REINFORCEMENT LEARNING WITH REAL VALUED TANGLED PROGRAM GRAPHS

by

Ryan Amaral

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2021

Dedicated to Zoe.

Table of Contents

List of Tables

List of Figures

Abstract

Tangled Program Graphs (TPG) represents a framework for evolving programs under an explicitly emergent model for modularity. The framework has been very successful at discovering solutions to tasks with delayed rewards (reinforcement learning) when the actions are limited to a single discrete action per state. In this thesis, an approach is proposed for generalizing TPG to the case of *multiple* real-valued actions per state. Two empirical benchmarking studies are performed to demonstrate these outcomes: ViZDoom over multiple tasks, and bipedal walker control. The former is used to compare to original TPG with single discrete actions per state, the later is used to demonstrate multiple real-valued actions per state. It is shown that the complexity of the resulting solutions decreases considerably compared to the original TPG formulation. However, in order to reach these results, significant attention has to be paid to the adoption of appropriate diversity mechanisms. This thesis therefore also proposes a framework for intermittently injecting new material into the TPG population during training. The modular properties of TPG enable this material to be absorbed on a continuous basis. Results are comparable with those identified under certain recent deep learning approaches.

Acknowledgements

First and foremost, the bulk of my thanks is given to Dr. Malcolm Heywood, my research supervisor. Plenty of ideas were tried throughout my degree, some working better than others. Ultimately I ended up with a thesis that I am proud of, credit due to Dr. Heywood.

Plenty of good times were shared between us members of Dr. Heywood's lab. From game nights, to research meetings, to publishing papers together. It was a great group to be a part of, certainly relationships to hold on to.

Lastly all of my non-research related relationships, friends and family are to be thanked for the support and interest in my work.

Chapter 1

Introduction

Genetic Programming (GP) with a "linear representation" attempts to construct a search for sequences of instructions in an imperative programming language [4], e.g. C or a task transfer language (processor independent assembly language). In the most general case such programs assume a variable length representation, in some cases solutions are allowed to evolve up to a maximum number of instructions. This is equivalent to learning the overall topology of a machine learning solution as well as the parameter values. Conversely many machine learning algorithms concentrate on finding the parameter values given a prior topology, e.g. the weight values for a pre-specified neural network architecture. In addition, genetic programming can also benefit from the ability to reuse code or encourage modularity. To do so, specific mechanisms are often introduced in an attempt to spot candidate code sequences for 'modularization' within a piece of 'non-modular' code [16].

This work takes a different approach to evolving modular solutions. Specifically, a symbiotic two population framework is assumed in which every program is a module. Programs have to answer the question as to when to perform an action, rather than attempt to directly define an action. Symbiosis implies that two populations coevolve, with one population searching for useful teams of programs and a second population searching for useful programs. Previous research has shown that such a framework (hereafter symbiotic bid based GP or SBB for short) is particularly effective under applications defined in terms of discrete scalar rewards, e.g. classification [38, 39] or a subset of reinforcement learning tasks [12, 28, 55]. The framework was also previously generalized to enable actions to include references to other teams, thus providing the basis for hierarchical and graph-like forms of modularity (Tangled Program Graphs, TPG for short) [24, 25, 27, 56, 57].

A fundamental limitation however has been that TPG and SBB are limited to

single discrete scalar actions per state. Previous research has shown that it is possible to learn real-valued actions for reinforcement learning tasks with memory, albeit limited to single outputs [29, 69]. In this work we are interested in addressing a related but distinct question. Can we develop TPG and SBB in such a way that the modular decomposition is retained, but actions can now take the form of multiple real-valued functions? Thus, the resulting agents are capable of benefiting from task decomposition (through TPG and SBB), but produce multi-dimensional vectors of action *per* state. Success in this regard opens the opportunity to address problems from robotics and real-valued reinforcement learning that have previously eluded both TPG and SBB.

1.1 Thesis Objectives

The TPG algorithm provides a framework for scaling GP to challenging tasks through modularity [24, 25, 27]. To do so, it simultaneously performs a search for good team members and programs. TPG assumes that programs represent modules and that each module expresses context (through program execution) and a scalar action. An action can take one of two forms, either a pointer to another team, or a single (scalar, likely discrete) instance of a task specific atomic action. Thus, in order to apply TPG to the Arcade Learning Environment (suite of Atari style console games [3]) the set of atomic tasks are defined in terms of the eight discrete directions that the joystick can take, both with and without a button press, plus the button press alone, and a nil action (i.e. a total of 18 actions). This implies that there can only ever be a single action at any state, and that actions can only be discrete values. TPG being designed in this manner can represent a considerable limitation when attempting to apply TPG to tasks with real-valued actions and/or tasks that benefit from being able to take *multiple* actions per state.

The primary goal of this thesis is to expand TPG's capabilities by enabling real valued vector outputs. Specifically, the limitation of discrete scalar actions implies that it is not possible to scale TPG to real-valued action spaces as discrete enumerations run into the curse of dimensionality. Moreover, the objective of this thesis is to do so without compromising on the other desirable properties of TPG, i.e. open ended modularity.

The hypothesis of this thesis is that replacing each module's scalar action with a linear GP program is sufficient for mapping between state and real-valued action. However, additional properties will also be investigated:

- Linear GP is actually capable of providing multiple outputs, thus multiple real-valued actions per state are to be supported, and;
- Given that the action associated with each TPG module is now a program, the complexity of a TPG individual (from the perspective of the number of modules per TPG graph) will also potentially decrease. This can lead to TPG graphs that are potentially less complex than under TPG with discrete scalar actions.

This thesis will use two reinforcement learning environments to demonstrate the effectiveness of the proposed approach: finding policies for simultaneously solving five ViZDoom benchmark tasks [31] and identifying policies for a bipedal walking benchmark from OpenAI Gym (Bipedal-Walker-v3) [5].[1] The ViZDoom environment represents a high dimensional partially observable task in which an agent is rewarded for surviving in different task scenarios. The bipedal walker represents a difficult real-valued control problem in which multiple actions have to be specified per state. Moreover, this thesis also addresses issues such as diversity maintenance with the multi-task setting from ViZDoom providing one source of variation and partial restarts during evolution being used with the bipedal walker. In both experiment sets a method of performing multiple mutations at a time is also used, though not necessarily analyzed.

1.2 Thesis Outline

This thesis is structured as follows: In chapter 2: Background, relevant topics are introduced to give an understanding of Reinforcement Learning (the broad classification of problems being addressed in this work), Continuous Control (encompasses the bipedal walker task used in this work), Evolutionary Algorithms, GP, SBB, and TPG.

Following that, in chapter 3: Real Valued Tangled Program Graphs (hereafter TPGr), the real value action implementation used for TPG in this thesis, is described

[1] `https://github.com/openai/gym/blob/master/gym/envs/box2d/bipedal_walker.py`

in detail. Descriptions of the potential use cases of TPGr is also provided. Note that the term TPG is used to describe TPG in general (real valued or discrete), and TPGr is used in specific reference to the real valued implementation/update, while TPGd will refer specifically to the original discrete formulation. TPGr is not so much a new version of TPG, it is more-so an option within TPG.

Then in chapter 4: Diversity Maintenance, the processes used to increase/maintain the level of diversity within GP, SBB, and TPG are described. This includes the concept of rampancy, interleaving SBB individuals into a TPG population under the biped walker task, and task sampling under ViZDoom.

Chapter 5: ViZDoom Experiments describes the experiments performed on the ViZDoom tasks, where the emphasis is on comparing TPGr with TPGd under a single action per state setting, one in which the state space is a 3D high-dimensional multi-task setting.

Chapter 6: Biped Walker Experiments describes the experiments performed on the bipedal walker task, which requires multiple real-valued control signals to be specified per state, given a real-valued observation described by tens of attributes, where the agent must learn to walk.

Lastly chapter 7: Conclusion, wraps up with a summary of the thesis including limitations and directions in which to carry this work in the future.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is a machine learning (ML) paradigm, where a given problem is posed as an interaction between an agent (instance of an ML algorithm) and an environment [63]. An "agent" is taken to imply an instance of an ML algorithm, for example a neural network, decision tree, or a genetic program. An environment is the problem an agent tries to solve, for example an environment could be an Atari game, or a drone in the real world trying to reach a destination.

The term "solve" here can be tricky to define, sometimes environments will have a human pre-defined score that is seen as acceptable for a solution to reach, so if an agent achieves at-least that score then it is considered to have solved the environment. To solve an environment could also mean that the agent performs at a task well enough that an observer (typically human) approves of the results. For example a robot walking up stairs, its footing may be shaky and it may almost fall backwards, but ascending the stairs completely is good enough regardless. Benchmarks are useful simply to compare different algorithms and their obtained scores, though sometimes there are explicitly defined scores that count as a solution which can even be outscored.

The environment first produces an observation, the representation of the current state of the environment (s_t). The agent takes this observation to produce an action (a_t). The environment is then updated based on the action, producing a new observation for the agent to use (s_{t+1}). In addition to each observation, the agent also gets a reward based on the current state of the environment (r_{t+1}). The ultimate goal of the agent is to maximize total cumulative reward or $\sum_{t=1} r_t$. Thus, an episode is the full set of interactions with the environment, from the start state ($t = 0$) to the last ($t = n$), when the environment ends and a terminal reward is received by the agent. Typically the dimensions of the observation and action remain consistent, just the values differ. The so-called reward isn't always positive, it could be a negative

reward, or punishment.

Often in RL results of an agent are averaged over multiple episodes. Overall results are often reported as the mean performance over multiple runs with the same parameterizations (often displaying the standard deviation as well).

2.1.1 Deep Reinforcement Learning

Deep learning (DL) is a very popular method used for function approximation in RL, and ML in general. DL assumes that the RL agent takes the form of a neural network with some form of deep learning architecture (e.g. convolutional and max pooling kernel applied over typically tens of layers under a "bottleneck" configuration in order to facilitate the development of a low dimensional encoding of the original state space [35]). The combination of RL with DL has gained particular popularity on account of several landmark results[1] that might include but are not limited to:

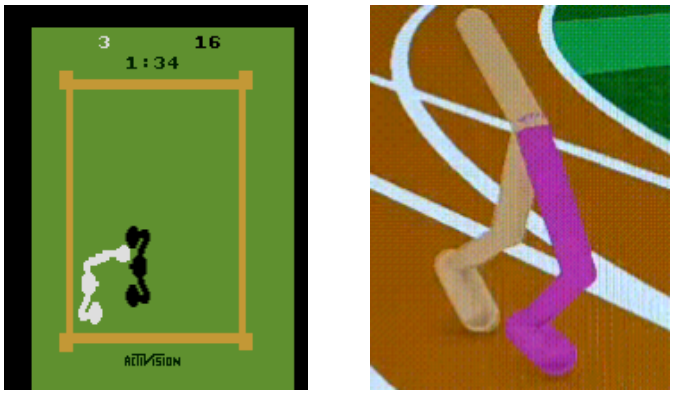
1. DQN as applied to the Arcade Learning Environment (suite of Atari console games [3]) without the use of a prior features [43]. State was described using a down sampled version of frames from the game engine, yet performance typically reached and in some cases bettered that of a human.

2. AlphaGo as applied to the game of Go exceeded the level of play from the world champion [53]. This result was further generalized to AlphaGo Zero which learnt through self play alone [54].

3. AlphaStar in which the earlier results were generalized to the StarCraft video game title [65]. This was significant because StarCraft is an example of a stochastic, partially observable environment, whereas both ALE and Go are examples of tasks with discrete state spaces and complete information.

An example of an Atari environment is explained in Figure 2.1 (a). Other common benchmarks are of the continuous control variety and will be described more in depth section 2.2: Continuous Control. Naturally, results from RL based on DL often establish state of the art performance for the task in question. However, results

[1] Taken from the perspective of the challenge posed by different gaming environments.



(a) Atari Environment

(b) Roboschool Environment

Figure 2.1: Two examples of distinct RL environments. On the left is an Atari environment as can be found in OpenAI Gym, more specifically the Boxing environment. In this environment the observation is every pixel visible on the screen, just as humans see, though in numerical representation. The expected action is a discrete value representing a controller input, for example, up and button press, or no button press and down. The reward comes back as a single value, in this case the player's score (3) minus the opponent's score (16), so -13. On the right is an environment found in OpenAI Roboschool, the RoboschoolWalker2d environment. In this environment the observations would be the angles of joints as well as other typically egocentric physical properties. A valid action would be the force to apply to each joint, for example $(0.2, 0.4, 0.9, 0.5, 1, 0)$, to apply 0.2 units of force the the left hip, 0.4 units to the left knee, etc. The score would then be a combination of the distance walked, movement cost, time steps without falling over, etc.

based on DL also represent a considerable computational overhead, with computation (even post training) requiring support from Graphics Processing Unit hardware (usually, though not entirely necessary). Conversely, solutions identified using genetic programming execute on regular CPU computing platforms. Indeed, the complexity of solutions identified by TPGd for the Atari suite of tasks are multiple orders of magnitude simpler than those identified using DL, with comparable performance in some cases [24, 25, 27, 56, 57, 59]. Moreover, TPGd has even been demonstrated on embedded computing platforms such as the Raspberry Pi [11].

2.2 Continuous Control

Continuous control (CC) environments are environments which require continuous/real valued actions, often multiple actions simultaneously. These are typically simulated/real robotics tasks (e.g. locomotion, or drone flying) or another form of physics based task, though they do not necessarily have to be physically grounded, e.g. [13]. Such environments are distinct from those experienced under tasks from the aforementioned video game tasks in that:

- State information typically takes the form of egocentric (relative) measurements of different components of the system under control. As such there is less emphasis on finding a low dimensional encoding (of a very high dimensional state space), as is often required for benchmarks based on game engines. Instead, the emphasis is more towards feature construction (i.e. discovery of predictive variables).

- Multiple real-valued actions often need to be specified for each state in order to define a complete policy (see Figure 2.1 (b)). This is distinct from many gaming environments where only a limited number of actions (possibly only one) need specifying per action, and the actions are discrete.

- Often more precision or precise control is needed in solutions. In a video-game, erroneously moving to the side could be inconsequential in many cases, whereas doing so in a locomotion tasks could very well lead to an irrecoverable failure (unless counter measures have been previously discovered).

Common environments used for continuous control tasks include the OpenAI gym classic control environments [13], OpenAI gym MuJoCo robotics environments[2], OpenAI Box2D environments [5], and PyBullet[7] gymperium environments.

2.2.1 Reinforcement Learning for Continuous Control

RL is useful for the more complex CC environments which may be unsolvable with closed-form solutions. Though it is still useful to compare RL solutions to optimal closed form solutions on simpler problems for reasons such as comparing learning time

[2] `http://www.mujoco.org`

for a good solution across algorithms, and seeing how close a certain RL algorithm can get to an optimal solution. Also the simple tests can serve as a basic benchmark to know if an RL algorithm idea has potential.

Agents do not necessarily have to solve the environments directly without any additional help. Various methods exist in aiding agents in solving RL problems, especially in more complex CC tasks, due to complexity. One such method is imitation learning, where an existing solution is used to help the agent, for example using real human walking data to help an agent to learn to walk within an anatomically accurate humanoid system [32, 36]. Another method is the use of oscillatory motion models to aid in locomotion (e.g. walking, crawling) tasks, where a sinusoidal wave is used to consistently move a part of the agent's body, and the agent is responsible for configuring the parameters of this model [47, 46, 44, 68]. In this thesis no help is given to the agents, aside from reward signals, and sinusoidal inputs for the biped walker task.

2.3 Evolutionary Algorithms

2.3.1 Overview

An Evolutionary Algorithm (EA) is a method of incrementally improving a group of individuals, often from a random starting point [14]. EAs are analogous to biological evolution, where in a group of individuals, those that are less (more) fit will be less (more) likely to reproduce to create new individuals. These new individuals take on traits of their parent (or parents), along with some random mutations to potentially introduce new behaviours.

EAs can be used along with a variety of different algorithms, such as neural networks or genetic programs (genetic programs will be described in section 2.4: Genetic Programming). To clarify, only one type of algorithm will be used within a given EA run, for example all individuals will be genetic programs. Each instance of an algorithm (e.g. neural network, genetic program) is an individual in an EA, the population being comprised of instances of distinct individuals.

One of the particularly unique properties of EAs as applied to defining solutions to RL problems is that the topology as well as model parameters are both adapted

by the credit assignment process. This is very distinct from other forms of ML in which the user is typically required to a priori define a specific instance of the model[3] after which the ML framework optimizes a fixed set of parameters.

2.3.2 Example

There is no single method of running an EA, they can be ran in a variety of ways with different steps. In this example an EA will be described that is similar to what is used later in this thesis in practice. This EA is being kept abstract, avoiding any specifics of an algorithm used along with it, a biological stand in of a creature and its genes are used. The goal of the population is to avoid predators. An algorithmic description is shown in Algorithm 1.

An EA will typically start with a group of random individuals, of population size N. Imagine a group of N creatures each with a random set of genes, these individuals would each tend to perform different behaviours/routines due to their genetic makeup.

Survival depends on fitness, how well the individuals perform in their environment. In this case the less evasive individuals which are then eaten by the predators have less fitness, all the others which get away without issue will have higher fitness. A portion of the population will be removed, based on fitness (i.e. a breeder formulation in which only the fittest explicitly survive). This can be modelled with a portion called *gap*, which is the portion of individuals that get removed from the population at each generation. Thus, $N \times (1 - gap)$ individuals survive and are able to reproduce, creating new individuals to replace the $N \times gap$ eaten individuals. For convenience, in this example the predators manage to eat exactly $N \times gap$ individuals each generation.

In stochastic environments, as is often the case in RL, individuals are typically evaluated over multiple episodes, this is represented by e, the number of episodes. Fitness will typically be the mean fitness across the e episodes.

In this example the individuals reproduce asexually, meaning that only a single parent is needed to create offspring. So a random surviving individual is selected, gets cloned, and that new clone then has some random mutations done to its genes (based on pre-set evolutionary parameters), and is added to the population. This

[3]For example, under DL the number of layers, kernel type and number of kernels per layer and network topology all need defining a priori.

Algorithm 1 ***EvolutionaryAlgorithm***: Performs evolution of a population in an environment.

INPUT: $\boldsymbol{N}$: Size of the population to maintain. $\boldsymbol{G}$: Number of generations to run for. ***gap***: Portion of population to delete each generation. **e**: Number of episodes to run each agent. ***mutateParams***: Variety of parameters used during mutation (algorithm dependent).

OUTPUT: Varies, could be the set of $1 - gap$ portion individuals representing the survivors at generation G or the single best performing individual.

1. $population = InitializeRandomPopulation(size = N)$
2. For (g In $1..G$)
3. For ($individual$ In $population$)
4. $individual.fitness = Perform(individual, e)$
5. $survivors = GetFittest(from = population, portion = 1 - gap)$
6. $population = survivors$
7. For (i In $1..(N \times gap)$)
8. $child = CloneRandomSurvivor(survivors)$
9. $child.Mutate(mutateParams)$
10. $population.Add(child)$

is repeated, each time with a random surviving individual (with replacement and independent and identically distributed), excluding the newly added individuals, until the current population size is back up to N. Each new individual will likely be quite similar to its parent, but with minor differences, such as a different behavior. It is possible that some changes to the genes of a new individual will have no immediate effect, but may still be useful in its latent state, possibly taking effect after mutations in later generations (provided the individual survives and passes on those genes).

The survivors and new individuals are then tested by the predators and the new survivors repopulate. In implementation, the existing survivors don't have to be tested again, as they already have a fitness, assuming the environment remains static. This process repeats, often for a fixed number of generations, G, with the goal of creating better individuals in later generations.

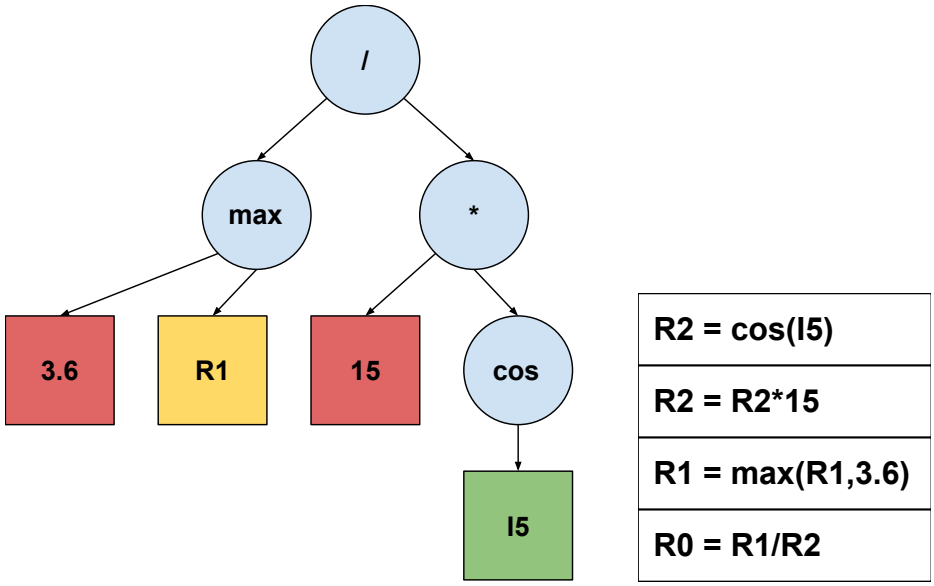
At this point nothing has been said regarding the representation, or how a genotype is mapped to a phenotype. Section 2.4 will detail the specific approach assumed in this thesis (for GP and the GP components of SBB and TPG).

2.4 Genetic Programming

2.4.1 Overview

Genetic programming (GP) is a class of machine learning algorithm which draws parallels to how genes behave in biological evolution and as such is quite similar to the original representation assumed for Genetic Algorithms, i.e. a string of bits [17]. In GP, an individual solution is known as a program, and is comprised of instructions in some representation. Instructions can contain mathematical/computational operations (e.g. cos, $\times$, $\div$, mov) or constants, and operate on some value, potentially from an input state of the current problem. A common formulation of GP is to have programs represented as trees of instructions [33]. Another common formulation which is used in TPG, known as linear GP, is to have programs represented as linear sequences of instructions [4]. These representations are contrasted in Figure 2.2.

A common feature seen in linear GP is the use of registers. Registers are used to store intermediary results and can be used as a form of memory (i.e. retain their value between program execution events). There are different ways to interact



(a) Tree GP (b) Linear GP

Figure 2.2: The equation $max(3.6, R1)/(15 * cos(I5))$ represented in tree based GP (left) and in linear GP (right). $R1$ means whatever is stored in register 1, $I5$ means whatever is in element 5 of the input. Blue circles are operations, and squares are values, red being constants, yellow being register values, and green being input values. In the linear GP example, register 0 stores the outcome.

with the registers, such as having dedicated register read and write operations (as in assembly programming), or by making operations read from either the given input or the registers and immediately writing the result to a register. After a GP individual is finished executing on a given input vector, one or more of the register values can be used as the output, TPG uses this method which will be described in more depth later. Programs have $prog_{regs}$ registers.

To clarify instruction composition, as used by SBB and TPG in addition to linear GP, there are four main parts making up the representation assumed for each instruction: mode, operation, destination, and source. Mode defines whether the instruction reads its source references relative to the application inputs (environment observation) or from its registers (Figure 2.2), it is Boolean valued. The operation selects a single math operation (from a finite set of operations) to perform on the value currently stored in the output register and/or the input. The destination represents which register to store the result of the given instruction in, and can take on a value up to $prog_{regs}$. The source is the index in the observation or register set to pull a value from

(based on mode), this value can range up to $max(Length(observation), prog_{regs})$.

As such linear GP interprets a tuple of integers specifying mode, operation, destination, and source for each instruction. A program consists of L such tuples, where each tuple is "decoded" into its corresponding instruction. The number of tuples need not be the same for each individual, and the variation operators manipulate the integers appearing in the tuples to provide different offspring programs that inherit material from their parent(s). In this work programs to not have an upper limit in size, only an upper limit in initial size (when a program is first created), denoted by $prog_{maxInit}$.

Marking the value in the destination register as x and the value as specified by source and mode as y, the following are all of the operations used in all algorithms in this work (op_set): $x + y$, $x - y$, $x \times y$, $x \div y$ (only if $y \neq 0$), $x = x \times -1$ (only if $x > y$), and $cos(y)$.

2.4.2 Evolution

The process of evolution followed by GP (at-least as used in this thesis) is equivalent to the breeder framework of Algorithm 1. The exact mutation process is illustrated in Algorithm 2. This same mutation process is also used with SBB and TPG programs. In short, a proportion of the instructions have fields from their tuple changed to new legal values. Thus, offspring inherit from their parent(s), but also potentially define new properties. Note however, that not all variation will introduce a corresponding variation in the observed policy of the individual. This is because some fraction of instructions will in some way be "non-functional" or neutral. That is to say, they might not appear in the path of execution (manipulate a value used in a result producing register) or might undo / negate a previous instruction. Such neutral code however, is important from the perspective of the search process, i.e. neutral code could be re-enabled at a later round of variation [4].

2.4.3 Execution

From the context of a learning algorithm making use of a program, the execution of a linear GP individual is quite simple as shown in Algorithm 3. The program executes, and the output value can be obtained from one of its registers (usually the first, for

Algorithm 2 ***MutateGp***: Mutates a genetic program. Function *Flip* is like a coin flip, where the probability provided is the probability of returning true. Passing "random" as an index means any random valid index, using "random1" and "random2" ensures two distinct values. Function *MutateInstruction* mutates a single random part of the instruction (mode, operation, destination, source), changing it to a random valid value.

INPUT: $\boldsymbol{instructions}$: List of instructions that make up a program. $\boldsymbol{inst_{del}}$: Probability of deleting a random instruction. $\boldsymbol{inst_{mut}}$: Probability of mutating a random instruction. $\boldsymbol{inst_{swp}}$: Probability of swapping two random instructions. $\boldsymbol{inst_{add}}$: Probability of adding a random instruction.

OUTPUT: A newly mutated program (list of instructions).

1. $original = Copy(instructions)$
2. While $(instructions != original)$
3. If $(Length(instructions) > 1$ And $Flip(inst_{del}))$
4. $instructions.Delete(index =$ "$random$")
5. If $(Flip(inst_{mut}))$
6. $instructions[$"$random$"$] = MutateInstruction(instructions[$"$random$"$])$
7. If $(Length(instructions) > 1$ And $Flip(inst_{swp}))$
8. $tmp = instructions[$"$random1$"$]$
9. $instructions[$"$random1$"$] = instructions[$"$random2$"$]$
10. $instructions[$"$random2$"$] = tmp$
11. If $(Flip(inst_{add}))$
12. $instructions.add(CreateRandomInstruction(), index =$ "$random$")

a single value output).

Algorithm 3 ***GetActionGp***: Gets an action from a GP individual.

INPUT: ***input***: An input from the environment (e.g. the pixels of an image, or sensor readings).

OUTPUT: An action, in this case a scalar.

1. $action = program.Execute(input)$
2. Return *action*

Note that in this example a real valued number would be returned, whereas in some later examples (SBB and TPG) a discrete value is returned. This is because GP is used later on to obtain a real value to determine which discrete action to return, though the use of GP itself remains the same within the later mentioned algorithms (SBB and TPG).

The process of executing the program, by iterating through all of the instructions is shown in Algorithm 4. In Algorithm 5 it is shown how operations take place, as well as the operation set which is used throughout this thesis.

2.5 Symbiotic Bid Based Genetic Programming

2.5.1 Overview

An extension of GP, known as Symbiotic Bid Based (SBB) GP [38], uses groups of programs called teams to obtain discrete actions, each team being an agent. To obtain an action, a process known as bidding first takes place. Bidding has each program execute, then whichever program produces the highest output (or bid) value is said to have "won" the right to suggest its action. The action is merely a single discrete scalar value (atomic action), where actions are assigned at initialization from the set of task specific atomic actions. Such a process is supported using the concept of coevolution through symbiosis [21]. This implies that there are two populations, a host or team population and a symbiont or program population. Individuals from the team population are merely a set of pointers to programs from the program population.

Algorithm 4 ***program.Execute***: Shows how a program executes. The modes, operations (ops), destinations (dests), sources, and registers are all member variables of the program.

INPUT: ***input***: An input from the environment.

OUTPUT: A numerical value from the first register.

1. For (i In 1..$Length(modes)$)
2. $x = registers[dests[i]]$
3. If ($modes[i] == 0$)
4. $y = registers[sources[i]\%Length(registers)]$
5. Else If ($modes[i] == 1$)
6. $y = input[sources[i]\%Length(input)]$
7. $registers[dests[i]] = ProgramExecuteOperation(x, y, ops[i])$
8. Return $registers[1]$

Algorithm 5 ***ProgramExecuteOperation***: Execute a given operation on the values.

INPUT: $\boldsymbol{x}, \boldsymbol{y}$: Two values to perfom an operation on. ***op***: the operation to perform.

OUTPUT: A scalar, the result of the operation.

1. If $(op == 0)$
2. Return $x + y$
3. Else If $(op == 1)$
4. Return $x - y$
5. Else If $(op == 2)$
6. Return $x \times y$
7. Else If $(op == 3$ And $y! = 0)$
8. Return x/y
9. Else If $(op == 4)$
10. Return $x \times -1$
11. Else If $(op == 5)$
12. Return $cos(y)$
13. Return x

An extension to SBB, known as hierarchical-SBB [28, 12], builds upon SBB by allowing existing teams from previous iterations to become the actions that the programs point to. First the base population is evolved to completion, as in SBB, with only discrete atomic actions. Then the next layer goes through evolution, where actions take the form of a pointer to members from the previous population of teams. This process can repeat any number of times to build multiple layers of hierarchy [55].

It is important that the base population of hierarchical-SBB be sufficiently diverse, as this is the only level of SBB that actually selects atomic actions for the given task. Once past the first layer of SBB there are no new possible atomic interactions with the environment, the new layers learn to recombine individuals from the lower layers together in an attempt to reapply the initial set of policies in new ways.

2.5.2 Evolution

SBB starts evolution by creating teams one at a time, creating random programs to go with each team (limited to a fixed number per team as a pre-established parameter). Programs which are attached to a team are encapsulated in what is called a learner, along with an action. So when a team of programs is mentioned, what is really meant (in terms of implementation) is a team of learners, which each consist of a program and an action (this applies to TPG as well). In effect, the program describes the context under which an action is applied. Each team is given at-least $team_{min}$ learners, each with randomly generated programs and atomic actions (or team actions if a higher layer of hierarchical-SBB). These initial teams will have between $team_{min}$ and $team_{maxInit}$ learners, selected uniformly.

Once the initial population is created, the only sources of variation are due to mutation [39]. Mutation has multiple steps, once a parent is selected the first step is to clone the parent (team). It is that clone that is affected by mutation to become a new distinct individual. Through mutating a team, the team can either have learners deleted, added, or mutated, provided that certain restrictions are followed, namely having at-least two learners with different actions, and not having more learners than a pre-determined maximum limit [39].

After cloning a team, first learner deletion happens. A random learner is deleted

with probability $lrnr_{del}$, if that is successful (and other conditions are met), another random learner in that team learner gets deleted with probability ${lrnr_{del}}^2$, and so on (with shrinking probabilities ${lrnr_{del}}^3$...). Deletion can happen as long as there are more than $max(2, team_{min})$ learners.

Learner addition then follows with the same shrinking probability pattern as learner deletion, with $lrnr_{add}$. Any learner in the entire population (of the current layer) can be added to a team provided that the learner is not already in the team. Learners can be added without restriction to team size unless $team_{max}$ is defined, which sets the maximum number of learners a team can have.

Then learner mutation happens. Each learner is given $lrnr_{mut}$ probability to mutate. When a learner mutates, both the program and the action have a chance of mutating, based on $prog_{mut}$ and act_{mut} respectively, at-least one of these must occur (attempts will be repeated until successful). An action is mutated by swapping out the current action for an eligible one, either a random (different) discrete action if a first layer SBB, or a (different) team from the previous layer for higher layers of SBB.

The learner population is defined as the set of all learners that are referenced by at-least one team. As per team mutation, when a learner is mutated, it first gets cloned, then the new clone has a chance to mutate its program, and its action has a chance to change as well (at-least one of these will happen). A clone of a learner is mutated because the original learner may also be used in any number of other teams. Hence, only the new team should initially have an instance of the mutated learner. Other teams using the original copy of the learner will retain the original variant.

The team population bases its survival on fitness obtained from the environment. The learner population's survival is based on being "useful". A learner is considered useful if it is used by at-least one surviving team. After all the teams undergo fitness evaluation and the weakest *gap* portion are deleted, any learners that are not referenced by any surviving teams are removed from the learner population.

Overall the evolutionary process of SBB follows the breeder framework of Algorithm 1, where the team population is represented by *population*, and the learner population is tracked implicitly (through the team population). So between steps 5 and 6, any unused learners are removed from the learner population.

2.5.3 Execution

SBB teams are synonymous with agents/ policies. They map from state, s_t, to action, $a \in A$, by executing each learner's program, identifying the winning bid and returning the associated action. This is shown in Algorithm 6.

Under hierarchical-SBB an agent represents a policy tree. To select an action, first the top level (agent) team goes through its bidding process to select an action, depending on the current layer this could either be another team or an atomic action. If the selected action is a team, the process repeats, and continues to do so until an atomic action is reached, which is then returned as the current output from the policy [12, 28, 55]. This means that only one team at any level of a policy tree is ever evaluated, potentially resulting in very efficient solutions to reinforcement learning problems [26].

Algorithm 6 ***GetActionHsbb***: Gets an action from a hierarchical-SBB individual. Function *learner.Bid* executes the learner's program and returns the value held in the first register.

INPUT: ***input***: An input from the environment (e.g. the pixels of an image, sensor readings).

OUTPUT: An atomic action (though since the function is recursive there may be an intermediary team return value, though the final returned value is always atomic).

1. $bestBid = -\infty$
2. $bestLearner = null$
3. For ($learner$ In $this.learners$)
4. $\quad bid = learner.Bid(input)$
5. $\quad$ If ($bid > bestBid$)
6. $\quad\quad bestBid = bid$
7. $\quad\quad bestLearner = learner$
8. Return $bestLearner.GetAction(input)$

2.6 Tangled Program Graphs

2.6.1 Overview

TPG is similar to SBB in that it uses a hierarchical structure, making use of different levels of abstraction to the environment. The key difference is that TPG allows programs at any level to point not just to teams (or actions if applicable) of the level below, but to teams of any level and atomic actions at any level of abstraction. So TPG teams can be comprised of a mix of different levels of abstraction to the environment, for example having one program point to an atomic action and another that points to a different team (which itself may point to another team, it does not matter). Whereas SBB forms a tree structure, TPG forms a graph structure with possible cycles, this is where the concept of a "tangled graph" comes from.

Only teams that are not referenced by any learner are considered to be agents, these top level teams are called root teams. Throughout evolution root teams may be subsumed by new teams, losing their root team status, but also if those new teams are removed from the population a former root team may become a root team again (provided that no other learners are referencing it at that time). An example of a TPG instance is shown in Figure 2.3 along with an example action execution.

2.6.2 Evolution

In evolution, TPG starts off like SBB, as a group of N teams, each with two to $team_{maxInit}$ learners. All of these teams are root teams, because they are all disconnected from each other, as all of their learners have solely atomic actions, so there are no learners pointing to any team. All of the learners are given random programs and random atomic actions.

After evaluation in the environment, a proportion, represented by *gap*, of the root team population is deleted. Then, until the team population is back up to N, remaining root teams are selected at random, cloned, mutated, and added to the root team population. Note the slight difference with the use of *gap* and N in TPG. Since there is a root-team (agent) population as a subset of the team population, *gap* is used to delete a portion of the root-team population R, not the team population (N). R is not a fixed value, there will typically be less root-teams than team in a TPG

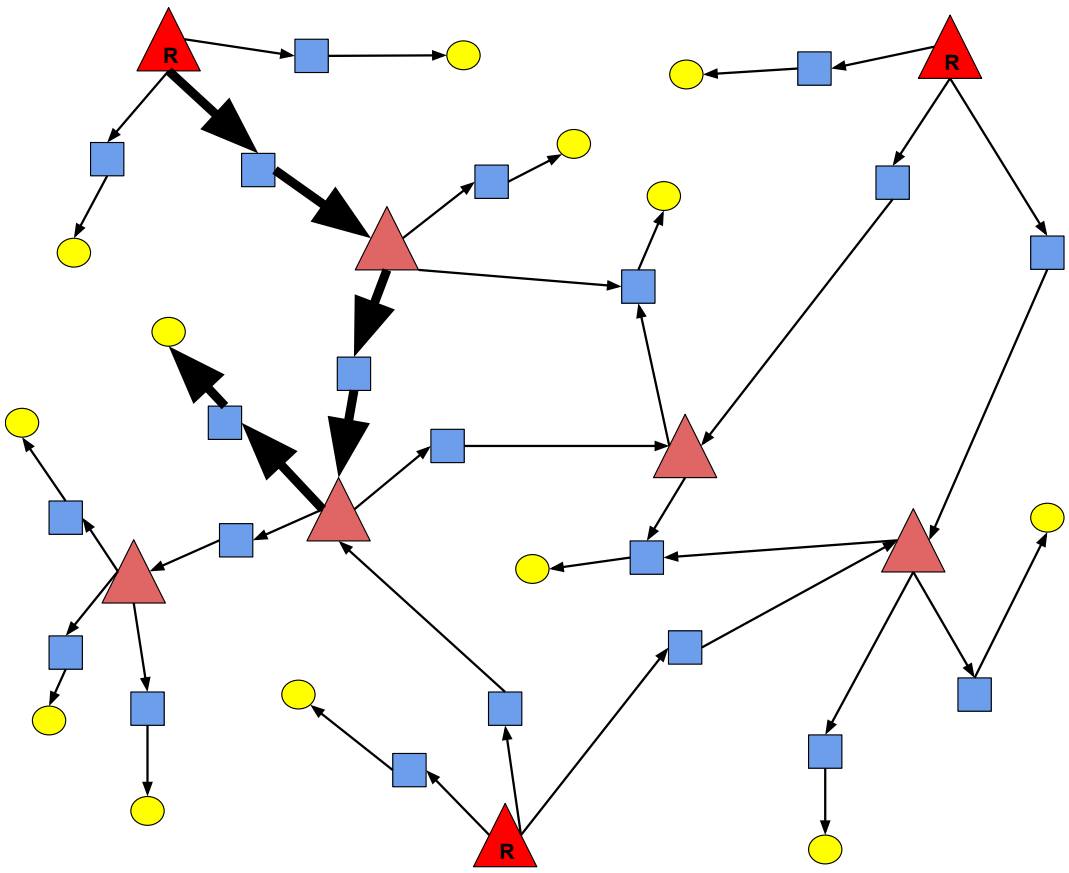


Figure 2.3: An example of the structure of TPG. Here actions are represented as yellow circles, learners are represented as blue squares, and teams are represented as red triangles. The more saturated red triangles with "R"s in them are root teams. The links between components are represented as directional arrows, and the larger bold arrows show a sample execution path to find an action from the top left root team. Note that this path does not show all of the teams and learners that would have been visited, just the final path to the selected action. All of a teams learners are visited when a team is visited (except for learners that point to teams that have already been visited in the current environment step).

population, and the value will fluctuate throughout evolution.

The mutation process is slightly different than in SBB. Being composed as a graph rather than a tree (as in SBB) could possibly introduce infinite loops during execution. This is overcome by not allowing the same team to be visited more than once in each environment interaction, as well as by ensuring that each team has at-least one learner with an atomic action. When adding learners during team mutation, an additional constraint is that an added learner must not have its action be a team action that refers to the team itself (the team being currently mutated).

Additional changes are made to the learner mutation process. The variable act_{atom} is added, representing the probability that when a learner's action mutates, it becomes an atomic action (vs a team action). If the action mutates into a team action (whether or not it already was a team action), the new team cannot be the team that the learner being mutated belongs to.

2.6.3 Execution

The process of action selection / execution in TPG is almost identical to SBB. Root teams are the teams that will interact with the environment, all other teams are involved as sub-processes of the root teams, execution starts with the root team.

The main difference with TPG is that there needs to be a memory of which teams have been visited already for a given input, as seen in Algorithm 7. When each node (team) of the graph is visited during a given environment interaction, it is added to a list of visited nodes. Before taking any learners' team action, the team is compared to the visited node list, if it is already in the list then the next best learner's action is taken (this process repeats until there is a valid action). This along with ensuring that each team has at-least one atomic action does away with any possibility of an infinite loop, ensuring that an action will always be returned in a timely manner. An example of execution is shown in Figure 2.3.

Algorithm 7 ***GetActionTpg***: Gets an action from a TPG individual.

INPUT: ***input***: An input from the environment (e.g. the pixels of an image, sensor readings). ***visited***: a list of already visited teams to not repeat.

OUTPUT: An action, in this case a scalar (though since the function is recursive there may be an intermediary team return value, though the final returned value is always a scalar).

1. $visited.Add(this)$
2. $bestBid = -\infty$
3. $bestLearner = null$
4. For ($learner$ In $this.learners$ If $learner.team$ Not In $visited$)
5. $\quad bid = learner.Bid(input)$
6. $\quad$ If ($bid > bestBid$)
7. $\quad\quad bestBid = bid$
8. $\quad\quad bestLearner = learner$
9. Return $bestLearner.GetAction(input, visited)$

Chapter 3

Real Valued Tangled Program Graphs

3.1 Overview

The central contribution of this thesis is the implementation and evaluation of a method of creating real valued actions in TPG. Unlike TPGd in which an action is assumed to be a discrete scalar value, TPGr generalizes actions to "functions". Thus, when a learner produces a winning bid, the corresponding action program would execute, producing a vector of actions. The set of states for which the learner wins is now associated with a function rather than a single scalar value (discrete or real-valued). That said, the only difference between TPGd and TPGr is the use of a program to produce actions. However, this has the potential to change the functional properties of TPG as well as opening up a range of different applications that previously were out of scope, i.e. multiple actions per state and real-valued actions.

The major motivation of TPGr is to make TPG more competitive in CC tasks, by not relying on discretization or re-using the output of the bidding programs. Though TPGr can also be used in tasks with only discrete actions, such as in the Atari domain, where action programs outputs are floored and modded to an appropriate value. This potentially allows for smaller less complex teams to explore full action spaces. A slightly different approach is explored in the ViZDoom tasks explained later.

3.2 Implementation

TPGr assumes programs for action creation/generation, or a function. Where before each learner consisted of a program, p, for identifying context (the bidding process) and a single action defined as a fixed discrete value[1], now learners define task specific

[1]Selected from a set of task specific atomic actions ($a \in A$).

actions as two programs, one for bidding (p_b) and one for action generation (p_a). In this work, programs assume the linear GP representation [4], thus defined in terms of an instruction set operating on registers (§2.4.1). This now means that action programs can produce a vector of actions relative to the subset of states that the bidding program is the winner.

Naturally, the number of actions required by a task domain sets a minimum bound for the number of registers per action program, $actProg_{regs} \leq |A|$. Just like bidding programs, action programs also have a maximum initial size, $actProg_{maxInit}$, with no set upper bound to growth.

In addition to action programs, TPGr still provides pointers to other teams. Thus, there are two types of actions that a learner can support: pointers to other teams and action programs. Each learner may only have one type of action, however, as TPG action mutation can flip between the types. This is exactly the same as previously described for TPG, except for the atomic actions now being represented by programs.

In TPGr when mutation takes place for a learner's action object, there is a 50/50 chance that the action program will be modified, if applicable. Along with a potential action program mutation, action mutation takes place normally where the action can either become an atomic action or a team action.

It should also be mentioned that this real valued paradigm can be applied to SBB as well, given that SBB is essentially a form of TPG[2]. Such a formulation of SBB will be referred to as SBBr, and is the only form of SBB used in this thesis. No such distinction is needed for GP, given that GP is already real action generator (as it is used for real action generation in SBBr and TPGr).

[2]SBB = TPG limited to single teams of programs, so no graphs of teams.

Chapter 4

Diversity Maintenance

4.1 Overview

Machine learning algorithms in general have to balance exploration and exploitation during credit assignment. Exploitation in evolutionary computation implies that the population is increasingly taken over by offspring from the better performing parents. This might be appropriate when there is a good chance that the training scenarios are "sufficiently" representative of the overall task.[1] However, under tasks that are in some way partially observable or stochastic it is likely that performance evaluation will be "noisy" or only capable of providing partial information. The resulting "partial" fitness evaluation may then be significantly different from the actual level of performance, or misrepresent the potential of a given pathway of evolution. Indeed various pathologies can result from such partial information, e.g. cycling, forgetting, disengagement [67].

There are a variety of ways in which diversity can be increased/maintained. Some specific examples might take the form of:

- Multiple objectives [8, 6]: implies that the objective of the task is actually expressed using multiple objectives. Such objectives might be in tension with each other to some degree[2] or be satisfied sequentially, i.e. solving one objective representing a pre-requisite to solving another. Several overheads may appear when applying multi-objective formulations the most significant of which being that multiple forms of fitness evaluation are necessary (one for each objective).

- Niching methods [9]: recognize that the same level of performance might be associated with different 'modes' of the task. Niching therefore attempts to

[1] By "sufficient" we do not imply exhaustive, only that the sampling of training instances is selected to cover the envisaged target application scenarios.

[2] For example, precision and recall.

distinguish between different modes using a distance metric defined relative to the representation space. However, distance metrics are sensitive to the definition of suitable thresholds to characterize what is sufficiently close enough to something else (i.e. how close / distant does something have to be before it is a member/ not a member of a niche?). Moreover, defining distances for representations based on collections of variable length programs (as per TPG) is not straightforward.

- Incremental evolution [18]/ layered learning [62]/ curriculum learning [45]: means that a set of source tasks are first designed. Solving each source task either sequentially and/or independently enables the overall target task to be solved. To varying degrees, it might be possible to actually learn which source tasks should be solved and in what order, i.e. competitive coevolution [55]. Aside from having to a priori design a "good" set of source tasks, such methods assume that there is enough control over the task domain to facilitate initialization from arbitrary start configurations.

- Novelty metrics [37]: imply that rather than attempting to learn how to solve a task using a task specific performance objective, e.g. maximize the distance traveled in a locomotion task, evolution is rewarded for discovering a diverse range of policies. To do so, appropriate "novelty" metrics need to be defined, such as finding sequences of actions or state-actions that are distinct from those discovered by other individuals from the population [26]. One source of difficulty can appear when the task domain has stochastic properties (such as the use of random start conditions) that then make it difficult to establish whether the source of "novelty" was due to the start condition, non-stationary properties of the environment or the agent under evaluation.

- Age layered Population Structure (ALPS) [22]: stratifies the population based on age in an attempt to make the competition between individuals with different age bands "fair". The lowest band consists of a fixed number of individuals that are randomly seeded at each generation. Individuals comprising the content of older bands must be fit in order to remain competitive within the band. Decisions regarding the number of bands could be mitigated by assuming a

Pareto multi-objective formulation [50].

In this thesis the biped walker task will maintain diversity by occasionally introducing new SBBr individuals as potential new actions to TPGr (which later on may become full teams), described in section 4.2. Such a scheme therefore attempts to continuously introduce new material into TPGr, i.e. a little like ALPS, but without the age based matching criteria. The motivation for doing so is to let TPGr plateau before introducing new material, with the goal of simulating new directions for development. TPGr can potentially make use of this material by introducing appropriate team references to the root teams. The biped walker represents an environment that can only be solved using multiple real-valued actions per state.

Conversely, the ViZDoom agent control task will make use of a curriculum of source tasks in order to maintain diversity (section 4.3). The ultimate goal is to discover a single agent that performs all source tasks. The ViZDoom task assumes discrete actions, but may support a variable number of discrete actions per state.

In addition to this, across all algorithms, a process called rampancy is used, described in section 4.4. The underlying motivation for rampancy is to introduce multiple modifications simultaneously across the multiple levels of TPG.

4.2 Intermittent SBBr Populations

A type of run used in this thesis for the biped walker uses a combination of TPGr and SBBr, referred to as TPGr+SBBr. Essentially a single TPGr population is maintained through the whole process, while periodically new SBBr runs will be started to integrate into the TPGr population, see Algorithm 8. This is similar to the concept of restarts with a fixed "momentum", as mentioned in [60].

From the context of the aforementioned diversity mechanisms such a process is a little like ALPS. However, TPGr enables us to directly subsume champions from the independent SBBr runs on a continuous basis, hence we do not need to protect individuals though the concept of age, hoping that crossover is sufficient for combining the best of old and new individuals. Instead, TPGr can directly index the new SBBr individuals as they are continuously introduced into the pool of eligible (team) actions.

These runs start out with an SBBr population that evolves for a fixed number of generations (G_{sbb}), and stops earlier if no improvements are made in best fitness for

Algorithm 8 ***RunTpgSbbPopulation***: Performs evolution of a TPGr population interleaved with SBBr populations. *RunSbbSubpopulation* receives a variety of parameters as described in Algorithm 9. *RepopulateTpg* goes through the whole selection and mutation process of TPG for the given generation.

INPUT: $\boldsymbol{g_{fail}}$: The maximum number of generations an SBBr or TPGr population can go without making improvements. Plus a variety of additional evolutionary parameters (e.g. $G, N, gap, e, mutateParams$).

OUTPUT: Nothing in this example, though a set of top performing individuals could be returned/saved.

1. $maxFitness = -\infty$
2. $noImprovements = \infty$
3. $tpgPop = InitializeTpgPopulation(size = N)$
4. For (g In $1..G$)
5. If ($noImprovements > g_{fail}$)
6. $sbbChampions = RunSbbSubpopulation(...)$
7. For ($individual$ In $tpgPop$)
8. $individual.fitness = Perform(individual, e)$
9. $tpgPop = RepopulateTpg(mutateParams, sbbChampions)$
10. If ($bestFitness(tpgPop) <= maxFitness$)
11. $noImprovements+ = 1$
12. Else
13. $maxFitness = bestFitness(tpgPop)$
14. $noImprovements = 0$

a consecutive number of generations (g_{fail}). At the end of such a SBBr run, a set number of top individuals from the population are saved for later use (n_{sbb}), these will be called the SBBr champions, and are swapped out after every SBBr run (Algorithm 9), and are used as potential actions in the TPGr population.

After the first SBBr champions are selected, TPGr commences evolution. At each generation the TPGr individuals are evaluated on the environment as usual, but when variation occurs, the SBBr champions are also used as potential actions that can be mutated into the TPGr population (thus making them TPGr teams).

As opposed to the SBBr populations, the TPGr population can evolve without a fixed generation limit (aside from the ultimate end generation of the run, G), provided that max fitness improvements does not stagnate for more than g_{fail} generations just like in the SBBr populations. In such a case another SBBr subpopulation will be started, and cycle repeats (until G generations).

The motivation for adopting such an approach is to attempt to "kick start" new search directions when TPGr evolution plateaus. Rather than target current content (that presumably does something well), we let SBBr (aka single teams) develop a new set of candidate teams.

It is hypothesised that this method of evolution will help the population improve beyond a local optima. If a population stagnates at a given part of the environment, introducing a new population could help to break up that stagnation. These SBBr populations can be especially directed to focus on the given local minima, by starting the SBBr individuals at a state that is near where TPG champions have failed (as is the case in the biped walker task).

Intermittent subpopulation were described here in terms of the real valued formulations of TPG and SBB. Though it could apply to the discrete formulations as well, and any other learning algorithm (or multiple, provided they are compatible as SBB and TPG are).

4.3 Curriculum learning

Under the ViZDoom task domain our goal is to learn policies that have the potential to perform well at multiple tasks. Thus, we are able to chose the task experienced by an agent. Specifically, at generation t `mod` τ a task is chosen uniformly without

Algorithm 9 ***RunSbbSubpopulation***: Performs evolution of a an SBBr subpopulation for use in TPGr.

INPUT: $\boldsymbol{G_{sbb}}$: The max number of generations an SBBr population can run for. $\boldsymbol{g_{fail}}$: The maximum number of generations an SBBr can go without making improvements. $\boldsymbol{n_{sbb}}$: Number of top individuals to return from the SBBr population to TPGr. Plus a variety of additional evolutionary parameters (e.g. $N, gap, e, mutateParams$).

OUTPUT: Nothing in this example, though a set of top performing individuals could be returned/saved.

1. $maxFitness = -\infty$
2. $noImprovements = 0$
3. $sbbPop = InitializeSbbPopulation(size = N)$
4. For (g In $1..G_{sbb}$)
5. If ($noImprovements > g_{fail}$)
6. Break
7. For ($individual$ In $sbbPop$)
8. $individual.fitness = Perform(individual, e)$
9. $sbbPop = RepopulateSbb(mutateParams)$
10. If ($bestFitness(sbbPop) <= maxFitness$)
11. $noImprovements+ = 1$
12. Else
13. $maxFitness = bestFitness(sbbPop)$
14. $noImprovements = 0$
15. Return $GetFittest(from = sbbPop, amount = n_{sbb})$

replacement from a set of T tasks. Thus, over a sequence of τ generations the same task is experienced by all agents before a different task is selected.

Fitness is accumulated per task, or

$$f(i,S) = \frac{1}{\tau}\sum_{k=0}^{\tau-1} gs_k(i,S) \tag{4.1}$$

where $gs_k(i,S)$ is the game score returned by the game engine for agent i in task S. A total of τ encounters between agent and the same task occur due to the stochastic initialization of a task.

In order to encourage agents to generalize to more than one task, fitness is actually formulated with respect to the previous three tasks (chosen stochastically), the curricula. However, each task's game score are over different ranges. Thus, a task's score is re-scaled relative to the best agent's score on that task encounter:

$$F(i,S) = \frac{f(i,S)}{\max_{j\in N} f(j,S)} \tag{4.2}$$

where N are the set of eligible agents (e.g. root teams in TPG, all the individuals from the team population in SBB).

Performance of an agent is now defined as that over the last R ($R = 3$ in this case) tasks encountered:

$$F(i) = \sum_{j=1}^{R} F(i, S-j+1) \tag{4.3}$$

In short, tasks are selected randomly from the curricula of five tasks without replacement and fitness defined over the last three tasks encountered. Once all the curricula has been encountered, the set of tasks appearing in the curricula is reset. The underlying assumption is that the population will act as a repository of multiple agent behaviours. Eventually, agents will emerge that are able to demonstrate skills that span multiple tasks. Such an approach was previously demonstrated to be effective for evolving solutions for up to 8 tasks [56, 58]. Here the focus is on whether action programs are able to accelerate and/or develop stronger multi-task agents.

4.4 Rampancy

Rampancy is the process of repeated mutation, and can be parameterized as a 3-tuple [1]. For a rampancy tuple $(freq, r_{min}, r_{max})$, $freq$ is how frequently to perform rampancy in generations, so with $freq = 5$, rampancy will take place every 5 generations. r_{min} and r_{max} define the range in which a random number of rampancy iterations will take place. If r_{min} and r_{max} are equal, then rampancy occurs for exactly that many iterations.

An iteration of rampancy is just a regular mutation. For GP, rampancy takes place at the program level. For SBB and TPG, rampancy takes place at the team level.

The primary goal of rampancy is speed up the process of evolution, creating a less gradual search (intermittently, depending on $freq$).

Chapter 5

ViZDoom Experiments

5.1 Overview

The ViZDoom platform consists of multiple task environments, each with similar base mechanics, but different goals [31]. However, all environments share the same visual state space, a $160 \times 120 \times 3$ RGB image, each pixel's colour being described using 8 bits. The three RGB colour values are concatenated into a single 24-bit representation in order to reduce the state representation to a single matrix of 160×120 cells [56]. A learning agent has to be able to infer what the task is from the visual information alone, and then develop a strategy for maximizing reward under that task.

Five task environments are used in this research: Basic, Defend the Center, Defend the Line, Health Gathering, and Take Cover.[1] The action space and reward differs among the different environments, specified as follows:

- Basic: spawns the agent in the center of the longer side of a rectangular room. A monster is spawned on the opposite wall at a random location. The agent may only turn left/right or shoot. The rewards are 101 for shooting the monster, -5 for a missed shot, and -1 for each time step and there are a maximum of 300 frames within an episode. The "high score" on this task is therefore 100 (under a very specific agent-monster initialization).

- Defend the Center: the agent must survive in a room with 5 re-spawnable monsters. The agent can only turn left or right and shoot, and the number of shots is limited. Moreover, the agent is only spawned in the centre of the room and, due to the limited actions, can only rotate. A reward of 1 is given for each monster killed, and the episode ends when the agent dies (which will happen due to the limited number of shots).

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

- Defend the Line: is similar to Defend the Center, except the monsters come back stronger each re-spawn. Moreover, the room is now a rectangle, with the agent spawned in the centre of one of the longer walls. Monsters are spawned randomly on the opposing wall, but there can only be three at a time. Again, the limited number of shots implies that episodes end with the agent losing its life.

- Health Gathering: rewards the agent for collecting health packs in order to counter the negative effect of an acidic floor. There are no monsters. The agent can only move forward and rotate left or right and the environment is limited to a single rectangular room. The health packs re-spawn randomly, so the agent has to continuously investigate the room in order to survive. A reward of 1 is given for each frame survived, up to a maximum of 2,100 frames.

- Take Cover: monsters are spawned repeatedly and can shoot projectiles at the player. The agent can only move left and right, i.e. no ability to shoot. As such the agent has to learn to "dodge" the incoming projectiles. A reward of 1 is given for each frame and more monsters appear as the episode length increases, implying that at some point there will be too many projectiles for the agent to avoid being hit.

The ViZDoom task will be used to compare TPGd to TPGr. In order to render action programs as "discrete" (as opposed to real-valued) each register associated with an action is subject to a threshold. If the register value is greater than zero, then the action is deployed, otherwise it is not. This potentially means that multiple actions can be deployed under any given state (allowed under the ViZDoom game engine). In addition, the variable number of actions per task is accounted for by supporting the union of all actions, but enforcing a mask relative to the task.

Post training, we will assess the ability of agents to both specialize (return the high-score specific tasks) as well as generalize (identify a single policy that is competitive under multiple tasks).

Ultimately this environment was selected due to it being a sufficiently complex task, in having a large observation space to index, and requiring emergence of multi-task behaviour in order to perform well. It is also well suited to using TPGr for

discrete action generation given that multiple actions could be selected at a time, a more open ended task than selecting a single action from a program.

5.2 Prior Results

ViZDoom represents a widely employed benchmark for visual reinforcement learning, in part on account of the challenging nature of the task (partially observable environment, high-dimensionality, 3-D representation, multiple actions per state) and efficient implementation [31]. That said, benchmarking generally takes one of two forms: evaluation on individual source tasks, or evaluation on some form of 'deathmatch'.

Thus, many reinforcement learning frameworks have been benchmarked on single source tasks, beginning with the original demonstration of the ViZDoom game engine [31] and continuing to the present [2, 64, 48]. The common theme being that they all assume some form of deep neural network architecture for reinforcement learning. Conversely, when training agents to participate in the 'deathmatch', deep learning approaches have either assumed: a curricula of hand designed progressions for deathmatches [71, 72], independently trained navigation and specialist fighting networks [72], or assumed a self play framework [48].

Outside of deep learning, the only previous research has been that involving TPGd. Initially, TPGd agents were demonstrated to be able to play multiple subtasks [56] and then, with the addition of indexed memory, agents were demonstrated in 'deathmatches' [57].

In this work, for computational practicality, we focus on learning 5 of the 8 source tasks simultaneously, i.e. an undertaking sufficiently difficult to illustrate the merit or otherwise of assuming action programs in TPGr.

5.3 Experiment Parameterizations

As noted above, the ViZDoom domain will be used to compare TPGd to TPGr. Both configurations assume rampant mutation (section 4.4).

The TPG approaches are parameterized as shown in Table 5.1. In short, the

Table 5.1: Parameters for the TPG ViZDoom runs. $*$ In the version of TPG used in these runs, learner mutation is attempted as each learner is added from a parent to the new team and program mutation is also attempted but not guaranteed.

Parameter	**TPGd**	**TPGr**
N	120	
G	9,000	
gap	0.5	
e	5	
$rampancy$	(1,5,5)	
$team_{min}$	12	
$team_{max}$	12	
$team_{maxInit}$	12	
$lrnr_{del}$	0.7	
$lrnr_{add}$	0.7	
$lrnr_{mut}$	?*	
$prog_{mut}$	?*	
act_{mut}	0.2	
act_{atom}	0.5	
$prog_{maxInit}$	128	
$prog_{regs}$	8	
$actProg_{maxInit}$	$\emptyset$	128
$actProg_{regs}$	$\emptyset$	7
$inst_{del}$	0.5	
$inst_{add}$	0.5	
$inst_{swp}$	1.0	
$inst_{mut}$	1.0	

only difference between TPGd and TPGr are the parameters for the action programs. Naturally, there is no concept of action program registers for TPGd, hence the null parameterization. In total 9,000 generations (G) are performed, with the task stochastically re-sampled every $e = 5$ generations. There are $N = 120$ root teams, with $gap = 50\%$ replaced per generation.

5.4 Memory

The VizDoom task is both high-dimensional (state space with over 19,000 inputs, i.e pixels) and partially observable. This means that an internal model of state also has to be learnt. To do so, this work adopts the probabilistic indexed memory model formulated in [57, 58]. There are two components to this framework: 1) a single

instance of indexed memory that the entire population 'share' and 2) a probabilistic write operation.

The shared indexed memory approach implies that the contents of indexed memory is only ever a 'cold start' at initialization. Thereafter each agent inherits the state of indexed memory as left by the agent that was previously evaluated. One implication of this is that indexed memory comes to represent a shared resource for communication between programs comprising a learner as well as different TPG agents. Any agent that 'pollutes' the contents of indexed memory therefore has an impact on the entire population. Previous research has shown that such a shared memory model is actually effective at establishing a common view of state [61].

Two additional instructions are assumed in order to enable programs to read or write to indexed memory. Classically, both operations require an address to be specified. However, the probabilistic indexed model of Smith and Heywood adopts a different approach [57, 58]. Instead of specifying a write address, the write operation (`write(R)`) assumes that indexed memory is organized into L columns of `MaxReg` cells, where `MaxReg` is the number of registers that any program may manipulate (see the linear GP representation of §2.4.1). The write instruction, when encountered, writes the contents of the `MaxReg` register values to each column of indexed memory such that the probability of a write follows the distribution [57, 58],

$$P_{write}\left(\frac{L}{2} \pm c\right) = \alpha - (\beta \times \mathrm{c})^2 \tag{5.1}$$

where c is a column index and $\alpha = 0.25$ and $\beta = 0.01$ control the height and width of the shoulders of the probability distribution.

This means that columns near 1 or L are less likely to be written to, whereas columns towards $\frac{l}{2}$ are more likely to be written to. This mimics a form of long (respectively short) term memory. The read operation take the form of `R[i] = read(k)` in which i is the register index and k is the memory index. Over time, programs appear to evolve specialist programs for writing data to indexed memory, while most programs read. Previous research has demonstrated that the probabilistic index model produces agents that discover how to navigate under the VizDoom and Dota 2 environments [57, 58] and perform considerably better than reactive agents

under the take cover task [30]. Note that the probabilistic indexed memory model is only used in the ViZDoom benchmark.

5.5 Results

5.5.1 Training

Training performance for each task is summarized by Figures 5.1, 5.2, 5.3, 5.4 and 5.5. Thus, although the population only experiences a single task per generation, we plot the population's previously recorded performance for each task. Naturally, this may introduce considerable variation in the performance of tasks over generations, i.e. improvements in one task, might result in reductions to others.

In each case, the fitness curves reflect average population fitness (solid line) and champion individual in the population (dashed line). The extent of the variance of the champion is also summarized by the shaded area (max to min of champion on the 10 evaluations performed per generation, where the task environment is stochastically initialized). The green (blue) curves represent TPGd (TPGr) respectively.

The following observations are made with respect to each task:

- Basic (Figure 5.1): both TPG formulations appear to solve this task (max fitness for this task is 100). However, the Action Programs do so while maintaining a higher average population fitness and the champion agent observes much less inter-generational variation (compare green to blue shaded regions).

- Defend the Centre (Figure 5.2): agents using Action Programs appear to have a considerable advantage on this task, with average population fitness of Action Programs matching the performance of the champion agent from Discrete Actions.

- Defend the Line (Figure 5.3): demonstrates a clear preference for TPGd throughout the run.

- Health Gathering (Figure 5.4): only TPGr is able to consistently achieve maximum fitness on this task (2100). Indeed, fitness for champions with Discrete Actions is significantly lower than the average population fitness with Action Programs.

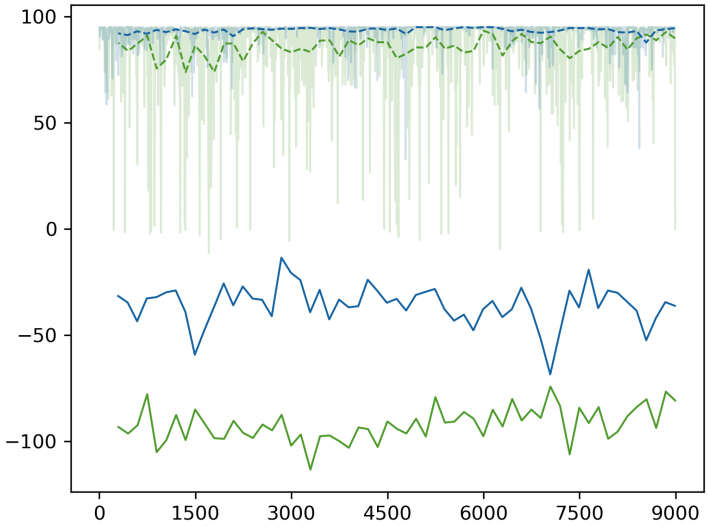


Figure 5.1: Fitness curves during training for Basic task. Blue (Green) curves indicate TPGr (TPGd) respectively. Average population performance is the solid line. Dashed lines are the average task performance of the champion with shading indicating the corresponding spread (max. to min.)

- Take Cover (Figure 5.5): champion fitness was very much the same for both TPG configurations, but the average fitness of TPGr was consistently higher than for TPGd.

Naturally, these curves are only able to reflect the performance of TPG agents under each independent task, i.e. the performance ceiling for each task / approach to action selection. The following post training assessment will assess the capacity of each configuration to identify agents that generalize across multiple tasks.

5.5.2 Generalization

Post training performance is assessed from the perspective of an agent to perform multiple tasks simultaneously. With that in mind, the top 20 agents are identified (from training) and performance averaged over 50 initializations of each task. Generalization will be assessed by evaluating performance over all combinations of 3, 4 and 5 tasks. However, to do so, we first need to normalize performance from individual game titles in such a way that they can be additively combined. For this purpose, use is made of the linear re-scaling normalization as often assumed for mapping features over a fixed range. In this context the worst and best performance on a title (across

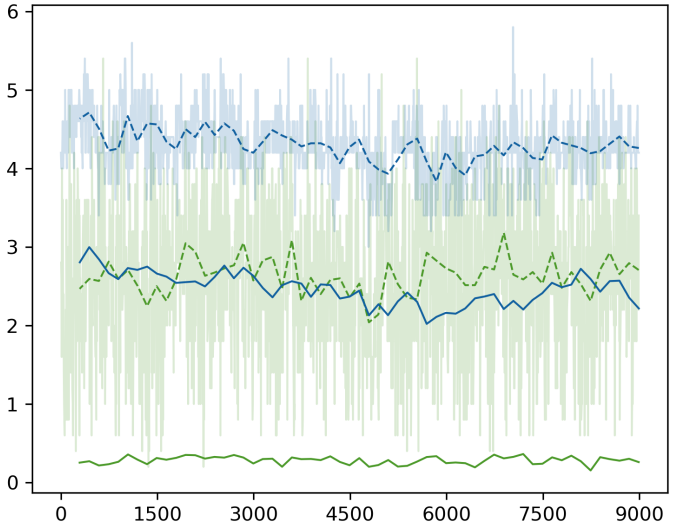


Figure 5.2: Fitness curves during training for Defend the Center task. Blue (Green) curves indicate TPGr (TPGd) respectively. Average population performance is the solid line. Dashed lines are the average task performance of the champion with shading indicating the corresponding spread (max. to min.)

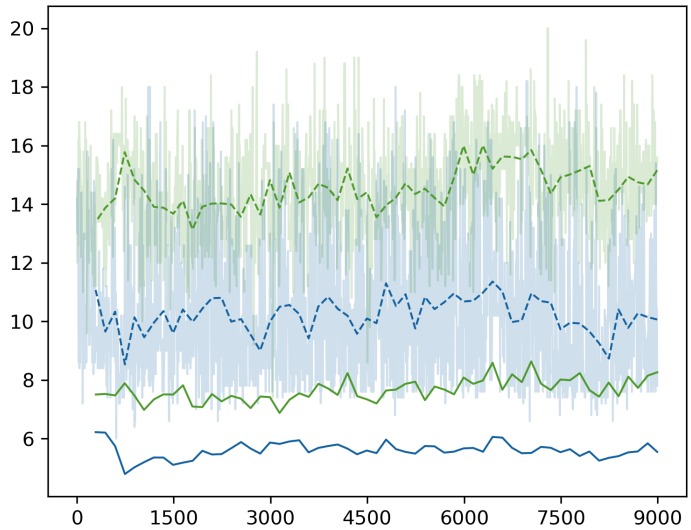


Figure 5.3: Fitness curves during training for Defend the Line task. Blue (Green) curves indicate TPGr (TPGd) respectively. Average population performance is the solid line. Dashed lines are the average task performance of the champion with shading indicating the corresponding spread (max. to min.)

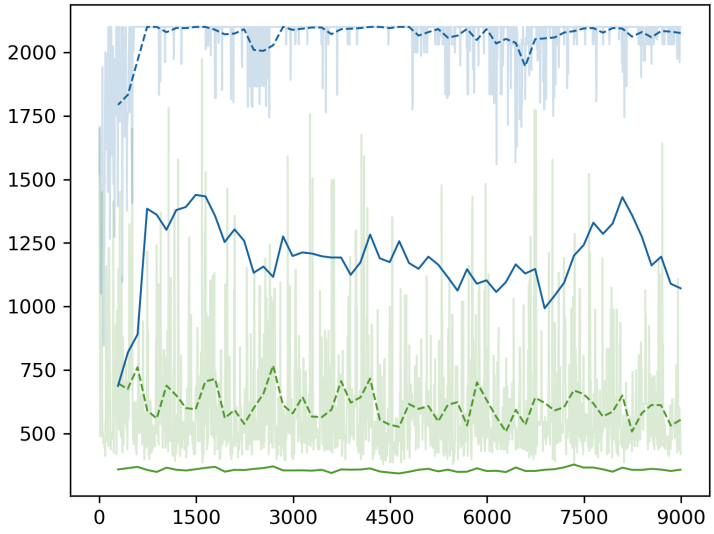


Figure 5.4: Fitness curves during training for Health Gathering task. Blue (Green) curves indicate TPGr (TPGd) respectively. Average population performance is the solid line. Dashed lines are the average task performance of the champion with shading indicating the corresponding spread (max. to min.)

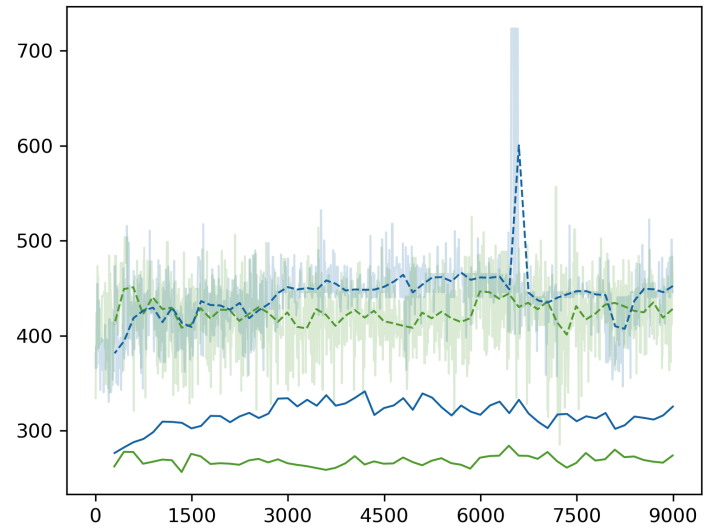


Figure 5.5: Fitness curves during training for Take Cover task. Blue (Green) curves indicate TPGr (TPGd) respectively. Average population performance is the solid line. Dashed lines are the average task performance of the champion with shading indicating the corresponding spread (max. to min.)

any agent) is used to map the average generalization performance of an agent to the interval $[0, 100]$. Thus, the normalized score, $\bar{s}$, of agent i on task g is defined as

$$\bar{s}_i(g) = 100 \times \frac{s_i(g) - \min_{k \in \mathcal{S}}(s_k(g))}{\max_{k \in \mathcal{S}}(s_k(g)) - \min_{k \in \mathcal{S}}(s_k(g))} \tag{5.2}$$

where $s_i(g)$ is the original score of agent i on task g (averaged over 50 initializations). $\mathcal{S}$ is the set of 20 agents under evaluation.

The generalization performance of agent i over a set of tasks ($g \in \langle$*b, bg, btl, dtc, tc*$\rangle$) is now the sum over the combination of tasks appearing in the assessment, or

$$G(i) = \sum_{g \in \mathcal{G}} \bar{s}_i(g) \tag{5.3}$$

where $\mathcal{G}$ is the particular combination of tasks included in the evaluation.

As there are 5 tasks, there are a total of 10 combinations of 3 tasks, 5 combinations of 4 tasks and one combination of 5. Table 5.2 details the performance of TPGd and TPGr agents over the combinations of tasks for the assessment of generalization. A ranked difference is then estimated as the basis for applying the Wilcoxon Signed-Ranks test for significance testing [10].

It is now apparent that TPGd never wins a generalization test for any combination of 3 to 5 task (significant at the 95th percentile). In summary, TPGd provides solutions that appear to be very specific to the "Defend the Line" task, but there is no evidence of generalization beyond this. On the other hand TPGr provides agents that are able to discount their lack of ability on Defend the Line with stronger performance on all other tasks.

5.5.3 Complexity

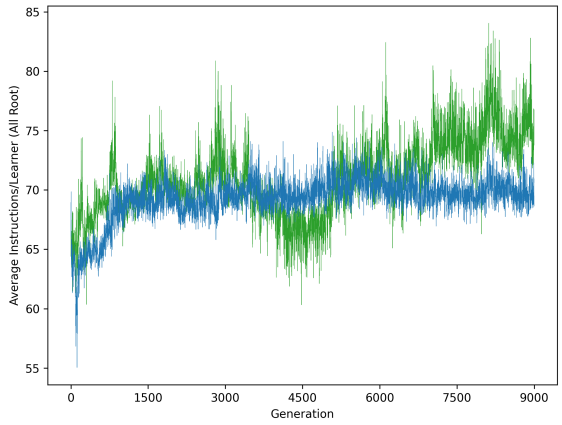
TPG incrementally constructs graphs of teams / programs during evolution. As such, the dynamic properties of this process can be visualized (Figure 5.6). It is apparent that TPGr and TPGd undergo very little variation in the number of learners per team. From the perspective of the number of instructions per learner (team), TPGr appears to converge on a preferred complement within the first 1,000 generations and is then relatively stable. Conversely, TPGd appears to have an underlying growth trend over the generations, resulting in solutions that are generally more complex

Table 5.2: TPGr and TPGd generalization performance over combinations of 3, 4 and 5 task combinations. ViZDoom tasks are identified as “b” for basic; “hg” for health gathering, “dtl” for defend-the-line, “dtc” for defend-the-center, “tc” for take cover. Rank is determined by how well TPGr performs over TPGd, from the least (rank 1) to the most (rank 16). This ranking system is used for the Wilcoxon Signed-Ranks test.

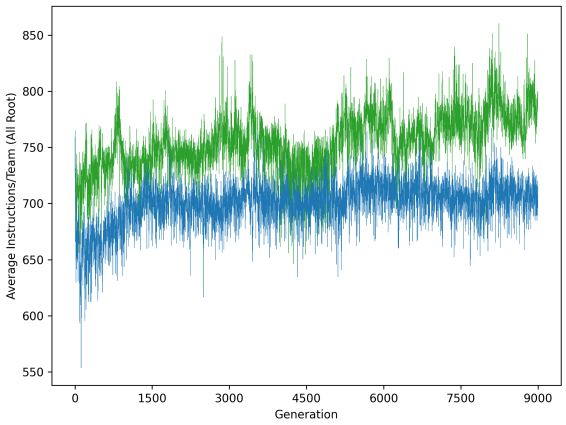
Task	TPGr	TPGd	TPGr - TPGd	rank
b-hg-dtl	152.3	104.3	48.0	5
b-hg-dtc	179.6	75.4	104.1	15
b-hg-tc	176.3	92.6	83.6	12
b-dtl-dtc	166.9	122.7	44.2	4
b-dtl-tc	163.6	139.9	23.7	1
b-dtc-tc	190.9	111.0	79.9	10
b-dtl-dtc	131.8	80.3	51.4	6
hg-dtl-tc	128.6	97.6	31.0	3
hg-dtc-tc	155.8	68.6	87.2	13
dtl-dtc-tc	143.1	115.9	27.2	2
b-hg-dtl-dtc	210.2	127.6	82.6	11
b-hg-dtl-tc	206.9	144.8	62.1	8
b-hg-dtc-tc	234.2	115.9	118.3	16
b-dtl-dtc-tc	221.5	163.2	58.3	7
hg-dtl-dtc-tc	186.4	120.8	65.6	9
all 5 tasks	264.8	168.1	96.7	14

than under TPGr. This is likely a factor of action programs simply being able to define a "range" of behaviours with action programs. Thus, for the same instruction count, many different behaviours exist. Conversely, under TPGd the only change an action can take is being "flipped" to a different discrete action, or the action type changing to a team index. Changes to a team index will split a previous single discrete action into multiple possible (discrete actions), but at the expense of introducing more complexity into the solution (an entire team or graph is added).

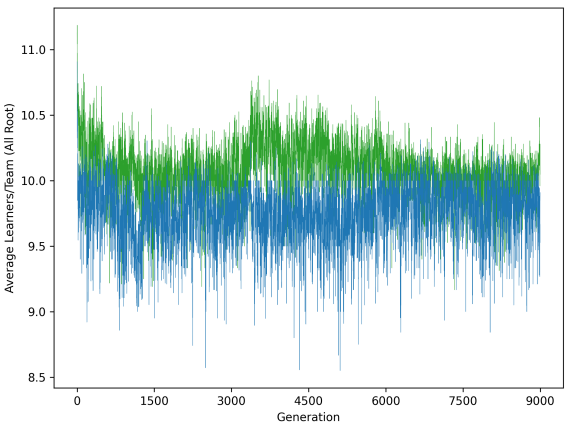
In short, TPGr is able to provide a new level of granularity into the complexity of solutions that did not exist under TPGd. Indeed, TPGd agents can increase to very large graphs, yet the vast majority of the teams comprising the graph might be hitchhikers [23, 1].



(a) Inst. per Learner



(b) Inst. per Team



(c) Learner per Team

Figure 5.6: Development of TPGd and TPGr champion complexity over the course of a run. Blue (green) represent TPGr (TPGd) respectively

Chapter 6

Bipedal Walker Experiments

6.1 Overview

Walker locomotion represents a task in which actions take the form of a vector of continuous valued actions. Thus, unlike the ViZDoom task, it is not feasible to enumerate the action space. Which is to say that too many arbitrary decisions would have to be made regarding the resolution of attempts to define a set of discrete actions. Moreover, changing the locomotion task would result in having to revisit the decisions made to build the set of discrete actions. As a consequence neural networks are often the preferred choice for attempting to derive control policies for locomotion tasks, but come with the caveat that the solutions are typically opaque/non-interpretable (§6.2).

The environment assumed in this work takes the form of the Bipedal-Walker-v3, which is implemented in the Box2D physics engine, and is released as an OpenAI Gym environment [5]. There is a normal version which features a relatively flat yet uneven terrain, and a hardcore version which contains additional obstacles. Only the normal version is used in this thesis due to time constraints.

This environment was selected because it required multiple real valued inputs, so we could test the capability of action programs in such an environment. Also, this environment is widely used, having multiple benchmark comparisons available. Lastly, such a starting point was perfect for beginning to analyse action programs given that it is a 2D environment, lacking the significant step up in difficulty of 3D environments (performing in 3D environments is a goal to build towards, and would have been tested if we had more time).

The action space consists of torque applied to four joints, the two hips and two knees (which a boxy hull/body sits on top of). Actions are clipped between -1 and 1.

The observation space consists of the hull angle, angular velocity, and horizontal and vertical speeds, the positions and angular speeds of the four joints, whether each

leg is touching the ground, and ten range finder measurements. This observation space is extended by adding $\{sin(t)/k \mid k\ in\ 1..3\}$ and $\{cos(t)/k \mid k\ in\ 1..3\}$, to further motivate the adoption of cyclical/sinusoidal motion which has been shown to be beneficial for bipedal locomotion [68, 44]. This brings the total observation space dimension to 30.

Reward is given for distance moved forward, -100 points is given if the agent falls over, and a small penalty is applied for energy use. The ultimate goal is to reach the end of the environment with minimal energy usage, the environment is deemed solved if the agent can do so with at-least 300 points in 1,600 time steps.

Results analysis source code for the bipedal walker experiments can found online[1].

6.2 Prior Results

Prior published results using GP on the OpenAI bipedal walker task could not be found, doing so is a notable contribution of this thesis. However, there are prior results on the Bipedal-Walker-v2 task which make use of neural networks, as well as GP results on different bipedal walking environments. Results from different locomotion tasks are not necessarily interchangeable because the underlying physics and / or sensor instrumentation vary between simulators. That said, there could be an opportunity for future research in terms of attempting to "transfer" ML solutions between tasks.

In the case of results using GP, the work of Ok et al. evolve control policies for a 3D bipedal walker task [46]. They use GP to create a model of the nervous system to synchronize the movement of the different joints, with each joint being controlled by a neural oscillator model (a basic rhythmic generator). Only up to 4 steps of walking were generated by their model, though this is a more complex environment then the one used in this work as it controls 3 times as many joints (12 vs 4) and exists in a 3D environment.

Wolff and Wahde also address bipedal locomotion using GP [68]. This work also makes use of more complex 3D physics with more joints to control, using two different task environments. In one environment they were able to obtain slow movement with small steps, much slower than a human would move. In the other environment the

[1] https://github.com/Ryan-Amaral/gp-bipedal-walker-analysis

biped was capable of at-most 2 steps before falling over.

Malagon and Ceberio use an earlier version of the Open AI bipedal walker [41]. They compare a method of optimizing neural net weights based on continuous univariate marginal distributions (UMDAc) to neural net weight optimization using an evolutionary algorithm (EA). In general, results showed that the UMDAc outperformed EA, with UMDAc optimization preferable to the EA with a probability of 0.821, albeit with a relatively large variance.

Finally, Landajula et al. showcases the use of symbolic (mathematical) policies as generated by deep neural nets [34]. These final symbolic policies are similar to GP programs, in fact depending on the operations allowed in GP the agent policies could come out exactly the same. Moreover, the symbolic nature of the solution implies that the specifics of the control policy are now known. They demonstrate that the generated symbolic policies outperform other deep learning methods in general by showing that the average rank of the approach for a variety of tasks is better than that of any other approach they tested. These results on the bipedal walker task are used for comparison in this thesis (though this thesis uses v3 rather than v2).

6.3 Experiment Methodology and Parameterization

The approach taken to benchmarking TPGr on the biped walker locomotion task is to first establish a baseline for GP and SBBr. Two formulations of TPG are then benchmarked. The first takes the form of TPGr as described in Section 3, the second (also TPGr) introduces hitchhiker removal and the periodic introduction of new genetic material taken from independent limited runs of SBBr (i.e. the diversity maintenance scheme of §4.2). In all cases the total generation limit is parameterized to be the same.

The specific parameterizations for the runs on the bipedal walker experiments are shown in Table 6.1. The SBBr subpopulations of TPGr+SBBr take on the same parameters as the TPGr population (aside from the omission of act_{atom}).

Not previously mentioned, g_{hh} represents the generational interval (in terms of TPG generations) in which to do hitchhiker removal (only for TPGr+SBBr). Hitchhiker removal seeks to remove any unused learners from root-teams' graphs.

For each of the 4 run types, 5 runs are done. All episodes of the environment are

Table 6.1: (Bipedal Walker) Parameters for the GP, SBBr, TPGr, and TPGr+SBBr runs.

Parameter	**GP**	**SBBr**	**TPGr**	**TPGr+SBBr**
N	360			
G	10,000			
G_{sbb}	$\emptyset$			500
g_{hh}	$\emptyset$			100
g_{fail}	$\emptyset$			100
gap	0.5			
e	5			
$rampancy$	(5,5,5)			
$team_{min}$	$\emptyset$	2		
$team_{max}$	$\emptyset$	4		
$team_{maxInit}$	$\emptyset$	2		
$lrnr_{del}$	$\emptyset$	0.3		
$lrnr_{add}$	$\emptyset$	0.2		
$lrnr_{mut}$	$\emptyset$	0.7		
$prog_{mut}$	$\emptyset$	0.5		
act_{mut}	$\emptyset$	0.7		
act_{atom}	$\emptyset$		0.8	
$prog_{maxInit}$	$\emptyset$	64		
$prog_{regs}$	$\emptyset$	8		
$actProg_{maxInit}$	512	64		
$actProg_{regs}$	8			
$inst_{del}$	0.5			
$inst_{add}$	0.5			
$inst_{swp}$	0.5			
$inst_{mut}$	0.5			

done with a max of 1,600 frames, each agent is evaluated with the mean score over 5 episodes.

Each individual bipedal walker run would take just under half a week using a consumer grade desktop with 24 CPU hyper-threads each running at 3.8 GHz. A run would typically use no more than 1-2 GB of memory. Ultimately it would take about 2 weeks to do all of the runs for each run type, and about 2 months for all of the runs mentioned in this thesis. As such, this limited the number and types of experiment that could be run due to time constraints.

6.4 Results

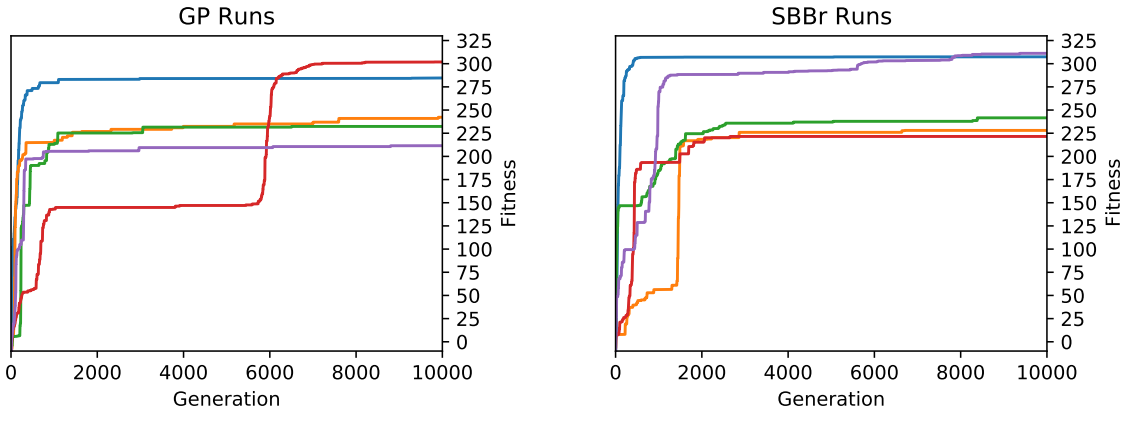
6.4.1 Evolution Comparison

Fitness Comparison

Fitness curves of the 5 runs of each run type is shown Figure 6.1. In the majority of GP runs, significant plateaus are shown to start around 1,000 generations, with minor improvements beyond. Similar happens in SBBr runs starting at about the 2,000 generation mark. TPGr performs with a wide variance, having the single worst run, and typically stalling within a few thousand generations. TPGr+SBBr shows relatively consistent improvements throughout evolution, with the earliest terminal plateau appearing at around 4,000 generations.

The run types each vary in the spread / standard deviation (SD) of final results as well, characterized along with mean results in Figure 6.2. GP and SBBr both show a relatively high SD compared to TPGr+SBBr, with TPG having a much higher SD. TPGr+SBBr achieved the highest mean final fitness value with relatively low SD.

These observations ultimately suggest that GP and SBBr are capable of performing well, though inconsistently, that TPGr generally performs poorly, and that TPGr+SBBr typically performs relatively well and consistently. This outcome is taken to imply that diversity maintenance is important for providing TPGr with useful material that can be absorbed into TPGr graphs. Also that potentially keeping the size of teams down (through hitchhiker removal) is generally useful.



(a) GP

(b) SBBr

(c) TPGr

(d) TPGr+SBBr

Figure 6.1: (Bipedal Walker) Fitness curves during training from 5 runs of each run type. Each different of a given run type is represented as different line/color.

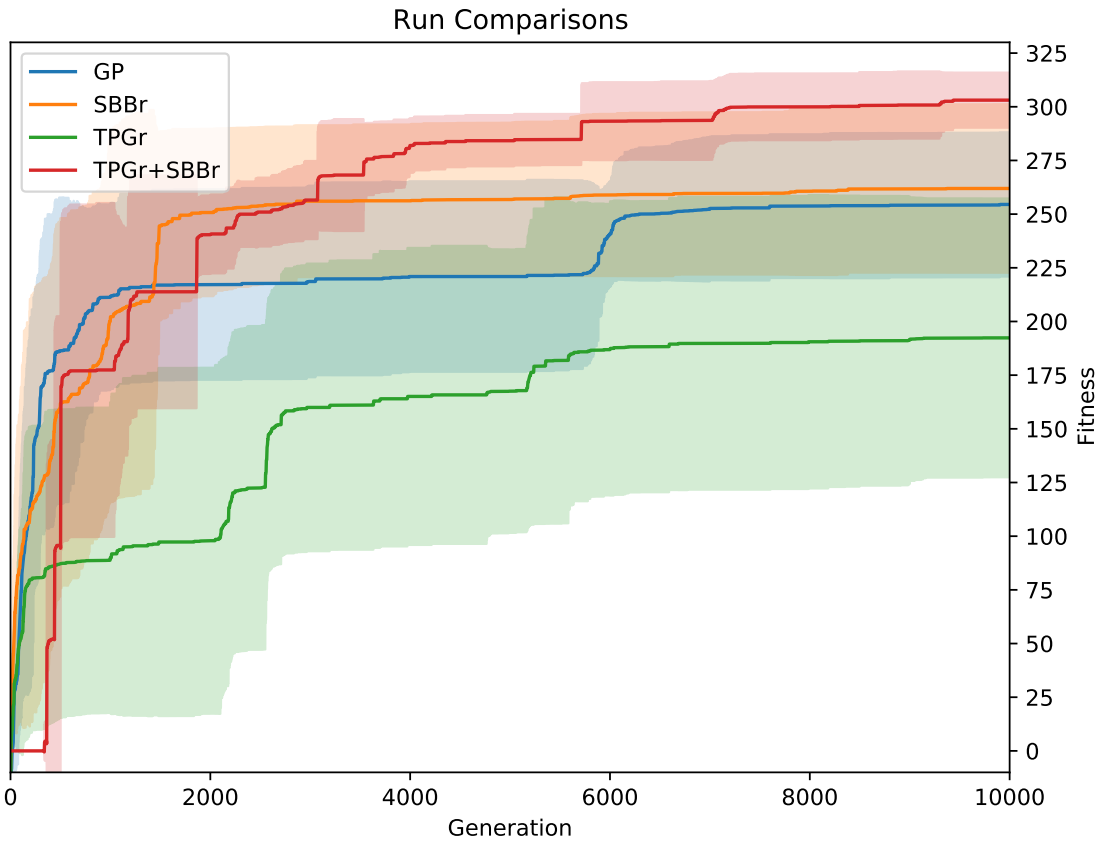


Figure 6.2: Mean fitness curves with standard deviation of each run type, each over 5 runs.

Agent Composition

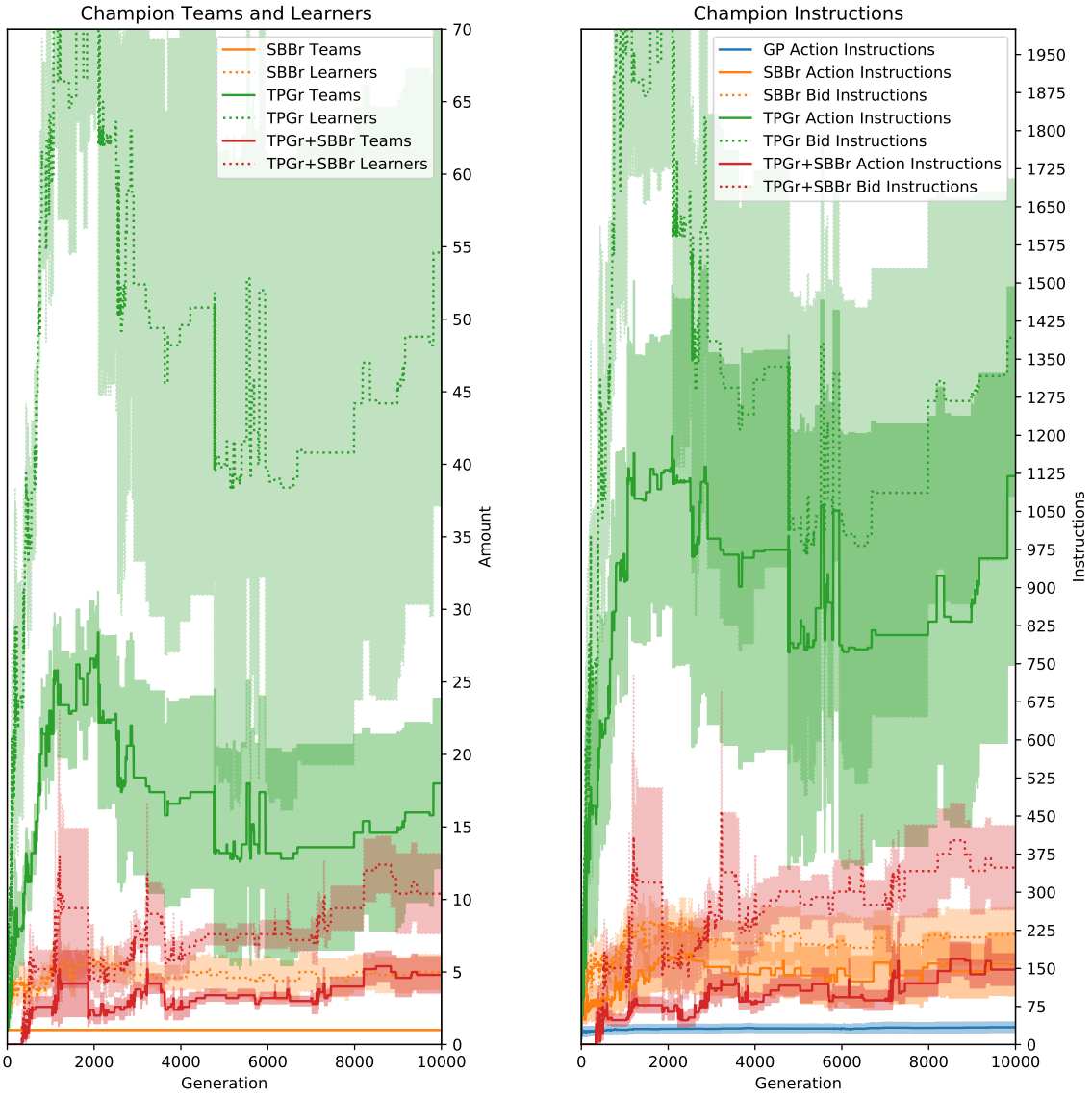
Figure 6.3 (a) shows the number of teams, and learners found within the champions of each run type (averaged over runs (mean) with $\frac{1}{2}$SD)[2]. TPGr developed significantly larger teams, such over complexity is likely a significant contributor to its relatively poor performance (Given that GP and SBBr are essentially subsets of TPGr, which also performed better). TPGr+SBBr typically ended up with champion graphs containing roughly 5 teams and 10 learners, around twice the number of learners found in SBBr champions.

Sub-figure (b) of Figure 6.3 shows the total number of instructions found within the champions of each run type (averaged over runs (mean) with $\frac{1}{2}$SD). For each run type (where applicable) there were more instructions from bidding programs than from action programs, as would be expected given that every learner has a bidding program, but not necessarily an action program (aside from SBBr). SBBr and TPGr+SBBr are found to have roughly the same number of instructions from action programs, although TPGr+SBBr tends to have significantly more bid program instructions. GP found the most simple solutions (based on instruction count), containing roughly 30 instructions, despite potentially allowing individuals to be initialized with up to 512 instructions. Note, however, that under GP and SBBr all programs are always executed, whereas under TPGr and TPGr+SBBr only the subset of programs in the path to an action program are executed to make a decision.

In Figure 6.4 we can see the number of instructions per program type found within the champions of each run type (averaged over runs (mean) with $\frac{1}{4}$SD). Both program types' sizes across all run types ranged similarly, between 25 and 35, with the only notable difference being SBBr bid instructions at 45. It is interesting that the single program solutions (GP) averaged similarly to most other program sizes.

Figure 6.5 shows the (mean and SD) number of teams and learners from the SBBr subpopulations that are in the main TPGr population. The number of teams remains roughly consistent throughout evolution whereas the number of learners is constantly on the rise. This makes sense because as TPGr evolves, the overall team population remains fixed (to N), whereas the learner population grows without restriction. The rising number of SBBr learners indicates that TPGr teams are making good use of

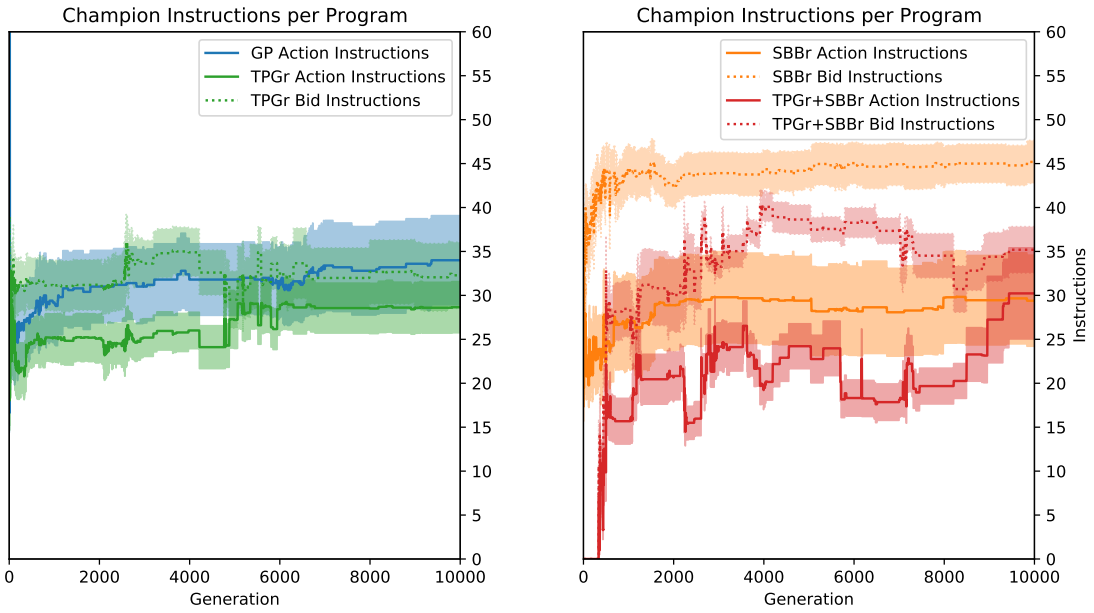
[2]Assumed to improve the legibility of the plots.



(a) Teams and Learners

(b) Instructions

Figure 6.3: Mean number of champions' teams and learners per run type with halved standard deviations (a). Mean number of champions' total instructions per run type with halved standard deviations (b). Standard deviations halved for neater presentation.



(a) GP and TPGr

(b) SBBr and TPGr+SBBr

Figure 6.4: Mean number of champions' instructions per program (bid and action) per run type with quartered standard deviation for neater presentation. GP and TPG shown in (a), SBBr and TPGr+SBBr shown in (b).

them. By the end of evolution, the average learner population was composed of roughly 62% learners imported from SBBr sup-population runs (§4.2).

The fitness curves obtained from TPGr+SBBr runs for the both TPGr populations and SBBr subpopulations are shown in Figures 6.6 to 6.10. Here we see that on their own, the SBBr sub-populations in their relatively short 500 generation runs can obtain a fitness approaching (in rare cases surpassing) the goal of 300. The TPGr population in TPGr+SBBr would therefore likely end up at-least as good. Though it can also be seen that TPG is capable of building itself up beyond the obtained SBBr champion levels, thus making use of SBBr champions' components. Indeed, this is supported by the earlier observation that most of the learner population (by the end of a run) comprises of material from the SBBr sup-population runs. It should be noted that the SBBr subpopulations start not at the beginning of the environment (as in the main TPG population), but at a point between 50% and 75% of the way to the end of the current best TPGr agent's test episode.

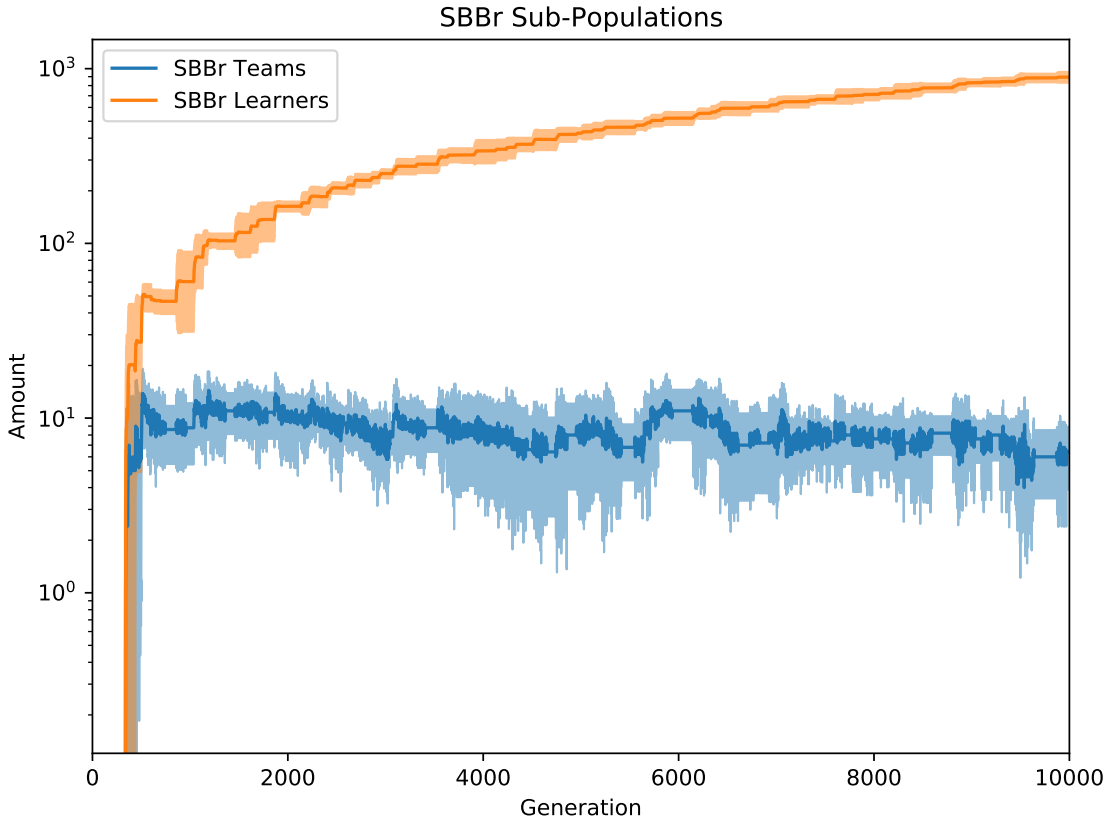


Figure 6.5: Logarithmic plot of the mean number of SBBr subpopulation teams and learners in the TPGr population with standard deviations.

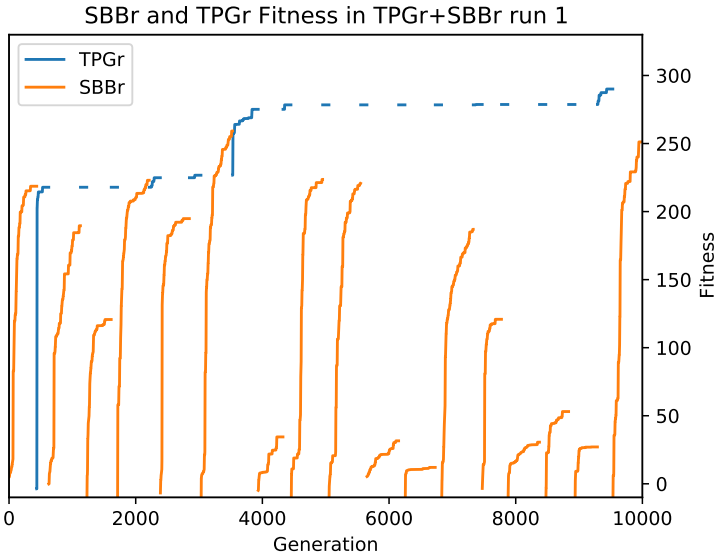


Figure 6.6: Fitness curves for SBBr and TPGr in TPGr+SBBr run 1.

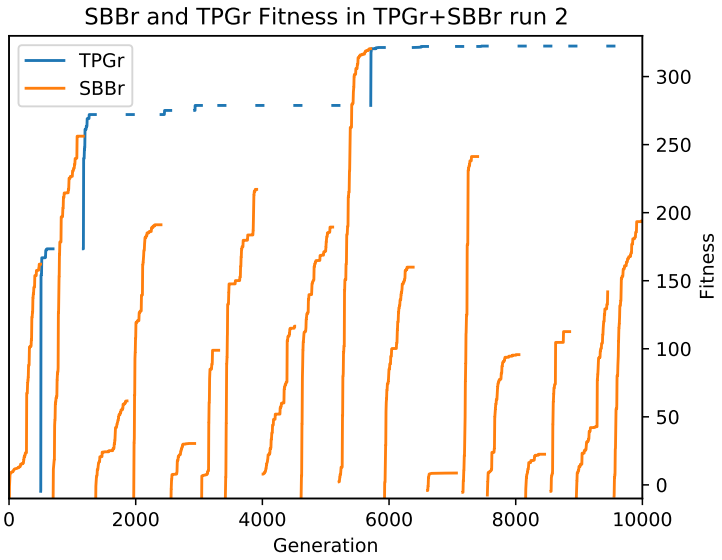


Figure 6.7: Fitness curves for SBBr and TPGr in TPGr+SBBr run 2.

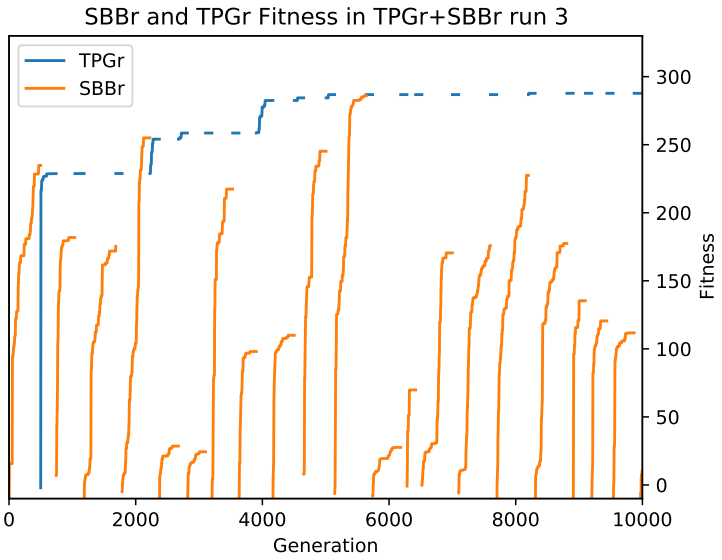


Figure 6.8: Fitness curves for SBBr and TPGr in TPGr+SBBr run 3.

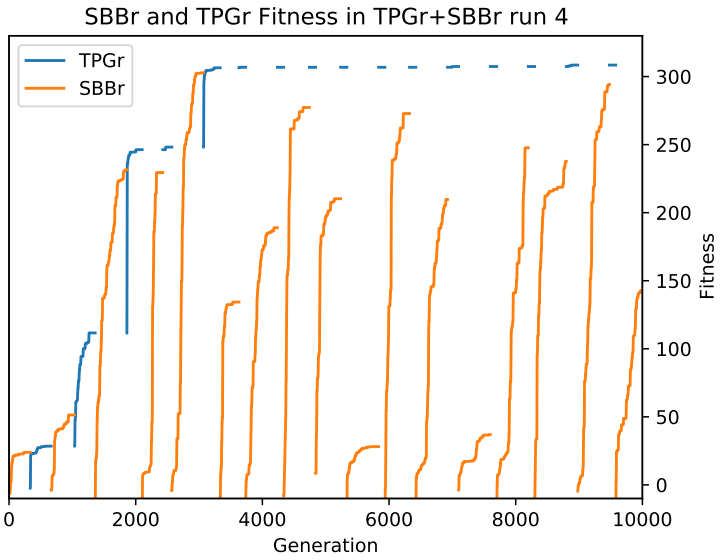


Figure 6.9: Fitness curves for SBBr and TPGr in TPGr+SBBr run 4.

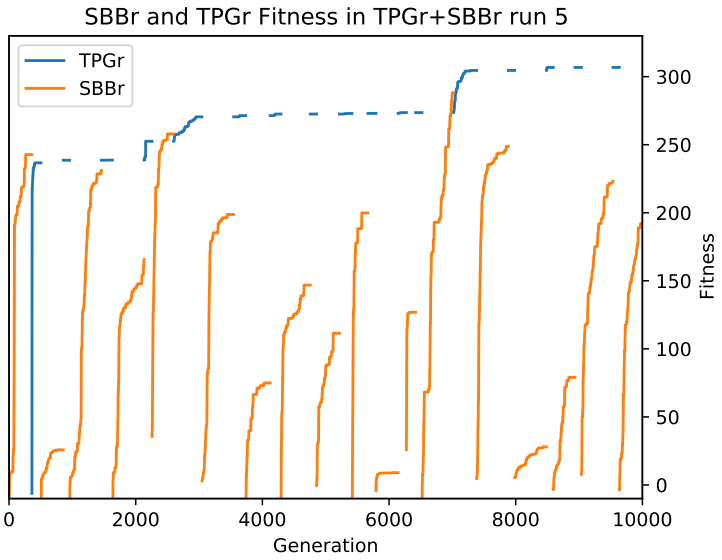


Figure 6.10: Fitness curves for SBBr and TPGr in TPGr+SBBr run 5.

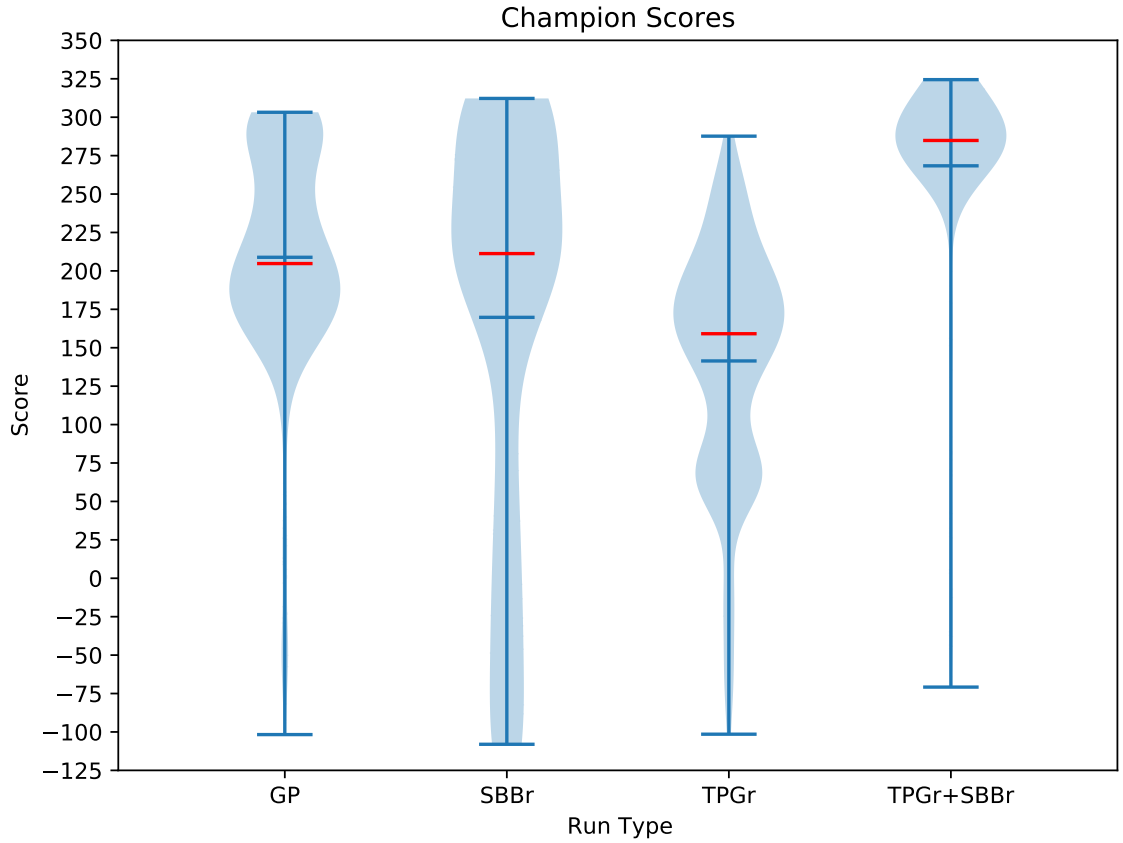


Figure 6.11: Distribution of scores obtained from champions with mean and median bars. Means in blue, medians in red. Plotted from 5 champions for each run type, each evaluated for 100 episodes. Gaussian kernel density estimation made with 500 points.

6.4.2 Champion Comparison

Post Training Evaluation

From each run type the top 5 agents were assessed over 100 episodes. The agent with the highest mean score from each run was selected ultimately as the champion of that run. The distribution of scores obtained from the champion of each run is shown in Figure 6.11, split by run type. TPGr+SBBr has notably higher mean and median scores than other run types and an overall higher distribution.

Across different runs the GP, SBBr, and TPGr champions show much less consistency overall when compared to TPGr+SBBr champions, though GP tends to show less standard deviation per individual champion scores, as seen in Table 6.2. TPGr

Table 6.2: The mean, median, and standard deviation of scores calculated over 100 evaluations for the champion of each run. Underlined run names represent the "best" agent of each run type based on mean score, other underlines show the best median and SD per run type.

Run	**Mean**	**Median**	**SD**
GP 1	236.0	254.2	67.5
GP 2	155.5	189.4	91.7
GP 3	186.8	190.8	27.4
GP 4	*294.3*	*297.9*	*19.8*
GP 5	171.7	171.9	21.0
GP Mean	**208.9**	**220.84**	**45.48**
SBBr 1	*267.9*	281.9	*54.7*
SBBr 2	31.8	-23.9	136.9
SBBr 3	189.3	219.2	83.1
SBBr 4	121.4	166.2	93.8
SBBr 5	238.4	*306.7*	118.1
SBBr Mean	**169.8**	**190.0**	**97.3**
TPGr 1	142.8	149.4	38.5
TPGr 2	120.7	157.4	82.4
TPGr 3	173.4	170.5	*22.3*
TPGr 4	*218.0*	*230.3*	59.7
TPGr 5	52.1	65.7	42.5
TPGr Mean	**141.4**	**154.7**	**49.1**
TPGr+SBBr 1	247.0	275.5	77.1
TPGr+SBBr 2	277.2	*319.0*	103.0
TPGr+SBBr 3	277.6	281.7	*22.4*
TPGr+SBBr 4	*287.4*	295.0	32.7
TPGr+SBBr 5	252.6	283.1	88.0
TPGr+SBBr Mean	**268.4**	**290.9**	**64.6**

champions performed relatively poorly in general. The mean scores consistently being lower than the median scores can be explained by how the environment's rewards work, giving a relatively large 100 point penalty on failure.

In Table 6.3, some results from this thesis's runs are compared to competitor algorithms. From this, we see that GP based solutions are typically comparable with deep learning algorithms. Specifically, there are 5 to 6 cases where a deep learning solution is better, versus 4 to 5 cases where the TPGr+SBBr solution is better. For average results from the TPGr+SBBr runs, a champion is expected to score around 86% of that scored by the strongest listed competitor.

Table 6.3: Scores obtained from the best GP and TPGr+SBBr agents (based on mean, GP 4 and TPGr+SBBr 4), as well as the mean of the TPGr+SBBr champion scores over the five runs, all compared to results from [34]. Results listed as $\frac{ours}{theirs}$ where *ours* is a results from the runs in this thesis and *theirs* is a result from an algorithm evaluated in [34], so a value of > 1 means a result from this thesis outperformed the comparative algorithm. *ours* are calculated over 100 episodes in Bipedal-Walker-v3, whereas *theirs* are evaluated over 1,000 in Bipedal-Walker-v2. The only difference between v2 and v3 is a slight change in how the "lidar sensors" worked. Underlined are the portions relating to the strongest competitor (DSP°).

Algorithm	GP Best	TPGr+SBBr Best	TPGr+SBBr Mean
DSP° [34]	*0.94*	*0.92*	*0.86*
TRPO [51]	0.95	0.92	0.86
TD3 [15]	0.95	0.93	0.87
SAC [20]	0.96	0.94	0.87
ACKTR [70]	0.98	0.96	0.90
PPO [52]	1.03	1.00	0.94
DSP [34]	1.11	1.09	1.02
Zoo [49]	1.11	1.09	1.02
A2C [42]	1.22	1.19	1.11
DDPG [40]	3.23	3.05	2.85

Action Selection

The selected champion for each run type makes use of the action space in different ways, though with some similarities, as can be seen in Figure 6.12. Each agent engages in some sort of cyclical pattern for each used limb, and in general each agent only controls 2 or 3 limbs (GP seems to slightly make use of a 4th limb).

For the multi-program/learner solutions (SBBr, TPGr, and TPGr+SBBr), surprisingly of the top champion for each run type, none of them make use of more than one learner (at-least in the first 200 frames measured). Essentially these solutions could be simplified into single program solutions. This suggests that SBBr, TPGr, and TPGr+SBBr are acting as "surrogate" environments from which different action programs are 'incubated'. The bidding programs are ultimately not necessary, aside from helping build towards the single program solutions. This is interesting as it means that the SBBr/TPGr/TPGr+SBBr formulations are not getting in the way of discovering simple solutions to a task, although TPGr could be left out of this group since it failed to produce adequate agents.

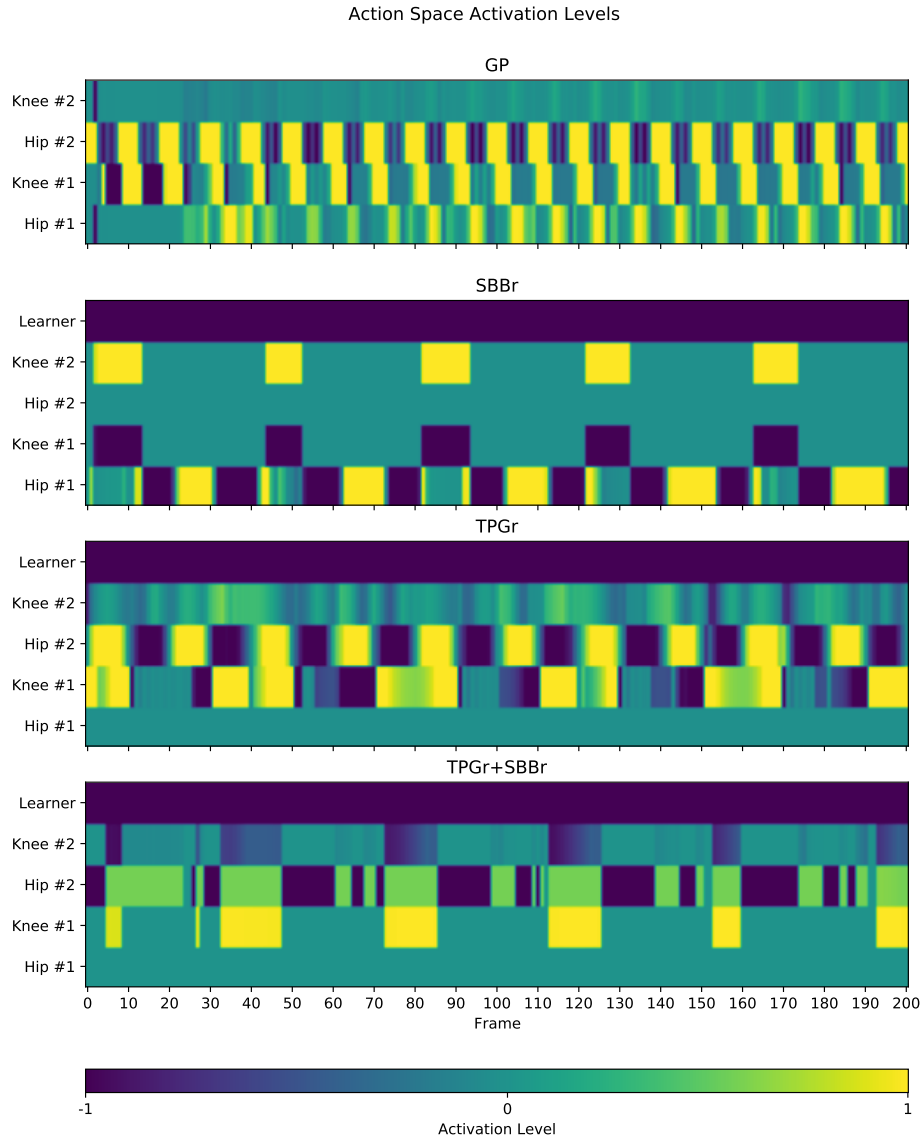


Figure 6.12: Activation levels (torques applied) to each joint throughout the first 200 frames of an episode. SBBr, TPGr, and TPGr+SBBr run types show activation for which learner is in use at a given time (though none of them make use of different learners).

The average GP and TPGr runs were not able to discover these simple effective solutions (though not to be forgotten is that the single highest scoring agent (based on mean) was from a GP run). Thus, the decomposition component of the SBBr/TPGr+SBBr formulations appears to provide a 'surrogate learning environment' for discovering such effective solutions. The function of the bidding programs appears to be to initially decompose the task, so encouraging the identification of useful action programs. As the action programs become more capable, the role of the bidding programs decreases. Conversely, neither GP (typically) or TPGr were able to support such a process.

In Table 6.4, the number of learners in total and the number actually used by each champion is shown. From this we can see that every run deemed the best previously only makes use of a single learner. In fact, for each run type, the run that is deemed the best has the least amount of learners in total, and generally the least amount of total instructions. The best solution from GP makes use of 72 instructions per time-step, the best from SBBr uses 157, the best from TPGr used 847, and the best from from TPGr+SBBr used 423 instructions per time-step. Note however that since all of these best solutions only make use of a single action program, most of the instructions used are from the unnecessary bid programs, further reducing the amount (averaging about 25-40 instructions per action program in general according to Figure 6.4).

Even with all of the extra instructions, these solutions end up less complex than the comparative neural net solutions. One such solution from Ha makes use of a relatively simple fully connected neural network, using 24 input nodes, 2 hidden layers of 40 nodes each, and 4 output nodes [19]. Therefore at each time-step, the equivalent of 5,524 instructions are executed (2,720 from weight multiplications, 2,720 elements to sum together, and 84 activation functions). Even with using the "extra" bid instructions from the TPGr+SBBr best solution, the neural net solution is still on the order of 10 times more complex, even on the order of 100 times more complex if counting only the action instructions. Though this neural network solution performs extraordinarily well, better than any benchmark solution mentioned in Table 6.3, with an average score of 347, meaning our average TPGr+SBBr solution performs 77% as well.

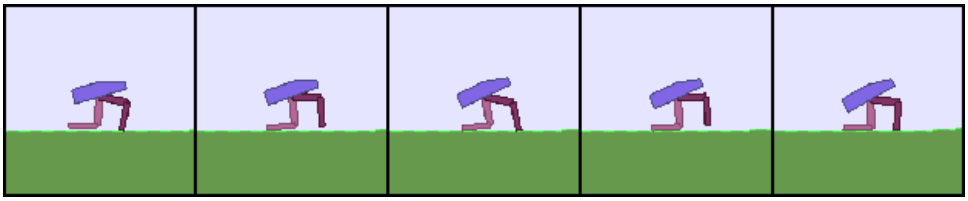
Figure 6.13 shows renders of how the different champions behave. Some general

Table 6.4: The number of learners and instructions in the champion of each run, both the total and the amount actually used in an evaluation episode. Underlines represent the run previously deemed as the best for that run type.

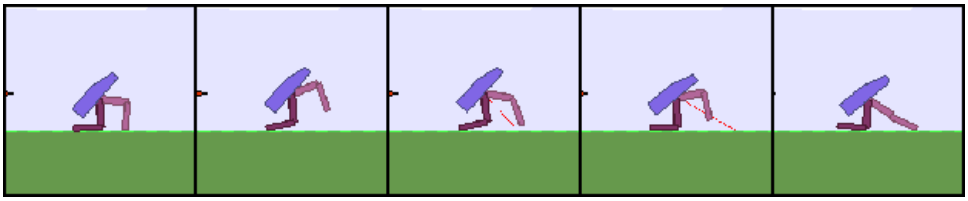
Run Type Count	Run 1	Run 2	Run 3	Run 4	Run 5
GP Inst. Total/Used	28	34	16	**72**	13
SBBr Learners Total	**2**	2	9	6	5
SBBr Learners Used	**1**	1	1	1	1
SBBr Inst. Total	**170**	110	669	338	486
SBBr Inst. Used	**157**	104	420	277	222
TPGr Learners Total	71	19	110	**19**	22
TPGr Learners Used	5	4	6	**1**	2
TPGr Inst. Total	4,086	1,115	3,902	**1,131**	995
TPGr Inst. Used	2,067	503	1,134	**847**	332
TPGr+SBBr Learners Total	12	12	19	**8**	9
TPGr+SBBr Learners Used	2	1	2	**1**	1
TPGr+SBBr Inst. Total	616	605	599	**456**	460
TPGr+SBBr Inst. Used	375	517	476	**423**	275

principals are followed by all agents, such as a kneeling gait, and a backwards tilt. The GP agent takes many small but steady steps. SBBr performs erratically. TPGr and TPGr+SBBr both make use of a longer lunge style gait.

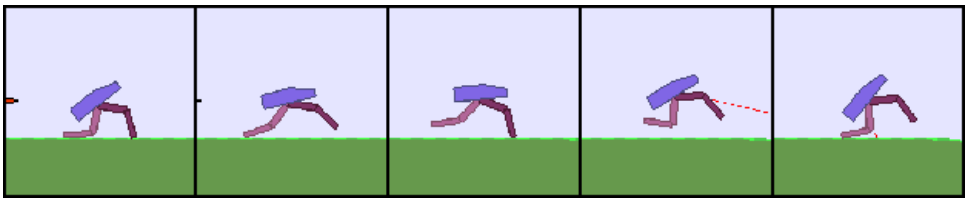
Future research might consider how solutions from the Bipedal-Walker-v3 task (as used here) could be transferred to the 'Hardcore' version of this task, or indeed form the basis for 3D versions of bipedal walking. However, recent results with the 'Hardcore' version of the Bipedal-Walker-v3 task (e.g. [66]) have required computing platforms that deploy hundreds of cores to discover appropriate solutions.



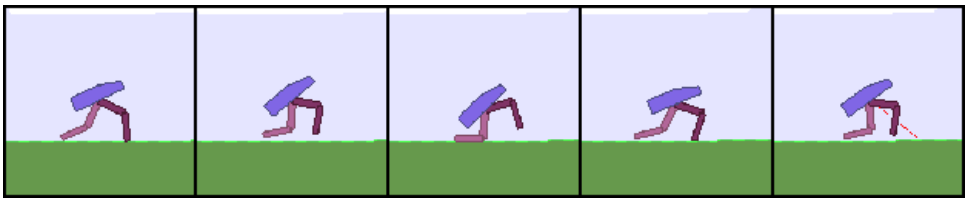
(a) GP



(b) SBBr



(c) TPGr



(d) TPGr+SBBr

Figure 6.13: Renders of the selected champion from each run type. GP render taken over 21 frames, SBBr taken over 40, TPGr over 36, and TPGr+SBBr taken over 47.

Chapter 7

Conclusion

7.1 Summary

7.1.1 Algorithm Improvements

TPG has been updated to make use of real valued actions through the use of action programs (TPGr). Granting TPG real valued actions allows it to be used in new sets of environments which require real valued actions, such as robotics and various other control tasks.

Also, action programs have been demonstrated to improve performance on environments which require discrete actions, by generating multiple discrete actions per state. Such an approach to problem decomposition allows TPGr to potentially create more efficient solutions, given that a single action program could theoretically cover the entire required action space, whereas with TPGd, at-least one program is required for each discrete action.

In addition to analyzing real valued actions in TPG, various approaches to diversity maintenance have been evaluated. This includes making use of new SBBr sub-populations within a TPGr run (unique to this thesis), curriculum learning from cycling through tasks in random order [56, 58], and rampancy, a process of repeated mutation [1].

7.1.2 ViZDoom Experiments Summary

Through the ViZDoom set of tasks, action programs from TPGr were used to generate multiple simultaneous discrete actions per state, to compare against the original TPGd formulation that was limited to single discrete actions per state (action programs vs action labels). Of the five tasks that the agents were required to simultaneously learn, TPGr was able to learn four tasks better than TPGd. Moreover, when

it came to agents generalizing over 3, 4 and 5 tasks simultaneously, TPGr was significantly better in all task combinations. Moreover, the development of TPG graphs was such that TPGr also resulted in simpler solutions (less instructions).

7.1.3 Bipedal Walker Experiment Summary

In the bipedal walker task, actions programs are a necessary prerequisite for solving the task, i.e. the original TPGd formulation cannot be applied to the task (without discretization or other forms of "preprocessing" which ultimately limits the explorable action space and creates more hyper-parameters). Agents were discovered which were capable of solving the environment at times (300+ points), and which could consistently walk through the entire terrain (though not always meeting energy requirements necessary to solve the task). Results from TPGr+SBBr on average have been shown to be competitive with some deep learning results, while not requiring computational acceleration through hardware such as GPUs.

TPGr+SBBr on average produced better solutions than other tested algorithms, both in terms of consistency with which solutions were discovered and in terms of solution simplicity (effectively single programs with in the order of 70 instructions). GP performed the next best, with similarly small programs. TPGr performed the worst and had the largest overall size (lots of learners and teams). This suggests that its not the structure of TPGr+SBBr agents that made it successful, more so the search process. Likewise the mechanism assumed for maintaining diversity was also important, with the periodic injection of new material from SBBr sub-populations providing the basis for new learner material for incorporation into TPGr.

7.2 Future Work

Future research might also consider the case of GP or SBBr with restarts, given the success of the approach with TPGr+SBBr. A conventional restart policy resets the entire population using restarts as more of a full reset than as a method to add new genetic diversity [60]. Abstaining from crossover, the GP reset runs would follow this conventional reset method, SBBr could be tested in this way as well.

The hardcore version of BipedalWalker-v3 could also be used in the future, which opens up more results to compare to. Moreover, having already evolved solutions

to the "regular" walker, these solutions could potentially be transferred to the more difficult version of the task. This might provide a more complex environment that makes use of the graph like properties of TPGr, while avoiding the considerable computational resources assumed in alternative neural network approaches as assumed to date.

A method of diversity maintenance to consider in the future in Quality Diversity (QD), which has been shown to be effective in various tasks [37]. QD goes beyond fitness based evolution in also considering diversity. Essentially, the more distinctly an agent behaves, the less likely it is to be removed from the population, even with a relatively poor fitness value. This approach can help get around local minima in the fitness search. In bipedal walker for example, the diversity criteria could be the average height of the hips, frequency of steps, average speed, or energy usage. Hopefully this method can help to consistently create agents which can solve the bipedal walker task (hardcore or not).

Bibliography

[1] Caleidgh Bayer, Ryan Amaral, Robert J. Smith, Alexandru Ianta, and Malcolm I. Heywood. Finding simple solutions to multi-task visual reinforcement learning programs with tangled program graphs. In *Genetic Programming Theory and Practice XVIII*. Springer, 2021.

[2] Edward Beeching, Jilles Dibangoye, Olivier Simonin, and Christian Wolf. Deep reinforcement learning on a budget: 3d control and reasoning without a supercomputer. In *Proceedings of the International Conference on Pattern Recognition*, pages 158–165. IEEE, 2020.

[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] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[6] Carlos A. Coello Coello. Evolutionary multi-objective optimization: a historical view of the field. *IEEE Computational Intelligence Magazine*, 1(1):28–36, 2006.

[7] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2020.

[8] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[9] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann, 1989.

[10] Janez Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[11] Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, and Maxime Pelcat. Gegelati: Lightweight artificial intelligence through generic and evolvable tangled program graphs. In Tomasz Kryjak and Andrea Pinna, editors, *DASIP*

'21: Workshop on Design and Architectures for Signal and Image Processing, pages 35–43. ACM, 2021.

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

[13] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016.

[14] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing, Second Edition*. Natural Computing Series. Springer, 2015.

[15] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *ICML*, pages 1582–1591, 2018.

[16] George Gerules and Cezary Z. Janikow. A survey of modularity in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 5034–5043. IEEE, 2016.

[17] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3:95–99, 1988.

[18] Faustino J. Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[19] David Ha. Reinforcement Learning for Improving Agent Design. *Artificial Life*, 25(4):352–365, 11 2019.

[20] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2017.

[21] Malcolm I. Heywood and Peter Lichodzijewski. Symbiogenesis as a mechanism for building complex adaptive systems: A review. In *Applications of Evolutionary Computation, Part I*, volume 6024 of *Lecture Notes in Computer Science*, pages 51–60. Springer, 2010.

[22] Gregory S Hornby. On run time libraries and hierarchical symbiosis. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 795–802. ACM, 2009.

[23] Alexandru Ianta, Ryan Amaral, Caleidgh Bayer, Robert J. Smith, and Malcolm I. Heywood. On the impact of tangled program graph marking schemes under the atari reinforcement learning benchmark. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 111–119. ACM, 2021.

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

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

[26] Stephen Kelly and Malcolm I. Heywood. Discovering agent behaviors through code reuse: Examples from half-field offense and ms. pac-man. *IEEE Trans. Games*, 10(2):195–208, 2018.

[27] Stephen Kelly and Malcolm I. Heywood. Emergent solutions to high-dimensional multitask reinforcement learning. *Evolutionary Computation*, 26(3):378–380, 2018.

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

[29] Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. A modular memory framework for time series prediction. In Carlos Artemio Coello Coello, editor, *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, pages 949–957. ACM, 2020.

[30] Stephen Kelly, Robert J. Smith, Malcolm I. Heywood, and Wolfgang Banzhaf. Emergent tangled program graphs in partially observable recursive forecasting and ViZDoom navigation tasks. *ACM Transactions on Evolutionary Learning and Optimization*, 1, 2021.

[31] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE Press, 2016.

[32] Łukasz Kidziński, Carmichael Ong, Sharada Prasanna Mohanty, Jennifer Hicks, Sean Carroll, Bo Zhou, Hongsheng Zeng, Fan Wang, Rongzhong Lian, Hao Tian, Wojciech Jaśkowski, Garrett Andersen, Odd Rune Lykkebø, Nihat Engin Toklu, Pranav Shyam, Rupesh Kumar Srivastava, Sergey Kolesnikov, Oleksii Hrinchuk, Anton Pechenko, Mattias Ljungström, Zhen Wang, Xu Hu, Zehong Hu, Minghui Qiu, Jun Huang, Aleksei Shpilman, Ivan Sosin, Oleg Svidchenko, Aleksandra Malysheva, Daniel Kudenko, Lance Rane, Aditya Bhatt, Zhengfei Wang, Penghui Qi, Zeyang Yu, Peng Peng, Quan Yuan, Wenxin Li, Yunsheng Tian, Ruihan Yang, Pingchuan Ma, Shauharda Khadka, Somdeb Majumdar, Zach Dwiel, Yinyin Liu, Evren Tumer, Jeremy Watson, Marcel Salathé, Sergey

Levine, and Scott Delp. Artificial intelligence for prosthetics: Challenge solutions. In Sergio Escalera and Ralf Herbrich, editors, *The NeurIPS '18 Competition*, pages 69–128, Cham, 2020. Springer International Publishing.

[33] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, 1989.

[34] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5979–5989. PMLR, 18–24 Jul 2021.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nat.*, 521(7553):436–444, 2015.

[36] Seunghwan Lee, Moonseok Park, Kyoungmin Lee, and Jehee Lee. Scalable muscle-actuated human simulation and control. *ACM Trans. Graph.*, 38(4), July 2019.

[37] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

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

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

[40] T. Lillicrap, Jonathan J. Hunt, A. Pritzel, N. Heess, T. Erez, Yuval Tassa, D. Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2016.

[41] Mikel Malagon and Josu Ceberio. Evolving neural networks in reinforcement learning by means of umdac, 2019.

[42] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas 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.

[44] Jared M. Moore, Catherine L. Shine, Craig P. McGowan, and Philip K. McKinley. Exploring Bipedal Hopping through Computational Evolution. *Artificial Life*, 25(3):236–249, 2019.

[45] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *J. Mach. Learn. Res.*, 21:181:1–181:50, 2020.

[46] S. Ok, K. Miyashita, and K. Hase. Evolving bipedal locomotion with genetic programming - a preliminary report. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 2, pages 1025–1032 vol. 2, 2001.

[47] Miguel Oliveira, Lino Costa, Ana Rocha, Cristina Santos, and Manuel Ferreira. Multiobjective optimization of a quadruped robot locomotion using a genetic algorithm. In António Gaspar-Cunha, Ricardo Takahashi, Gerald Schaefer, and Lino Costa, editors, *Soft Computing in Industrial Applications*, pages 427–436, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[48] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7652–7662, 2020.

[49] Antonin Raffin. Rl baselines3 zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

[50] Michael D. Schmidt and Hod Lipson. Age-fitness pareto optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 543–544. ACM, 2010.

[51] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[53] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[54] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

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

[56] 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 *LNCS*, pages 135–150. Springer, 2018.

[57] Robert J. Smith and Malcolm I. Heywood. Evolving Dota 2 Shadow Fiend bots using genetic programming with external memory. In *Genetic and Evolutionary Computation Conference*, pages 179–187. ACM, 2019.

[58] Robert J. Smith and Malcolm I. Heywood. A model of external memory for navigation in partially observable visual reinforcement learning tasks. In *European Conference on Genetic Programming*, volume 11451 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2019.

[59] Robert J. Smith and Malcolm I. Heywood. Evolving a dota 2 hero bot with a probabilistic shared memory model. In Wolfgang Banzhaf, Eric Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel, editors, *Genetic Programming Theory and Practice XVII*, pages 345–366. Springer International Publishing, 2020.

[60] Michael Solano and Istvan Jonyer. Performance analysis of evolutionary search with a dynamic restart policy. In David Wilson and Geoff Sutcliffe, editors, *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA*, pages 186–187. AAAI Press, 2007.

[61] Lee Spector and Sean Luke. Cultural transmission of information in genetic programming. In *Annual Conference on Genetic Programming*, pages 209–214. Morgan Kaufmann, 1996.

[62] Peter Stone. *Layered learning in multiagent systems - a winning approach to robotic soccer*. Intelligent robotics and autonomous agents. MIT Press, 2000.

[63] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[64] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. In Carlos Artemio Coello Coello, editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 414–424. ACM, 2020.

[65] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[66] Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth O. Stanley. Enhanced POET: open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In *Proceedings of the International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9940–9951. PMLR, 2020.

[67] R. A. Watson and J. B. Pollack. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 702–709. Morgan Kaufmann, 2001.

[68] Krister Wolff and Mattias Wahde. *Evolution of Biped Locomotion Using Linear Genetic Programming*. 10 2007.

[69] Matthew Wright. Providing real-valued actions for tangled program graphs under the cartpole benchmkark. Master's thesis, Dalhousie University, Faculty of Computer Science, August 2020.

[70] Yuhuai Wu, Elman Mansimov, Roger B. Grosse, Shu Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *NIPS*, 2017.

[71] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *Proceedings of the International Conference on Learning Representations*, 2017.

[72] Marek Wydmuch, Michal Kempka, and Wojciech Jaskowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 11(3):248–259, 2019.