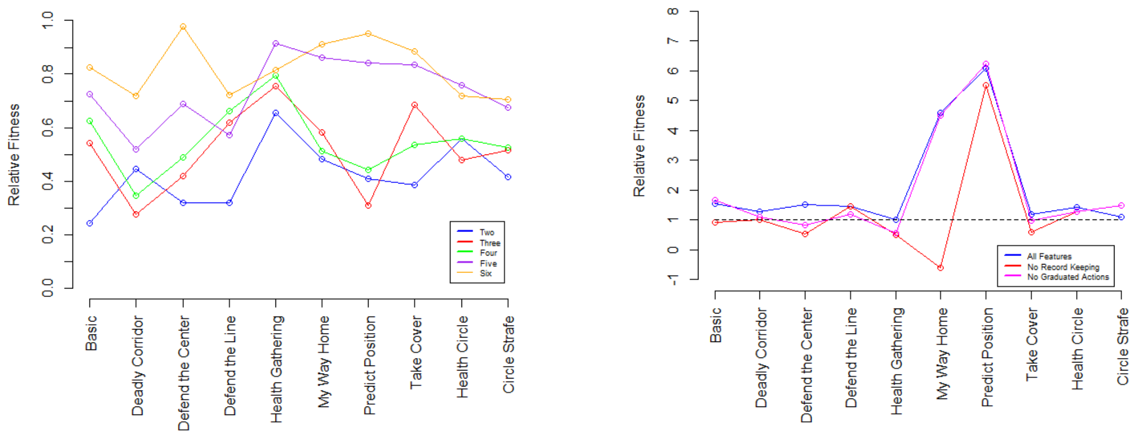
The human results were gathered from a full test run by the researchers. Each scenario was played 10 times with the same mechanical restrictions presented during a normal training scenario. The values in Table 3 reflect a moderate level of familiarity with the ViZDoom control scheme and a high amount of familiarity with first person shooters in general. The labels in Table 3 have the following interpretation: 1) *Singe task* reflects the best single agent on a specific task scenario (larger scores are better). Hence, this need not be the same agent on each task. 2) *Multi-agent* reflects the best agent across all task scenarios. It is interesting to note that the 'single best agents' also actually returned best performance on more than one game. Moreover, the Multi-agent actually had better median performance for 3 of the 10 tasks.

Some of the human results shown in Table 3 are the result of discovering how to exploit the specifics of the task environment. For example, the Take Cover scenario has a simple dominant strategy where you slowly sidestep all the way to one side and then sprint back to the other side to avoid all projectiles. Similarly, the Deadly Corridor scenario is seemingly difficult for learning agents, but generally simple for human players to complete. The score of 2280 in this scenario is reasonably close to the maximum for Deadly Corridor. This also applies to the result in Health Gathering, which requires a player to continuously collect health kits on the floor in order to not die. Since this scenario is timed, understanding the bounds of the problem ahead of time provide prior knowledge that a human player is able to use to their advantage. In short, humans are still most effective at playing task scenarios of the VizDoom environment.

Relative to contemporary research using deep learning or hybrid deep learning and neuroevolution, we note that results only appear to exist for the 'Basic' and 'Health Gathering' task scenarios. Moreover, such results

Table 4: Comparison with deep learning results under 'Basic' and 'Health Gathering' task scenarios. Skip Count reflects the frequency with which an agent is required to make decisions.

Framework	Skip Count	Basic task Average score (Std. Dev.)	Health Gathering task Average score (Std. Dev.)
Deep learning [10]	0	51.5(±74.9)	unknown
Deep learning [10]	4	82.2(±9.4)	≈ 1,300
Hybrid [1]	unknown	unknown	657.1(±397.1)
TPG Multi-task	0	63.3(±4.91)	680.8(±97.7)
TPG Single-task	0	76.6(±4.5)	1020.9(±18.7)



(a) Relative to best fitness

(b) Relative to average game score

Figure 1: Score of the top team across an increasing number of scenario sessions (bags) using (a) normalization of single best agent, and (b) normalization relative to average game score.

represent the performance of the agent trained on a single task. Caveats aside, Table 4 compares scores for each agent under the two tasks for which we could find comparative results. Note also, that it is also likely that different numbers of testing events are assumed in each case (for TPG we used 200 task initializations).

Kempka et al. recognized that increasing the Skip Count from zero (requires the agent to make a decision at each frame) to an optimal value of 4 skipped frames had a significant positive impact on the quality of deep learning agent policies [10]. All TPG runs were performed with a Skip Count of zero. We note that the comparator solutions are only significantly better than TPG when the non-zero Skip Count was employed. Future work could revisit this parameterization.

5.2 Population and Team Results

Incremental population champions were saved at the conclusion of each bag, the skills of which are shown in Figure 1(a). This graph shows the normalized scores of that bag's champion when compared to the highest scoring individual of each category. It should be noted that the champion is not necessarily the same root team at each level or task scenario.

As the number of bags increases, we see that the range of scenario scores are slowly moving toward unity, but do so relatively uniformly. For the first three bags only the Health Gathering task saw any significant improvement and thus the results show very low rewards. In the later bags, individual teams were capable of successfully completing those scenarios and drew their relative position higher. Since this experimental design rewarded better overall capabilities rather than localized specialization, this is an outcome we would expect to see as the TPG agents become more sophisticated throughout the learning process.

Figure 1(b) illustrates the impact of removing key scaling factors from the approach as outlined in Section 4. That is to say, 12 independent runs were also performed with a single missing scaling factor, after which we again summarize performance relative to the best agent from each task scenario. Specifically, without incremental record keeping during fitness evaluation (Section 4.5), it is not possible to maintain a consistent front of development

Table 5: TPG Model complexity for Multi-task agent. Num. Teams/Instr/Pixels represent the average number of Teams (graph nodes), Instructions executed (including introns), Pixels indexed per decision

Task Scenario	Num. Teams	Num. Instr	Num. Pixels
Basic	3.18	17,238	10,951
Deadly Corridor	3.53	18,038	11,847
Defend the Centre	2.86	16,081	10,692
Defend the Line	3.34	16,709	11,637
Health Gathering	3.41	16,524	10,685
My Way Home	3.91	17,423	11,225
Predict Position	3.37	16,952	11,749
Take Cover	2.76	15,746	11,488
Health Circle	3.34	17,785	10,653
Circle Strafing	3.47	17,537	11,866

across all task scenarios. Specifically, the 'MyWayHome' task scenario seemed to fail to improve at all without incremental records. Removing graduated actions did not seem to have a significant positive or negative effect.

The structure of the champion TPG agent is illustrated in Figure 2. This individual corresponds a 'Multi-task agent' with performance as summarized in Tables 3 and 4. Each node represents a team (of programs) and each arc indicates the action of the program, i.e. pointer to either another node (a different team of programs) or an atomic action. There are a total of 18 teams, organizing a total of 72 programs which index a total of 33,708 unique pixels from the original visual input (or 43.9%). However, only a fraction of these programs (pixels) are executed (indexed) when TPG determines what action to suggest for any given state, Table 5. Thus, in order to make any particular decision between 14% (min) to 15.5% (max) of the input space is actually used. Likewise, only 2 to 4 teams are evaluated per decision, implying that although the TPG solution might be 'complex', the path from root to action is still short.

6 Conclusion

We demonstrate the scaling up of tangled program graphs (TPG) to a set of visual reinforcement learning tasks described in the ViZDoom environment. Specifically, TPG agents are evolved from the entire content of the $320 \times 240 = 76,800$ pixel state space. Several mechanism are investigated for achieving this scaling of which basing the fitness function on recent previous task scenarios as well as the current task scenario appears to be the most significant.

TPG is still able to discover solutions that only index a fraction of the input space and only executes a subset of the programs in order to make each decision. At the same time, scaling across multiple task scenarios is demonstrated, where this is achieved through the ability to organize over 70 programs hierarchically using the discovery of appropriate graph connectivity. This capacity for emergent task decomposition/modularity provides a unique approach to discovering solutions under visual reinforcement learning problems. It is also fundamental to providing solutions that can be discovered and deployed without specialized hardware support (as is generally the case for deep learning). Comparison with contemporary deep learning results (for a subset of the task scenarios) indicates that the quality of the resulting policies does not appear to be compromised.

Acknowledgments. This research was supported by NSERC grant CRDJ 499792.

References

[1] Alvernaz, S., Togelius, J.: Autoencoder-augmented neuroevolution for visual Doom playing. In: IEEE Conference on Computational Intelligence and Games. pp. 1–8 (2017)

[2] Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research (2012)

[3] Braylan, A., Hollenbeck, M., Meyerson, E., Miikkulainen, R.: Reuse of neural modules for general video game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 353–359 (2016)

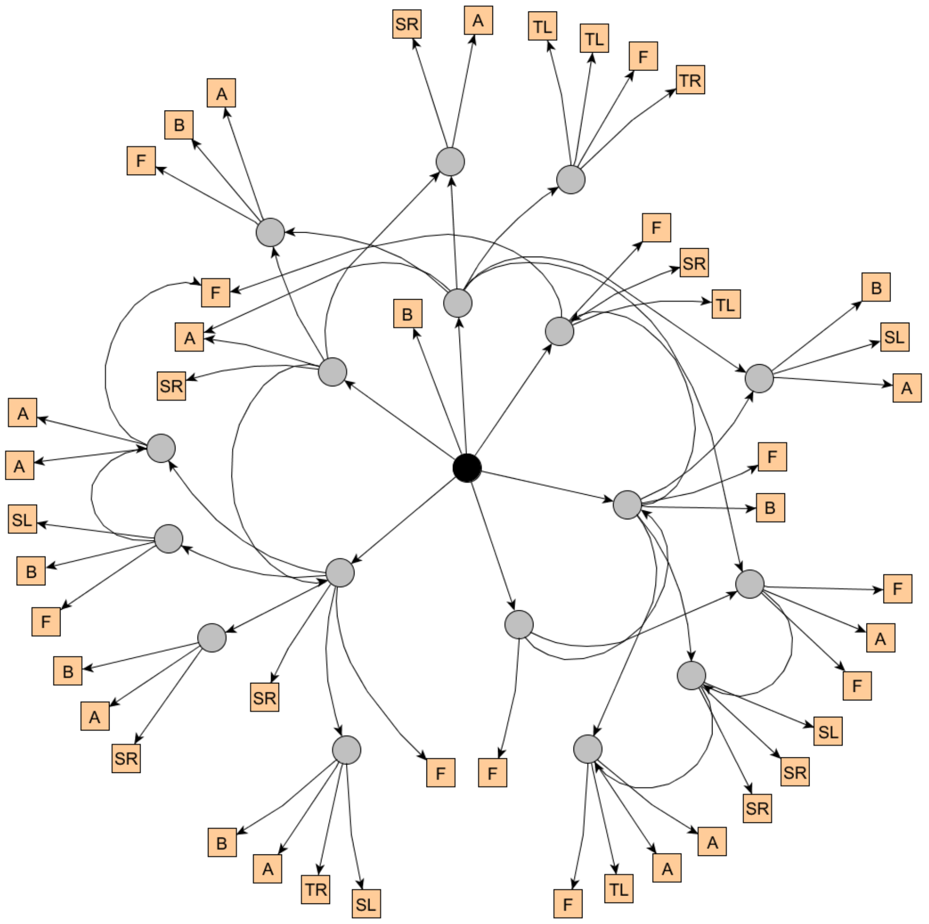


Figure 2: TPG agent with highest average score over all tasks. The root node (black) defines the starting point for agent policy evaluation. The process of action selection follows the path defined by the arc of each node's winning bid until an atomic action is chosen. Because the path through the TPG graph is based on sequential node-wise bidding in which no node can be revisited, the search for an atomic action represents a low computational cost.

[4] Guo, X., Singh, S., Lewis, R., Lee, H.: Deep learning for reward design to improve Monte Carlo Tree Search in atari games. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 1519–1525 (2016)

[5] 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)

[6] Jia, B., Ebner, M.: Evolving game state features from raw pixels. In: European Conference on Genetic Programming. LNCS, vol. 10196, pp. 52–63 (2017)

[7] Kelly, S., Heywood, M.I.: Knowledge transfer from Keepaway soccer to half-field offense through program symbiosis: Building simple programs for a complex task. In: ACM Genetic and Evolutionary Computation Conference. pp. 1143–1150 (2015)

[8] Kelly, S., Heywood, M.I.: Emergent tangled graph representations for Atari game playing agents. In: European Conference on Genetic Programming. LNCS, vol. 10196, pp. 64–79 (2017)

[9] Kelly, S., Heywood, M.I.: Multi-task learning in Atari video games with emergent tangled program graphs. In: ACM Genetic and Evolutionary Computation Conference. pp. 195–202 (2017)

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

[11] Kirkpatrick, J., Pascanu, R., Rabinowitz, N.C., Veness, J., Desjardins, G., Rusu, A.A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., Hadsell, R.: Overcoming catastrophic forgetting in neural networks. CoRR abs/1612.00796 (2016)

[12] Kunanusont, K., Lucas, S.M., Pérez-Liébana, D.: General video game AI: Learning from screen capture. In: IEEE Conference on Computational Intelligence and Games. pp. 2078–2085 (2017)

[13] Lample, G., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 2140–2146 (2017)

[14] Lichodzijewski, P.: A symbiotic bid-based framework for problem decomposition using Genetic Programming. Ph.D. thesis, Faculty of Computer Science, Dalhousie University (2011)

[15] Lichodzijewski, P., Heywood, M.I.: Symbiosis, complexification and simplicity under GP. In: Proceedings of the ACM Genetic and Evolutionary Computation Conference. pp. 853–860 (2010)

[16] Loiacono, D., Lanzi, P., Togelius, J., Onieva, E., Pelta, D., Butz, M., Lonneker, T., Cardamone, L., Perez, D., Sáez, Y., Preuss, M., Quadflieg, J.: The 2009 simulated car racing championship. IEEE Transactions on Computational Intelligence and AI in Games 2(2), 131–147 (2010)

[17] 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)

[18] Poulsen, A.P., Thorhauge, M., Funch, M.H., Risi, S.: DLNE: A hybridization of deep learning and neuroevolution for visual control. In: IEEE Conference on Computational Intelligence and Games. pp. 1–8 (2017)

[19] Ratcliffe, D.S., Devlin, S., Kruschwitz, U., Citi, L.: Clyde: a deep reinforcement learning DOOM playing agent. In: AAAI Workshop on What's Next for AI in Games. pp. 983–990 (2017)

[20] Whiteson, S., Kohl, N., Miikkulainen, R., Stone, P.: Evolving Keepaway soccer players through task decomposition. Machine Learning 59(1), 5–30 (2005)

[21] Wu, Y., Tian, Y.: Training agent for first-person shooter game with actor-critic curriculum learning. In: International Conference on Learning Representations. pp. 1–10 (2017)

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