PROVIDING REAL-VALUED ACTIONS FOR TANGLED PROGRAM GRAPHS UNDER THE CARTPOLE BENCHMARK

by

Matthew Wright

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

at

Dalhousie University
Halifax, Nova Scotia
August 2020

Table of Contents

List of Tables

List of Figures

List of Acronyms Used

ALE Arcade Learning Environment.

GP Genetic Programming.

NEAT NeuroEvolution of Augmenting Topologies.

REINFORCE REward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility.

RL Reinforcement Learning.

SBB Symbiotic Bid-based Programming.

TPG Tangled Program Graphs.

Abstract

The Tangled Program Graph framework (TPG) is a genetic programming approach to reinforcement learning. Canonical TPG is limited to performing discrete actions. This thesis investigates mechanisms by which TPG might perform real-valued actions. Two approaches are proposed. In the first, a decision-making network extracts state from TPG's internal structure. A gradient-based learning method tailors the network to this representation. In the second, TPG is modified to generate a state representation in an external matrix visible to the decision-making network. No additional learning algorithm is used to configure the 'decision-making network'. Instead, TPG adapts to use the default configuration. This thesis applies these approaches to a modified version of the classic CartPole environment that accepts real-valued actions. This enables the comparison between discrete action configurations of the task and the real-valued formulation. Results indicate that there is no additional complexity in TPG solutions under real-valued action versus discrete action configurations.

Chapter 1

Introduction

Tangled Program Graphs (TPG), a genetic programming-based Monte-Carlo reinforcement learning algorithm developed by Kelly and Heywood [17], has demonstrated an ability to compete with state-of-the-art deep learning approaches to reinforcement learning, demonstrating high performance in both the Arcade Learning Environment [5] [16] and VizDoom [18] [30]. However, the algorithm as originally implemented suffers from the limitation that it is only applicable to environments that accept discrete actions; the current action suggestion mechanism built into TPG is inherently unable to take real-valued actions. This is due to the pre-assignment of sampled discrete actions to the modules within TPG responsible for action selection. This work attempts to modify TPG using additional data structures that facilitate real-valued action selection without a fundamental modification to TPG as currently developed.

Real-valued action selection in reinforcement learning problems is typically achieved using a decision-making network (usually a neural network or deep neural network) that takes the environment state as input and produces one or more real values as output. To continually encourage exploration, these suggested actions are then typically used to build a normal distribution or distributions centered around the output values(s). The final actions taken are then sampled from those distributions. This allows the agent to occasionally take unexpected actions that may lead to increased performance. These networks are computationally expensive to both train and use. For example, the network used by DeepMind in [25] to play Atari games used 32 8x8 convolutional filters in its first convolutional layer, 64 4x4 convolutional filters in its second convolutional layer, 64 3x3 convolutional filters in its third convolutional layer, and a final 512-unit fully-connected hidden layer. TPG successfully solves similar problems with multiple orders of magnitude less computational cost [17] [16].

This work adopts a hybrid approach by attaching a linear decision-making network to a TPG instance. The TPG instance is responsible for engineering features

which are provided as input to the decision-making network. Neural networks can achieve better performance more quickly and using a less computationally expensive architecture when presented with deliberately-engineered features [15]. This work allows TPG to assume the role of feature engineer. This is done on the basis that TPG's action-selection mechanism necessarily requires that TPG is already building a robust state representation/feature set deliberately suited for action selection. The hybrid algorithms examined in this work involve providing this feature set to a lightweight neural network capable of suggesting real-valued actions. By 'lightweight' it is implied that the mapping from features to real-valued action is of the form $y = mx + c$ (i.e. linear). This minimizes the complexity of the mapping, but assumes that TPG can find features that are sufficiently informative to derive a useful real-valued action.

This work explores two possible implementations of the hybrid algorithm for operation under the conditions described above. Initially, a neural network-based learning algorithm, REINFORCE [33], was passed data from within existing TPG data structures as its input. These data structures hold TPG's own internal state representation, which it uses to suggest actions in the case where no additional decision network is involved, and so are presumed to hold useful features that might be leveraged by another learning algorithm. In this implementation, TPG is unmodified and REINFORCE is only provided with TPG instances that have already been trained in isolation and with no knowledge of REINFORCE. REINFORCE is then responsible for learning to make use of the features provided by TPG. Following this, a second approach is explored in which TPG is encouraged to perform explicit feature-engineering in an external 'indexed memory' data structure. Actions are chosen by a separate decision-making network which acts as the sole action-selection mechanism at all times to ensure that TPG must learn to provide useful features.

In all experiments, the classic CartPole environment [4] was used. This was chosen because it is a widely-used reinforcement learning benchmark [27] [12] [26] [1] [3] and known by the author to be readily tractable to TPG. Specifically, the OpenGym Python implementation was assumed[1]. All other aspects of this thesis were then coded from first principles using Python. Such an approach enabled a particularly

[1] `https://gym.openai.com/envs/CartPole-v1/`

tight coupling between the various components of TPG, indexed memory and REINFORCE.

The remainder of this thesis is organized as follows. Chapter 2 provides a background for readers who may be unfamiliar with one or more of the major components discussed in this work. Specifically, this chapter: explains the principles of reinforcement learning; explains the principles of evolutionary algorithms and genetic programming in particular; discusses the major pre-cursors to TPG to ensure readers understand the major concepts behind the TPG algorithm and the path that led to its development; provides a detailed description of TPG as implemented in this work such that it could be reproduced; explains gradient-based reinforcement learning in general and the specific gradient-based reinforcement learning algorithm used in this work, REINFORCE; and reviews the CartPole reinforcement learning environment on which the experiments in this thesis are performed. Chapter 3 discusses the motivation behind the algorithms examined in this work. Specifically, it discusses the current obstacles to extending TPG to produce real-valued actions and how the hybrid algorithms discussed in this thesis overcome those obstacles. It provides an overview of the hybrid algorithms examined. Chapter 4 provides a detailed description of the experiments carried out in this work. This section provides precise implementation details of the algorithms discussed in chapter 3. Chapter 5 lists the results of the experiments carried out in this work and discusses the meaning of these results. Chapter 6 includes a summary of the results collected, a review of their meaning with regard to the goals set out in this work, and a discussion of future work that may be performed based on the results collected.

Chapter 2

Background

This work involves the use of reinforcement learning algorithms originating from two quite different paradigms in machine learning: genetic programming and gradient-based learning. This section provides a description of the topics upon which this work is built, including reinforcement learning in general, the REINFORCE algorithm, and genetic programming, from basic linear genetic programming to the variants of TPG used in this work.

2.1 Reinforcement Learning

Reinforcement learning (RL) refers to a branch of machine learning in which the goal is to develop policies, often referred to as agents, through direct interaction with an environment. This is distinct from most other forms of machine learning in which learning is performed relative to a static data set. RL agents are intended to learn policies through a process of continual interaction with the target environment, which responds to the agent's performance with a reward. Policies specifically refer to action selection given the state of the environment, and may or may not be a function of both present and previous state. For example, an agent may be a robotic arm (or more precisely the software controlling the robotic arm) and the environment a specific build sequence in an assembly line. The arm may be intended to learn how to separate boxes on a conveyor belt based on size and move them to designated locations. The robot is intended to operate continually through time, picking up and moving a new box after the previous one has been placed. Because different machine learning paradigms often overlap considerably in their methods and objectives, hard and fast lines between reinforcement learning and other areas of focus in machine learning are difficult to draw. However, classical reinforcement learning applications are typically defined as attempts to solve the problem of finite Markov decision processes [6]. This is a formalization of sequential decision making where actions taken by an

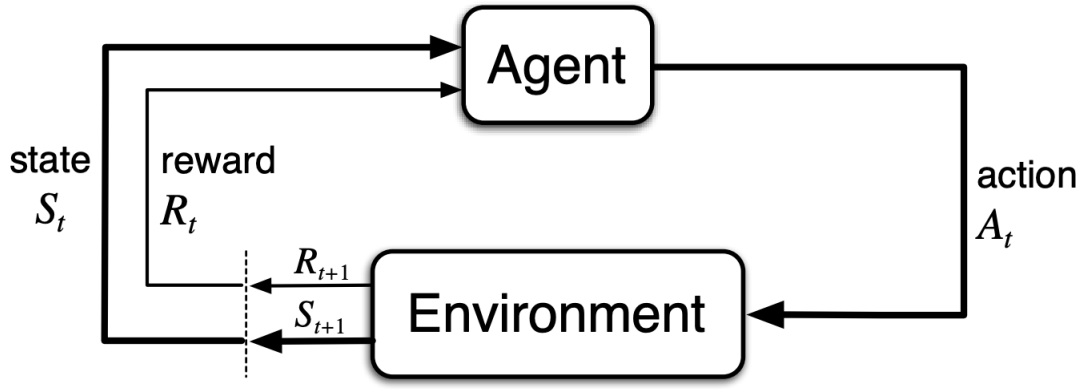

Figure 2.1: The standard reinforcement learning loop. This loop summarizes the problem of finite Markov decision processes, where the agent affects the environment, which later affects the agent, and so on. Taken from Sutton and Barto [32].

agent influence the future state of the environment, i.e. the learning agent explicitly interacts with an environment with the objective of maximizing long term reward. An agent's policy therefore represents a strategy for maximizing the long term reward under a specific environment. This problem is well summarized by the depiction of the agent-environment interface taken from Sutton and Barton's canonical text on reinforcement learning [32] and shown in figure 2.1. In this diagram:

- t refers to the current timestep in a sequence of discretized timesteps
- S_t is the state of the environment at timestep t
- A_t is the action taken at timestep t
- R_t is the reward signal received by the agent from the environment at timestep t
- S_{t+1} is the new state of the environment at timestep $t+1$ after taking action A_t
- R_{t+1} is the reward signal received by the agent at timestep $t+1$ after taking action A_t

These items are discussed further below.

2.1.1 Continuing vs Episodic Tasks

To fully characterize time as viewed by a learning agent, it is necessary to define the way in which the duration spent on the task is discretized and seen by the agent. Typically, the duration spent on a task is divided into discrete timesteps: rather than receiving a continuous stream of information from the environment, the agent samples the environment at discrete times, referred to as timesteps in the context of reinforcement learning.

Reinforcement learning tasks are often broken down further into two categories: continuing or episodic. While a continuing task may theoretically continue endlessly without interruption, an episodic tasks has a terminal state. This may be defined in terms of an environment configuration (for example, the task was solved or the agent has irrecoverably failed) or a pre-determined maximum number of timesteps or a combination of both.

2.1.2 State

In a given environment, a learning agent is exposed to a finite and well-defined amount of information contained in the environment. This information is referred to as the *state* of the environment and the state of the environment at timestep t is represented as S_t. This state information defines the input on which an agent operates.

The components of the state are highly dependent on the requirements of the environment and the capabilities of the agent, and are ultimately decided by the researcher or application designer. For example in the case of the robot arm, the state may be comprised of video feeds from cameras focused on the arm and boxes as well as pressure sensors in the arm itself. This state representation may be chosen as the most feasible given the environment and available hardware and with the belief that this likely contains sufficient information for the agent to operate. At timestep t, the state S_t is comprised of, for example, images sampled from the cameras and sensor readouts sampled from the pressure sensors. How this information is formatted and presented to the agent is up to the researcher or application developer and may depend on the algorithm used. For all algorithms use in this work, the state is represented as a vector of floating-point values sampled from the environment at each timestamp.

2.1.3 Actions

The action A_t is the action suggested by the agent at timestep t. The definition of actions is highly dependent on the environment and the agent. In the case of the robot arm, the action taken at timestep t may be the assertion of motor controller values by the software to reposition the arm. In that formulation, A_t would be the expression of a vector of values calculated by the agent as a response to the state S_t.

Much like the distinction made between continuing and episodic tasks, there is an important distinction to be made between continuous actions and discrete actions. A discrete action is an action selected from a pre-defined set of possible actions. In the case of the robot arm, a (partial) set of possible actions may include the option to clasp or not to clasp. At any given timestep, the arm must decide whether or not to attempt to clasp a box and suggesting an action entails sampling from these behaviours. Although the clasping example given here is a binary decision, the set of actions may be arbitrarily large.

In the case of continuous actions, the action is a real value chosen from a continuous number line, typically from within a prescribed range. For example, in the case of the robot arm, the agent may be required to calculate the amount of pressure necessary to apply to the box to successfully lift it, which may be a floating-point value in Pascals. Continuous actions are typically floating-point values, though lines may be blurred at times. For example, if the robot expresses Pascals of applied pressure by writing an unsigned integer value from a finite range to a digital-to-analog converter then the real-valued action could be re-formulated as a discrete action selected from a very large set of unsigned integers. In this work, the distinction is simplified: discrete actions describe actions selected from a low-dimensional (e.g. binary) set, while continuous actions describe the calculation of one or more floating-point values.

2.1.4 Rewards

Reinforcement learning problems are typically formulated such that at any given timestep, the agent receives a reward from the environment after taking its action. The reward signal provided to the agent is the agent-environment interface's primary feedback mechanism. It is intended to inform the agent whether it has been successful in its task. The combination of reward and state provide the agent with enough

information to direct itself during operation: the state lets the agent know how the environment is configured, and the reward signal provides context to the agent that allows it to direct its behaviour with the goal of receiving a higher reward.

The design of the reward signal is dependant on the environment and varies in practice. However, the reward signal typically conforms to some common characteristics:

- The reward signal is often a numeric value.
- A positive reward signal typically means the agent has performed well.
- A zero or negative reward signal means the agent has performed poorly.

In the case of the robot arm, the agent may receive a reward of +1 every time a box is re-positioned correctly and -1 every time it drops the box during movement. The reward R_{t+1} represents the reward received by an agent at timestep $t+1$, which is by definition in response to action A_t. A similar notion is the *return*, represented as G_t. This is the cumulative reward received by an agent, calculated as the sum of all rewards seen from timestep t until the end of the episode. For example, G_0 is the sum of all reward signals received by an agent over a single episode.

Rewards and returns define goals for the agent, which must ultimately learn to maximize these signals. A solution to the reinforcement learning problem is an algorithm that conforms to the Markov decision process formulation and can adapt its own behaviour to maximize its return.

The separation of reward and return lead naturally to two classes of learning algorithms: Monte Carlo methods and temporal-difference (TD) methods. In the former, episodes are allowed to play out in full. The agent's policy, i.e. the agent's action selection mechanism when presented with some environment state, is left unchanged over the course of the episode. At the end of the episode, the agent or learning algorithm retains a full record of all states, actions and reward signals encountered during the episode and uses this complete information to attempt to modify the policy in such a way as to generate a higher return on the next episode. In the latter case, the learning algorithm updates the agent's policy simultaneously to acting in the environment. In this configuration, the learning algorithm makes use of each new reward

signal and the corresponding state and action that led to it to update the agent's behaviour definition at each timestep. While TD methods attempt to maximize their reward signal during an episode, Monte Carlo methods are agnostic to the dynamics of the reward accumulation system and attempt only to maximize the final return. All algorithms considered in this work are Monte Carlo algorithms.

2.2 Genetic Programming

2.2.1 Evolutionary Algorithms

Genetic programming is a variation of evolutionary algorithms, which are themselves a category within the wider field of evolutionary computation [10]. Evolutionary algorithms attempt to simulate aspects of natural evolution with the goal of optimizing a solution to a given problem [2]. Under an evolutionary algorithm, solutions are formulated such that they can be copied (i.e. that they can *reproduce*, often referred to in the literature as *cloning*), that they can be incrementally modified (i.e. that they can be *mutated*, often referred to in the literature as *variation*), and that they can be assigned a quantitative measure of their quality as a solution to the given problem, where this quantitative measure is typically referred to as the solution's *fitness*. As described by Bäck et. al. [2] and modified here, a general evolutionary algorithm operates according to the following steps:

1. Randomly initialize a population of candidate solutions.
2. Assess the fitness of each member of the population.
3. Remove some portion of the least fit candidates.
4. From the remaining candidates, refill the population by stochastically:
   (a) Selecting from the remaining candidates.
   (b) Reproducing (i.e. cloning) a new candidate from the selection.
   (c) Mutating the new candidate to produce a modified solution (i.e. performing variation).

5. Repeat from step 2 until some termination criteria, e.g. pre-determined adequate fitness, is reached. At this point, the most fit candidate solution is the final result and output from this algorithm.

Unlike most approaches to machine learning genetic programming will:

- Assume multiple candidate solutions simultaneously.
- Modify candidate solutions without use of gradient information.
- Map between representation (genotype) and search (phenotypic) space.
- Support the emergence of structure in addition to the optimization of a specific structure.

This thesis will assume a *linear* genetic programming representation, described in section 2.2.2 and adopt the Tangled Program Graph framework for addressing the last point in particular, described in section 2.4.

2.2.2 Linear Genetic Programming

Genetic programming (GP) is an approach to evolutionary algorithms in which the candidate solutions are computer programs. In this approach, the fitness of an individual is assessed by executing the program and evaluating the degree to which the program solves the given problem. Early work in genetic programming structured the programs as trees of instructions or operations [19]. This work uses an alternative approach known as linear genetic programming [7] in which the candidates are structured as a sequential list of instructions. A candidate may be reproduced by creating a copy of this list, and mutation is a matter of altering one or more instructions in the sequence. This approach to GP provides the basic representation for programs assumed in the primary algorithm investigated in this work, Tangled Program Graphs (TPG), though TPG may assume any representation of GP.

The following is a description of the linear GP model built into the TPG implementation used in this work. Under this formulation, a GP candidate is a list of instructions that operate on a vector of input attributes as well as eight floating-point registers. The vector input is highly dependant on the application being developed.

Under reinforcement learning it is typically the state S_t, though it can also be one or more samples from a static dataset in non-reinforcement learning applications. The eight registers are general purpose registers that may be used by the GP candidate to store the results of calculations performed on a combination of either or both of the input vector and other register values. For example, a program may execute the following instructions:

$$R[2] \leftarrow R[2] + S[1]$$
$$R[0] \leftarrow R[0] - R[2]$$

where $S[index]$ indicates indexing into the state vector and $R[index]$ indicates indexing into a particular register.

At the end of it's execution, a GP candidate will hold eight floating-point values in its registers which are the result of modifications made during execution. The values in these registers are the output of program execution. Different applications will interpret these outputs in varying ways. For example, a linear GP candidate being trained as a binary classifier may use the value in register 0 as its confidence in a positive classification. In a multiclass classifier, several registers may be taken together as the candidate's confidence scores across multiple classes. When developing a reinforcement learning agent, it is up to the developer to extract an action suggestion from the registers. In this work, the action selection mechanism is not left to individual GP programs but formulated as a process across multiple programs using the TPG framework, described in section 2.4.

Most instructions used this formulation of linear GP are binary operations of the form

$$R[i] \leftarrow R[i] \; op \; V[j]$$

where

- i and j are the indices into the registers and vector V respectively.
- The vector V is either the input vector or the program's registers depending on the mode of the instruction.
- op is the mathematical operation to be carried out.

The instruction set used also includes unary operations of the form

$$R[i] \leftarrow op\ V[j]$$

Finally, the instruction set contains a conditional operation of the form

$$R[i] < V[j]\ ?\ R[i] \leftarrow -R[i]\ :\ \text{no-op}$$

A complete instruction comprises four components:

- The target index
- The source index
- The instruction mode
- The op-code

The target index is the index i from the examples above and defines into which register the result will be stored. The source index is the index j from the examples above and defines the index into the source or input vector, represented as V in the examples above, from which the operand will be retrieved. The instruction mode is a binary setting that indicates whether the source vector V will be the program's input vector (i.e. the state in the context of reinforcement learning) or the program's registers. Finally the op-code represents the operation to be carried out. A full list of the instructions used in this work's formulation of linear GP is described in table 2.1.

Op-code	Operation
Addition	$R[i] \leftarrow R[i] + V[j]$
Subtraction	$R[i] \leftarrow R[i] - V[j]$
Multiplication	$R[i] \leftarrow V[j] * 2.0$
Division	$R[i] \leftarrow V[j]/2.0$
Cosine	$R[i] \leftarrow cos(V[j])$
Conditional	$R[i] < V[j]\ ?\ R[i] \leftarrow -R[i]$: no-op

Table 2.1: Instruction used in this work's formulation of linear GP

The conditional operation has been described using the C-style ternary operator syntax for compactness. Note that the multiplication and division operations have

been formulated as unary doubling and halving operations, rather than as binary operations similar to addition and subtraction, as might be expected. This is intentional and part of an effort to ensure stability of the operations carried out during program execution. This is discussed further in section 4.1.

2.3 Precursors to Tangled Program Graphs

Tangled Program Graphs (TPG) is a genetic programming framework proposed by Kelly and Heywood [17] and developed for enabling the organization of programs into teams of programs and further into graphs of teams of programs. The underlying motivation being to support a process of emergent modularity, and therefore enable the search process to scale across high-dimensions and multiple tasks. The primary focus of this work is the extension of TPG to perform continuous actions. TPG emerged as a natural extension of an earlier genetic programming framework for the emergent discovery of teams of programs, Symbiotic Bid-Based programming (SBB); itself a natural extension of Bid-GP. The following sections describe Bid-GP and SBB, which provide the necessary foundation for understanding TPG.

2.3.1 Bid-GP

The following is a summary description of the Bid-GP algorithm, described by Lichodzijewski [20] [22], simplified so as to focus on the details relevant to Bid-GP's role as a precursor to SBB. Bid-GP was developed as a classification algorithm and will be described as such, with the bridge to reinforcement learning described later.

In the description of linear GP provided previously, it was noted that the solution is a single linear GP program evolved to perform the desired task. This makes the assumption that the task is 'doable' by a single linear GP program. This approach also discards all usefulness that may be found in the remaining agent population after the top-performing agent has been extracted. Bid-GP is an attempt to leverage the usefulness of populations of linear GP agents by having a population of agents collectively 'bid' for the opportunity to present themselves as the candidate solution. This allows a sub-set of programs to be identified, each with its own useful and potentially specialized behaviour. Under this approach, programs whose usefulness is only apparent in certain circumstances, such as for a particular subset of exemplars,

may be retained in the final solution and put to further use, despite possibly poor performance when confronted with exemplars outside of that set. This has the effect of lightening the load placed on any single agent and redistributes the work of classifying exemplars across an entire population.

In canonical linear GP, each program comprises a complete solution to the task at hand. During evolution of a solution to, say, classification problems, the output from canonical linear GP is a class label. It follows that fitness is then some overall measure of classification accuracy, which in turn is used to rank each program of the population. Classification is accomplished by interpreting a program's register values as confidence scores across classes. For example, in a multi-class classification problem with n classes (where $n < 8$), a high-level algorithm might be:

1. For each GP program:
   (a) For each exemplar in the dataset:
      i. Execute the program's instructions using the examplar as input.
      ii. Predict that the examplar is a member of class i where i is the index of the program register with the highest value.
      iii. Tally each correct prediction.
   (b) Assign the program a fitness equal to the ratio of correct predictions out of the total number of examplars.
2. Order the programs by their assigned fitness.
3. Delete some percentage of the least fit individuals and regenerate new ones from the remaining fit individuals.

In Bid-GP (as well as SBB and TPG), programs are modified to include a single scalar value selected at random during agent creation from the set of valid agent outputs. Let this scalar value be the action. In the case of a classifier, the set of valid outputs comprises the set of classes from which the classifier is making its prediction. This action remains with the agent for its lifetime, though the mutation operation during reproduction may alter this value in offspring of a given agent. Rather than formulating a program's output as an interpretation of its registers in terms of a

class label, each program's output is now the value stored in R[0], referred to as that agent's *bid*. Only, the program with the highest bid gets to provide its action for comparison against the true class label.

This means that in Bid-GP, programs are not expected to be able to classify all exemplars. Rather, in a classification problem, any given agent is only evaluated on its ability to correctly identify whether a given exemplar is or is not a member of the class assigned to that agent via its action. Under Bid-GP, classification is accomplished by allowing the entire population of programs to execute their instructions with a given exemplar as input. After all programs have completed instruction execution, their bids (the value stored in R[0]) are compared and the exemplar is predicted to be the class stored in the highest-bidding program's action. Each program is then acting as a binary classifier by asserting their level of confidence that a given exemplar is or is not a member of that program's assigned class.

This bidding mechanism has the dual effect of not only allowing a program to submit itself as a solution to the given exemplar, but also to bow out gracefully via a low bid if the agent is not well-suited to the given exemplar. This will, of course, typically happen when the program recognizes that a given exemplar is not of its assigned class. However, this also allows for the case where programs specialize in subsets of their assigned class. For example, in a classification task where each exemplar is a photograph of an animal and the task at hand is to identify is the animal is a dog, cat or bird, some programs may be highly successful in identifying tabby cats while others are successful at identifying Siamese cats. Though both sets of programs will be assigned the 'cat' class, the tabby identifiers can bow out with a low bid when presented with a Siamese cat, and vice versa. In this way, programs not only are freed from the burden of having to generalize across all tasks or classes, but they can further specialize within that class or task.

At each step of evolution the fitness of all programs is evaluated and the same removal and reproduction process described in the previous section on linear GP takes place such that the total population size remains constant. The mechanics of fitness assessment are not described here as they differ significantly from the assessment used for TPG agents in this work; it is adequate to say that programs score high fitness when they either bid highly when presented with their assigned class or bid

low when presented with a class outside of their assignment. After evolution, the entire population is itself considered a solution to the task being optimized. After evolution, the same bidding procedure is followed using the resulting population to classify new exemplars.

Lichodzijewski and Heywood found that Bid-GP significantly outperformed canonical linear GP in multiclass classification tasks. This result supports the approach of encouraging linear GP agents to specialize rather than generalize and to co-operate to form a complete solution. However, Bid-GP pre-defines a population size that is never varied. This means that the bidding process is broadcast across the entire population, with all members of the population assessed on each exemplar before the single program with largest bid value can be identified. Moreover, any children that represent degenerate behaviours (e.g. bid infinity on all exemplars) will need identification before performing fitness evaluation. Instead, it would be much more useful if a search for teams of programs could be performed at the same time, but independently of the search for useful (Bid-GP) programs.

2.3.2 Symbiotic Bid-Based Programming

Developed by Heywood and Lichodzijewski [23] [21], Symbiotic Bid-Based programming (SBB) is an extension of Bid-GP that addresses some of the shortcomings of Bid-GP identified above. SBB introduces an additional concept on top of the linear genetic program instance, the *team*. Teams are (variable length) vectors of references to programs, also referred to as *learners* in in both SBB and TPG. A complete solution to the problem at hand now takes the form of a team. This means that the search for good combinations of programs is explicitly defined by members of the team population. Should a degenerate program appear, it will only impact on the performance of the subset of teams in which it appears. That subset of teams will be penalized by a low fitness ranking. Selection will cull these teams, and any (Bid-GP) program not associated with a team will also be deleted. New teams and programs can then be introduced relative to the remaining (successful) team and program population content. Note that there is no attempt to propagate fitness as evaluated at the team level back to the level of programs. This is important as it helps mitigate

the 'relative overgeneralization' pathology [29][1].

Teams represent a variable length GP population whose content is defined in terms of the Bid-GP programs from the learner population, shown in figure 2.2. During evolution, reproduction and mutation operators are applied not just to individual learners but to the teams as well, with the effect that the algorithm searches for both useful learner behaviour as well as useful team sizes and arrangements. This section gives a general overview of SBB and conveys its central ideas as they apply to TPG. Specific implementation details can be found in Lichodzijewski and Heywood's original paper [23] but, like Bid-GP, are not provided here so as not to distract from the implementation of TPG.

SBB begins from the position that canonical linear GP and Bid-GP are designed primarily to search the space of useful linear GP agent behaviours. These algorithms also point to the limitations of individual GP agents, evidenced by the increased classification performance seen by Bid-GP over canonical linear GP. Finally, the two algorithms also suggest that the evolutionary algorithm applied in this way can search not just for general purpose classifier agents, but for agents that conform well to an enforced configuration, whether that be operation in isolation as in canonical linear GP or bidding behaviour within a population of co-operative agents as in Bid-GP. This suggests the configuration itself as the next target of evolutionary search operators. This is the standpoint from which SBB introduces the team as the searchable configuration space.

As described, teams are vectors of references to learners from within a population of learners. This is shown in figure 2.2. In the C programming language, this might be implemented as an array of pointers to linear GP agents. These learners keep the additional action value introduced in Bid-GP. The vectors are the candidate solutions to the problem to be optimized and act as direct stand-ins for the population of learners used in Bid-GP. Solution execution entails allowing all learners within a team to execute their instructions, each with the same exemplar or state as their input. The learner with the highest bid gets to put forth its action as the classification prediction or action. In fact, Bid-GP can be thought of as a limited instance of SBB in which

[1]Relative overgeneralization is when components of a solution receive an average reward over all the (team) interactions they participated in. This works against the development of specialized components, that might be very important, but only to a small number of teams.

there is only one team, i.e. equal to the population size.

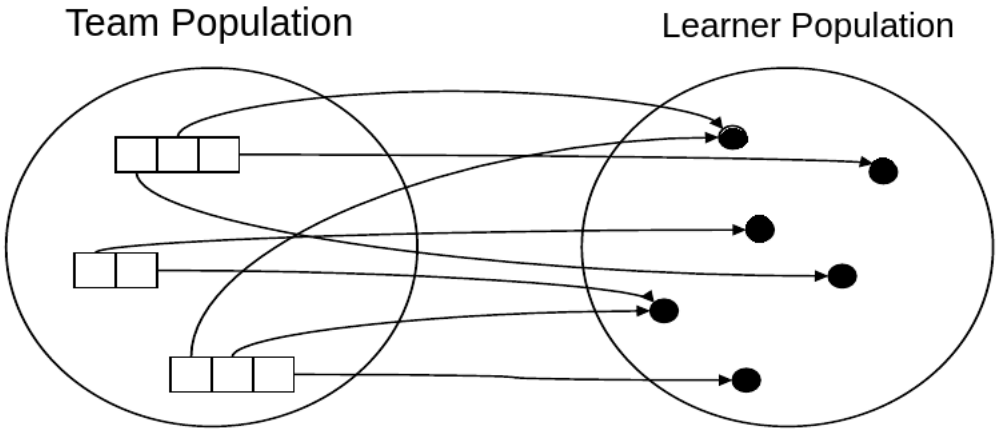

Figure 2.2: Team and learner population dynamics of SBB, borrowed and modified from Lichodzijewski [23].

SBB also introduces search operators that operate on the teams. Specifically, teams are kept relatively small; Lichodzijewski's implementation asserts a maximum size of 10 agents per team. In addition to the population of learners, SBB also maintains a population of teams. While canonical linear GP and Bid-GP's search operators act directly on the GP agents, SBB's search operations act directly on the teams which has a trickle-down effect to indirectly affecting their programs. What follows is a simplified description of the complete SBB algorithm described by Lichodzijewski [23]:

1. Initialize population of randomly-created teams:
   (a) For each new team, randomly generate 2 unique agents and store pointers to them in the vector of pointers (i.e. the team).
   (b) Add these agents to the agent population.
2. Until a candidate solution (i.e. team) meets some pre-defined fitness criteria:
   (a) Generate new teams from previous teams to fill the team population to its maximum size.
   (b) Evaluate all teams on the task at hand and assign each team a fitness value based on its performance (e.g accuracy in a classification task).

(c) Reduce the team population size by removing a pre-defined fraction of the least fit teams.

From this description it is apparent that this algorithm closely follows the algorithm used to search the space of programs, though in the case of SBB the search is performed over the space of teams. The search is still accomplished via mutation and reproduction, though of teams rather than learners. This occurs during the initialization and team generation procedures. At the end of initialization there is both a population of teams and a population of learners into which the teams point.

At each step of the evolution phase of the algorithm, teams and learners are subject to the reproduction and mutation operators that enable the search of both populations. New team generation at this phase is as follows:

1. From the remaining fit individuals, a team is selected at random.

2. This team is reproduced and a copy is added to the team population. This team holds identical pointers into the learner population as its parent.

3. The team is mutated:

   (a) Each learner in the team is considered and removed with uniform probability $p_r^{(i-1)}$ where i is the number of teams remaining (thus the probability of removing a learner decreases with each deleted learner). No team may reduce its size below two learners. It is important to note that the learners themselves are not deleted and remain in the learner population. The team simply deletes its pointer to that learner. This ensures that other teams that may be referencing that learner are unaffected.

   (b) Each learner in the population of learners is considered and a reference added to the team with uniform probability $p_a^{(i-1)}$ up to the maximum number of learners per team.

   (c) Each learner in the team is considered and mutated with some probability p_m via the mutation operations described in canonical linear GP. When mutating a learner, the following steps are taken to preserve the original learner in the event that other teams are also indexing it:

i. The learner is copied.

ii. The team in question's pointer to the original learner is re-assigned to point to the copy.

iii. The copy is then mutated.

After the team population has been filled back to its maximum capacity, the population of learners is reviewed and any learners to whom no teams are pointing are deleted from the population.

While standard linear GP and Bid-GP only facilitate direct search of the space of learners, the algorithm described above searches both the space of learners and configurations of learners. By maintaining a population of learners, SBB allows a library of learner behaviours to accumulate. The combined effect of steps 3(a) and 3(b) is that of searching through this library for learners that complement the team's current behaviour. Much like Bid-GP, SBB thus encourages the generation of specialist learners since all learners must act together to form a complete solution. Unlike Bid-GP, though, SBB's maximum program size and removal/re-adding steps provide a filtering mechanism to weed out degenerate learners. Moreover, should a learner appear in a 'wrong' team it is still free to 'hitchhike' to other teams as long as it is not explicitly detrimental to team fitness. In this way, SBB can retain the benefits of learner co-operation developed in Bid-GP while reducing needless computation.

Steps 3(a) and 3(b) alone are inadequate to search the space of solution candidates; these steps presume the existence of adequate learners in the learner population, which would otherwise be fixed at initialization, and are limited to searching the space of combinations of current learners. Through step 3(c), SBB expands this search to include the entire space of programs and teams. The learner space is searched via learner mutation. This lifts the restriction of searching only the space of combinations of existing learners, since the population of existing learners is now dynamic.

When compared to canonical linear GP, Lichodzijewski found that SBB displayed a definitive improvement over linear GP in classification accuracy over canonical linear GP under four classification tasks. Additionally, Lichodzijewski's analysis showed that learners under SBB do indeed 'specialize'. Lichodzijewski examined the learners in the final task solution candidates to determine into which features of the input space the learners indexed during execution of their instructions. For two of the

four datasets, it was shown that over 90% of the features indexed were indexed by only one learner within the team. This suggests that, while developing their bidding behaviour, programs learn to focus on a subspace within the exemplar space. Finally, Lichodzijewski also found that in most cases the number of effective instructions (i.e. only instructions that have an effect on the final bidding behaviour) for an SBB candidate solution was less than that for a competing lone canonical GP learner on the same task, despite the SBB candidate containing multiple learners itself. Not only does this translate to lower computational effort even compared to canonical GP, let alone Bid-GP, but it suggests that added constraints on a program (i.e. the requirement that a program generalize across all classes in a classification task rather than focus on one) lead to program bloat; the learner will be forced to execute programmatic contortions via a much larger list of instructions to produce a successful program. This extra complexity presumably translates to a more arduous search of the space of programs.

These findings strongly support the use of the teaming paradigm in linear GP. However, SBB still suffers from some limitations. Specifically, while SBB permits a search of learner configurations, these configurations have imposed on them both a maximum size and an assertion that the members of the configuration are necessarily all linear GP programs. While SBB proves the value of searching the space of configurations, it artificially limits scope of this space. These limitations are addressed by TPG, which is the primary focus of this work and is explained in the following section.

2.4 Tangled Program Graphs

Developed by Kelly and Heywood, Tangled Program Graphs (TPG) is a genetic programming-based reinforcement learning algorithm and as such represents a Monte Carlo approach to reinforcement learning, i.e. it is episodic and only updates after the interaction with the environment reaches a suitable end condition. It builds directly on top of SBB by incorporating the same teaming paradigm. However, while SBB is limited to constructing solutions in the form of a tree consisting of a single

node[2], TPG allows solutions to take the form of arbitrarily structured graphs. This is accomplished via a modification to the learner definition. In the algorithms discussed previously, learners have been made up of a list of sequential instructions, a bank of 8 floating-point registers, and a single scalar action value. If a learner wins a bid, either in the context of a population (e.g. Bid-GP) or team (e.g. SBB) they win the right to suggest their action value. TPG extends the learner definition such that the suggested action upon winning a bid may be a scalar action value *or* a reference to another team. When a learner in the latter case wins the bid, this pointed-to team then executes all of its learners as per SBB. This is shown in figure 2.3, taken and modified from Kelly [17]. Its highest-bidding learner puts forward its action, which is either the final scalar action value or a pointer to another team, and so on.

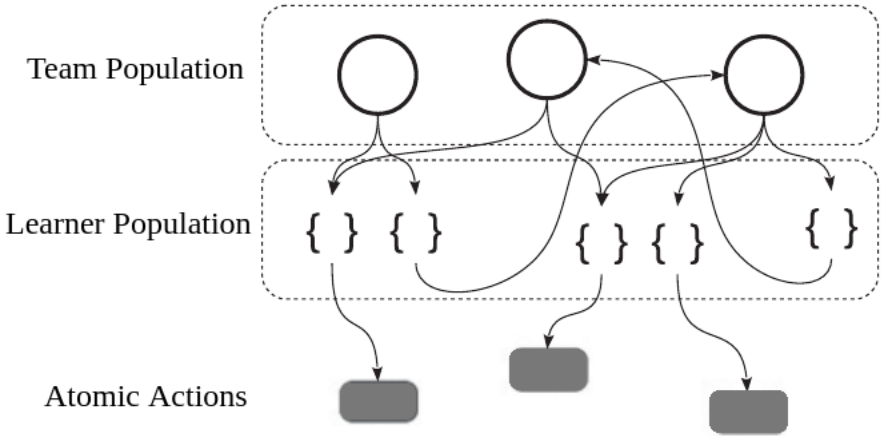

Figure 2.3: Team-to-team pointer dynamics of TPG, borrowed and modified from Kelly [17].

While graphs are not guaranteed to be acyclic, acyclic execution is enforced by disallowing any one team from being reached more than once during solution execution; if a team is encountered a second time during execution, the second highest-bidding learner puts forward its action instead. The process repeats until either a new team or a scalar action value are encountered. This is further guaranteed by enforcing the rule that each team must contain at least two scalar actions.

By allowing the formation of arbitrarily large graphs, TPG completes a research theme that began with the extension of single linear agents to populations of agents

[2]Extensions to equal depth trees were also pursued [11], but only by applying evolution in independent phases (one per tree level).

in Bid-GP: under the paradigm where the smallest unit of execution (a module) is the genetic program bidding to submit their solution to a given instance of the problem at hand (e.g. classification exemplar, state value, etc.) TPG allows the evolutionary algorithm complete freedom in designing both linear genetic program instances as well as arbitrarily-ordered hierarchical combinations of these instances, simultaneously leveraging the advances discovered in bid-based execution, populations, and teaming, while broadening the space of candidate solutions being searched to include solutions of arbitrary size and complexity.

TPG has proven itself to be highly capable of solving a wide range of reinforcement learning problems [17] and is the primary focus of this work. Indeed, TPG solutions to the Arcade Learning Environment [5] (ALE) visual reinforcement benchmark are competitive with a wide range of deep learning solutions [16], while producing solutions that are a fraction of the computational cost. What follows is a detailed description of TPG as implemented by the author and used in the experiments discussed here.

2.4.1 Linear Genetic Programs

Like Bid-GP and SBB, TPG uses linear genetic programs with the addition of a fixed action value as its fundamental unit of computation.[3] The linear genetic programs used in this work are identical to those described in section 2.2.2 in their instruction set, mutation operations, register definitions, and execution. TPG modifies these programs so that the action value decided at program creation may be either a discrete scalar action value from the set of valid (application specific) actions or a reference (or pointer) to a pre-existing team. Discrete scalar (application specific) action values under TPG are typically referred to as *atomic actions*.

TPG maintains a single population of learners into which teams may reference (or point). An additional learner component, introduced in SBB but first discussed here, is a counter owned by each learner. This counter tracks the number of teams that hold a reference to this learner. This lets learners track a crucial piece of their status within the learner population. Namely, this allows for the identification of orphaned learners, i.e. learners in the population to which no team is pointing. The learner

[3] As per Bid-GP and SBB, TPG does not require that these units must implement linear genetic programming; any genetic programming paradigm that supports mutation and reproduction operations and is capable of submitting a scalar bid upon execution may be used.

Parameters	Value
p_{add}	0.6
p_{del}	0.6
p_{mut}	0.6
$p_{mut.action}$	0.6
p_{atomic}	0.5
s_{min}	8
s_{max}	128
Number of registers	8

Table 2.2: Parameters used in this implementation of linear GP programs

population and the use of this counter is described later in section 2.4.3, in which the evolutionary algorithm is fully defined. Table 2.2 defines all parameters used in this implementation of the linear genetic programs.

Initialization

On creation of a new program, a number of initial instructions is chosen between and including s_{min} and s_{max} using a uniform random number generator. This number of instructions is then created by randomly generating mode (i.e. register or input), target index, source index and op code values. These are also generated using uniform random number generators from within the ranges described in section 2.2.2. These instructions are appended to a list owned by the genetic program.

In addition to its instructions, a new program also generates an atomic action that it will reference for its lifetime. This is created by sampling the set of atomic actions with uniform probability. During reproduction, offspring from this learner may generate a new action value that is either another atomic action or a reference to a team.

Mutation

Four mutation operations are defined on a linear genetic program instance: instruction addition, instruction removal, instruction modification, and action modification. The

addition and removal mutation operators, 'addRandomInstruction' and 'deleteRandomInstruction' stochastically remove randomly-selected instructions. The modification operators, 'mutateRandomInstruction' and 'randomlyModifyAction', stochastically modify randomly-selected instructions and potentially the learner's action respectively. The algorithms used all employ the 'weighted coin flip', defined in algorithm 1.

Algorithm 1: weightedCoinFlip

Input: Probability p

Output: True or False

$r \leftarrow$ number generated from URNG between between [0.0, 1.0]

if $r < p$ **then**
    return true

else
    return false

In these algorithms, uniform random number generators are referred to as URNGs. The program instance's current list of instruction is referred to as 'instructions'. This structure is and array that is assumed to have insertion and deletion operators 'insert' and 'delete' defined on it that can insert a new element or delete an element, respectively, at a certain index, as well as a 'length' operation that returns the list's length. The four operations are defined below in algorithms 2 to 5.

Algorithm 2: addRandomInstruction

if *weightedCoinFlip*(p_{add}) *and instructions.length* $< s_{max}$ **then**
    inst $\leftarrow$ randomly generated instruction
    index $\leftarrow$ index generated from URNG between [0, instructions.length)
    instructions.insert(inst, index)

Algorithm 3: deleteRandomInstruction

if *weightedCoinFlip*(p_{del}) *and instructions.length* $> s_{min}$ **then**
    index $\leftarrow$ index generated from URNG between [0, instructions.length)
    instructions.delete(index)

Algorithm 4: mutateRandomInstruction

```
if weightedCoinFlip(p_mut) then
    index ← index generated from URNG between [0, instructions.length)
    component ← uniform random selection from [MODE, TARGET,
     SOURCE, OPCODE]
    bounds ← 0
    if component == MODE then
        bounds ← number of instruction modes
    else if component == TARGET then
        bounds ← number of registers
    else if component == SOURCE then
        bounds ← length of input vector
    else
        bounds ← number of possible opcodes
    value ← integer generated from URNG between [0, bounds)
    while value == instructions[index][component] do
        value ← integer generated from URNG between [0, bounds)
    instructions[index][component] ← value
```

where

- 'number of instruction modes' is always 2 in this implementation, since an instruction can be a register mode instruction (i.e. the second or primary operand is indexed from the program's registers) or an input mode instruction (i.e. the second or primary operand is indexed from the input vector) The 'MODE' component is a flag indicating the type of instruction.
- 'number of registers' is always 8 in this implementation.
- 'length of input vector' is dependent on the environment or task at hand.
- 'number of possible opcodes' is 6 for this implementation (though this changes if memory operations are introduced, as described later in section 2.4.4).

The final while-loop ensures that mutation does indeed create a new instruction, even in the event that the first new component created would actually result in no change to the instruction.

Algorithm 5: randomlyModifyAction

```
Input: Probability p_atomic
if weightedCoinFlip(p_mut_action) then
    action ← nil
    if weightedCoinFlip(p_atomic) then
        action ← atomic action sampled with uniform probability from the set
          of atomic actions
    else
        action ← reference to team selected with uniform probability from
          team population
    program.action ← action
```

Note that while p_{atomic} typically takes the value defined in table 2.2, it may optionally be set to 1.0, i.e. the new action is guaranteed to be atomic. This is done when that learner is the last learner in the team whose action value is atomic. Since all teams must contain at least one learner whose action is atomic this learner is not allowed to convert its action to a team reference.

The full mutation operation is implemented in the learners 'mutate' operation which simply calls the previous functions in sequence, shown in algorithm 6.

Algorithm 6: mutate

```
Input: Probability p_atomic
addRandomInstruction()
deleteRandomInstruction()
mutateRandomInstruction()
radomlyModifyAction(p_atomic)
```

Note that the final mutation operation takes an argument. This facilitates setting p_{atomic} to 1.0.

2.4.2 Teams

The fundamental unit of configuration in TPG is the team. In general, the implementation of teams under TPG is as described for SBB in section 2.3.2, where a team

Parameters	Value
p_{add}	0.7
p_{del}	0.7
p_{mut}	0.3
s_{max}	8

Table 2.3: Parameters that define team-specific evolution dynamics in TPG.

is a vector of references to learners that implements mutation and reproduction operations, learner addition and removal operations, and a bidding/action mechanism. These operations are fully described below.

As it does for learners, TPG maintains a single population of teams. As described in the introduction to TPG in section 2.4 these teams may, via one or more of their learners, point to other teams within the population. Correspondingly, they may be pointed to by any other team in this population. This introduces the concept of *root teams*: a root team is any team to which no other teams point. These root teams provide the entry point for program execution and so define a candidate solution/agent: for a given input, a solution is exercised by executing the root team's learners on that input and taking the action suggested by the highest-bidding learner. This may then be a pointer to another team in which case the process is repeated. This process is described fully below. In this implementation of TPG, teams own a counter that tracks the number of learners pointing to this team. This counter is maintained during evolution and allows for a team's status as root or non-root to be polled directly.

Table 2.3 defines several parameters that determine the evolution dynamics of teams under this implementation of TPG. Most of these parameters will be explained as they appear in the mutation algorithms described below. The parameter s_{max} refers to the maximum team size, i.e. the maximum number of learners allowed per team. The following sections describe team-specific operations, including mutation, reproduction and learner addition and removal.

Initialization

Unlike in the case of learners, which have a high level of autonomy during their self-initialization procedure, the initialization of teams in TPG is handled primarily by

the evolutionary algorithm itself since team creation requires knowledge of the learner population. On creation of a new team, the team generates a maximum size using a uniform random number generator that selects an integer value between and including 2 and s_{max}. This value is never altered and the team enforces that maximum size during all future operations. The team's complement of learners is managed by the evolutionary algorithm through the team's learner addition and removal operations.

Mutation

Teams implement operations for both adding or removing a learner that exists in the population to its complement, and three mutation operations that are built on these addition and removal algorithms. All are described below. As in the description of linear genetic programs, these algorithms utilize the 'weightedCoinFlip' algorithm. The algorithms below reference several variables that are presumed to exist already:

- 'team.learners' refers to the team's vector of references to learners. This vector implements the 'append' function which appends an element to the vector, the 'remove' function which deletes an element from a vector that contains it, and exposes a 'length' variable that reports the number of learners in the vector.
- 'team.max' is the maximum number of learners allowed in the team, established at team creation.
- 'team.num_atomic_actions' is a counter that tracks the number of learners in the team's complement whose action is atomic.
- 'learners' (note the absence of the prefixed 'team') refers to a global vector of learners the comprises the learner population, described in section 2.4.3.
- 'teams' refers to a global vector of teams that comprises the team population, described in section 2.4.3.
- 'learner.num_referencing_teams' is the counter tracking the number of teams pointing to this learner.
- 'learner.atomic' is a boolean flag that is true if the learner's action is atomic.

The algorithms that define mutation at the team level are described below.

Algorithm 7: addSingleLearnerToTeam

Input: team, learner

if *learner is in team.learners* **then**
  return
learner.num_referencing_teams ← learner.num_referencing_teams + 1
team.learners.append(reference to learner)

The sole purpose of algorithm 7 is to add a learner to the team's vector of learners and perform any associated book-keeping. It is not a mutation operation on its own.

Algorithm 8: removeSingleLearnerFromTeam

Input: team, learner

if *learner is not in team.learners* **then**
  return
learner.num_referencing_teams ← learner.num_referencing_teams - 1
team.learners.remove(reference to learner)

The sole purpose of algorithm 8 is to remove a learner from a team's vector of learners and perform any associated book-keeping. It is not a mutation operation on its own.

Algorithm 9: randomlyAddLearners

Input: team

$p \leftarrow p_{add}$
while $weightedCoinFlip(p)$ **do**
  $p \leftarrow p * p_{add}$
  if *team.learners.length == team.max* **then**
    return
  candidates ← learner population minus learners that already exist in this team's complement and learners whose action is a pointer to this team
  new_learner ← learner selected with uniform probability from candidates
  addSingleLearnerToTeam(team, new_learner)

Algorithm 9 is a mutation operation. It adds some number of new learners from the learner population to the team's complement. Due to its stochastic nature, this function will not necessarily add a new learner every time it is called. The statement $p \leftarrow p * p_{add}$ ensures that the probability of adding a new learner decreases on each iteration of the loop. A similar mechanism is used in learner removal and learner mutation.

Algorithm 10: randomlyRemoveLearners

```
Input: team
p ← p_del
while weightedCoinFlip(p) do
    p ← p * p_del
    if team.learners.length == 2 then
        return
    if team.num_atomic_actions > 1 then
        candidates ← team.learners
    else
        candidates ← team.learners minus the remaining learning with an
          atomic action
    learner_to_delete ← learner selected with uniform probability from
      candidates
    removeSingleLearnerFromTeam(team, learner_to_delete)
```

Algorithm 10 is a mutation operation. It stochastically removes existing learners from the team's complement. The learner is not deleted from the learner population. This encourages other teams to add this learner to their own complement. Due to its stochastic nature, this function will not necessarily remove a learner every time it is called. The statement $p \leftarrow p * p_{add}$ ensures that the probability of adding a new learner decreases on each iteration of the loop. A similar mechanism is used in learner removal and learner mutation.

Algorithm 11: randomlyMutateLearners

```
Input: team
for learner in team.learners do
    if weightedCoinFlip(p_mut) == false then
        return
    p ← p_atomic
    if team.num_atomic_actions == 1 and learner.atomic then
        p ← 1.0
    new_learner ← copy of learner
    new_learner.mutate(p)
    removeSingleLearnerFromTeam(team, learner)
    addSingleLearnerToTeam(team, new_learner)
    learner.append(new_learner)
```

Algorithm 11 is a mutation operation. It stochastically selects some number of learners from the team's complement and invokes the learner's own mutation operation. All mutations are performed on copies of the learner so that the original is always preserved in the learner population. This ensures that other teams referencing this learner are not disrupted. Due its stochastic nature, this function will not necessarily mutate a learner every time it is called.

Algorithm 12: mutate

Input: team

removeLearner(team)

addLearner(team)

mutateLearners(team)

Teams implement the "mutate" operation shown in algorithm 12 which calls each of the previous mutation operations in succession.

Program Execution (Bidding and Action Selection)

Action selection at the team level is performed as follows:

1. For each learner in the team's learners:
   (a) Execute that learner's instructions on the current state vector.
   (b) Identify the learner with the highest value in register 0.
   (c) If the learner's action is atomic, return that action.
   (d) Else, if the learner's action is a reference to a team repeat these steps using the new team's learners.

Action selection at the solution level is identical with the caveats that the first team whose learners are invoked is a root team, and that no team may be invoked twice. This results in the following modified steps:

1. Identify the root team corresponding to the solution to be executed.
2. Create an empty set of teams, representing teams that have been visited during execution.
3. Execute the root team (i.e. perform action selection with this team).

Parameters	Value
r_{size}	200
$r_{episodes}$	Variable
$r_{generations}$	Variable
r_{gap}	0.3

Table 2.4: Parameters used by TPG's evolutionary algorithm.

4. If the learner's action is atomic, return that action.

5. Else, if the learner's action is a reference to another team:

   (a) Add the current team to the set of visited teams.

   (b) If the pointed-to next team is *not* in the set of visited teams, repeat steps 3 to 5 with the new team's learners.

   (c) Else, if the pointed-to team has been visited already, identify the current team's learner with the next highest value in register 0 and repeat steps 4 and 5.

The set of visited teams prevents cycles during execution, while the assertion that each team must contain at least one learner whose action is atomic ensures that the algorithm will terminate and return an atomic action.

2.4.3 Evolutionary Algorithm

The evolutionary algorithm used to generate a solution is, at a high level, equivalent to the algorithm described in section 2.2.1, modified to use TPG-specific implementations of solution candidates, candidate reproduction, and candidate mutation. The evolutionary algorithm implemented for this work does differ in one aspect: rather than halting evolution after a candidate is discovered that achieves some pre-defined desired level of fitness, TPG halts evolution after a pre-defined number of generations (iterations of selection, reproduction and mutation).

TPG evolution is configured through the setting of several parameters, described in table 2.4. The parameter r_{size} refers to the number of root teams enforced at the beginning of every generation, and r_{gap} is the fraction of root teams remaining after selection is performed. The number of root teams comprise the population of

solution candidates and emerges as a property of the current team population; because team mutation can add or remove connections to and from other teams within the population, the number of root teams is variable after mutation is performed and must be re-discovered at the beginning of each new generation.

The parameter $r_{generations}$ refers to the number of generations during evolution, i.e. the number of iterations of candidate selection, reproduction and mutation. The table above lists this as 'Variable': this refers to the fact that different experiments in this work modified this parameter differently; however, for a given experiment and execution of the evolutionary algorithm, this value is fixed.

The parameter $r_{episodes}$ is introduced in this implementation of TPG. This parameter is specific to TPG's use as a reinforcement learning algorithm in episodic environments. At each generation, the fitness of a solution is assessed by executing the solution in an environment that directly assigns a score to the agent at the completion of an episode and using that score as the candidate's fitness. In the environment used in the work, higher score means higher fitness, i.e. better performance. Because the starting configuration of the environment is randomized and because some starting configurations are more difficult than others, $r_{episodes}$ was introduced to provide a better representation of a candidates's fitness: rather than evaluating the agent over a single episode, the candidate is evaluated over $r_{episode}$ episodes and their fitness is taken to be the mean of all scores received. The high-level steps for the evolutionary algorithm are as follows:

1. Initialize the team population and learner population.
2. Repeat the following steps $r_{generation}$ times:
    (a) Generate new root teams using the team reproduction and mutation operations until there are r_{size} root teams.
    (b) Evaluate each root team over $r_{episode}$ episodes and assign each one a fitness value.
    (c) Select and remove the $(1.0 - r_{gap}) * r_{size}$ least fit root teams.
    (d) Select and delete any learners in the learner population that are no longer referenced by any teams.

After $r_{generation}$ iterations, the root team with the highest fitness is taken to be the solution to the environment.

Initialization

Initialization (step 1 in the previous section) can be broken into two components: the initialization procedure for a new team, and the initialization procedure for the team population. These procedures are described below. Both presume the existence of both a team and learner population. These are global vectors available during evolution. The following steps describe the initialization of an *individual team*:

1. An empty team structure (i.e. a team with a zero complement of learners) is created.
2. Two unique learners are randomly initialized, as described in section 2.4.1. Note that the action of a newly-created learner is necessarily atomic.
3. References to these learners are added to the team via its 'addSingleLearnerToTeam' procedure (algorithm 7).
4. These learners are also added to the learner population.
5. The team is added to the team population.

The following steps describe the initialization of the *team population*:

1. Repeat for $r_{size} * r_{gap}$ iterations:
   (a) Perform the individual team initialization procedure described previously. At this stage the team population contains $r_{size} * r_{gap}$ teams, each with two unique learners whose actions are atomic.
   2. For each team in the team population:
   (a) While the current team's size is less than its maximum size:
      i. Randomly select a learner with uniform probability from the learner population. If this learner is already in the team's complement, continue selecting learners until one is found that is not in the team's complement.

ii. Add a reference to this learner to the team using the 'addSingleLearnerToTeam' procedure.

After these steps are complete, the team population is made up of $r_{size} * r_{gap}$ teams, each with a full complement of learners, all of whose actions are atomic. Note that at this stage, all teams are root teams. The learner population holds $2 * r_{size} * r_{gap}$ learners whose actions are all atomic.

Generation

Generation of new root teams at each generation is accomplished via the reproduction and mutations operations. At the end of the reproduction step, the team population is guaranteed to contain at least r_{size} root teams (as well as any number of non-root teams). The generation procedure is as follows:

1. While the number of root teams is less than r_{size}:
   (a) Randomly select a team to reproduce.
   (b) Create a new empty team structure.
   (c) Assign the new team the same maximum size as the team being reproduced.
   (d) Copy the vector of references to learners from the reproducing team to the new team. This is accomplished via the 'addSingleLearnerToTeam' operation (algorithm 7), performed on the new team, so that each learner's count of referencing teams is incremented appropriately.
   (e) Invoke the 'mutate' operation on the team (algorithm 12). Note that because this team may include a learner which is the only learner pointing to another team, mutation of these learners may result in the mutation of that pointer into either a pointer to another root team, a pointer to another non-root team, or an atomic action. This means that there are no guarantees about the effect of a single iteration of the generation loop on the total number of root teams, and so the number of root teams must be continually re-calculated at step 1.

(f) Add the team to the team population.

Note that any team created this way is a root team, regardless of whether the original reproducing team was a root team. This is due to the fact that on creation, a team can not be pointed to by any learners.

Selection

Selection is performed on both the root team and learner populations. It is performed after all root teams have been evaluated in the current environment and assigned a fitness.

After evaluation, the root teams are sorted by their fitness. The $r_{size} * r_{gap}$ root teams are left unchanged. All others are deleted. The following steps describe the process of deleting a root team:

1. All learners are removed from the team via the 'removeSingleLearnerFromTeam' operation. Because this operation decrements the learner's count of referencing teams, this may result in learners to which no team is pointing.

2. The team is deleted completely from the team population.

After root team selection, there may be learners in the learner population to which no team is pointing, referred to as *orphaned learners*. These are deleted via the following procedure:

1. All orphaned learners are checked to determine if they contain one or more pointers to other teams. These teams' referencing learner counters are then decremented in anticipation of the removal of the learner. Note that this may create new root teams.

2. All orphaned learners are deleted completely from the learner population.

2.4.4 TPG Memory

The implementation of TPG described in the previous sections will be referred to as canonical TPG in this work and closely follows implementations by Kelly [17] (with the notable difference that this work implements a different instruction set in its linear

GP instances; this is discussed further in section 3.2). However, since the original successes of TPG, the algorithm has been expanded by Smith and Heywood [30] [31] to include a mechanism that acts as both a long- and short-term memory. Smith and Heywood have successfully used this mechanism to produce TPG agents capable of some degree of long-term planning as well as acting in environments with only partial observability. This mechanism is used in parts of this work, although not necessarily as a memory device (as discussed in section 3.2).

The mechanism introduced by Smith and Heywood is comprised of an additional single $n \times r$ matrix, where r is the number of registers used by linear GP agents (8 in canonical TPG, TPG with memory, and this work's implementation of TPG) and n has historically been set to 100. In addition to this matrix, two new op-codes were introduced to the instruction set: a write operation that probabilistically copies values from a learner's registers into the memory matrix, and a read operation that directly pulls a value out of memory and stores the value in a learner's register. The memory matrix itself represents a single instance of a global structure. Thus, it is available not only to all learners within a given TPG solution, but to all learners across all candidate TPG solutions that are present during evolution. In addition to providing TPG solutions with memory, this global availability has three major effects:

- In allowing all learners across a single TPG solution access to the same (global memory) matrix, all learners in a solution are provided with a mechanism through which they can communicate with each other. This means that individual learners have the potential to participate in and become aware of the global state of TPG

- By allowing all learners across all TPG solutions access to the same matrix, learners are encouraged to arrive at a consistent use and interpretation of what information is stored where in the memory matrix. This is significant because learners/teams are essentially independent at initialization. By providing a single global instance of memory, Smith encourages learners/teams to treat the memory as a commons. If specific learner's 'pollute' the commons or do not have a common view as to where short and long term memory are, then they will develop incompatible assumptions about how to use memory. As teams/learners

become integrated into a larger TPG graph, the use of a single instance of global memory helps ensure compatibility between different teams/learners as the TPG graph emerges.

- Global memory is never reset. When one agent completes evaluation, the state that global memory is in will be inherited by the next agent. This represents a further mechanism by which 'respect for the commons' is encouraged. This does not preclude different uses, thus some TPG teams/learners might concentrate on long term memory content, others short term, and indeed only specific learners emerge that perform write operations [31].

The mechanism described above is used in this work in general, though its exact formulation is slightly modified. These modifications are discussed in section 3.2.2 in which the memory matrix's use in this work is defined. Implementation details for the memory matrix as used by Smith and Heywood are described below.

Write Operation

As described, the write operation is probabilistic and is implemented as an additional op-code and instruction in its own right. The algorithm for a single write operation is described in algorithm 13. The operation is defined on the program data structure and assumes the existence of the following variables:

- A global instance of the memory matrix, referred to as 'mem' and indexed as 'mem[row][column]', which defines a 'rows' property equal to the number of rows in the matrix and a 'columns' property equal to the number of columns in the matrix.

- An array 'registers' which is owned by the program and comprises the array of 8 doubles that make up a program's registers

Algorithm 13: writeOperation

Input: learner

for *row in mem.rows / 2* **do**
- r1 ← mem.rows/2 - row
- r2 ← mem.rows/2 + 1 + row
- p ← $0.25 - (0.01 * row)^2$
- **for** *column in mem.columns* **do**
  - **if** *weightedCoinFlip*(*p*) **then**
    - mem[r1][column] ← learner.registers[column]
  - **if** *weightedCoinFlip*(*p*) **then**
    - mem[r2][column] ← learner.registers[column]

The outer loop iterates over the rows of of the memory matrix two at a time from the middle outward with the probability decreasing quadratically toward the first and final rows. This is shown in figure 2.4.

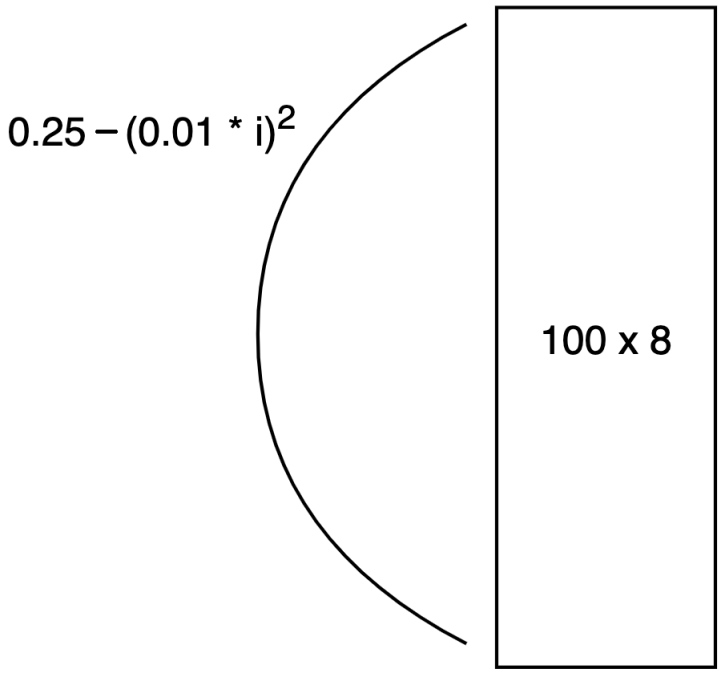

Figure 2.4: Graphical representation of memory matrix and probabalistic write access.

The inner loop then iterates over each entry in the two current rows in turn and optionally writes the corresponding register value to that location. Using a probabilistic curve as shown in figure 2.4 is intended to encourage programs to derive the

concepts of long- and short-term memory: values stored in rows nearer to the middle will be frequently over-written and so are only useful for short-term storage, whereas values stored in rows nearer to the ends will be over-written less frequently and so will remain accessible for a longer period.

Read Operation

The read operation is not a probabilistic operation, but rather a deterministic unary operation. It takes a source index which is here used to index into the memory matrix. The instruction is implemented as:

Algorithm 14: readOperation

Input: learner, i, j

$j \leftarrow j \mod (n * r)$

$row \leftarrow floor(j/r)$

$column \leftarrow floor(j \mod n)$

learner.registers[i] $\leftarrow$ memory[row][column]

where 'i' and 'j' are the target and source indices described in section 2.2.2 on linear GP.

2.5 Gradient-Based Reinforcement Learning

Previous sections focused primarily on the genetic programming paradigm and its application to reinforcement learning. However, this work makes heavy use of another reinforcement learning algorithm, REINFORCE [34] [33] [32]. REINFORCE is a Monte Carlo gradient-based learning algorithm that can produce arbitrary action types, from discrete (atomic) actions from within a pre-defined set, to real-valued actions.

While this work is singularly concerned with the expansion of TPG to real-valued actions, REINFORCE plays a significant role in that expansion. Being a gradient-based approach, the methodology behind REINFORCE differs substantially from the genetic programming paradigm discussed so far. The following sections provide a brief overview of the workings of gradient-based methods and the REINFORCE algorithm itself.

2.5.1 The Policy Gradient Theorem

Gradient-based reinforcement learning methods formulate their policy as a differentiable probability distribution over actions a, conditioned on states s, and parameterized over some collection of weights θ. The policy is represented as:

$$\pi(a|s,\theta)$$

where the result is the probability of taking action a under the current conditions defined by s and θ.

Williams [34] defines a family of reinforcement learning algorithms, which he names REINFORCE algorithms, that formulate their policies as described above and use the gradient of this formulation to perform policy updates via the weights θ. Sutton et. al. formalized this approach and defined the policy gradient theorem [33], which states that:

$$\nabla J(\theta) \propto \sum_s \mu_\pi(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$

where

- $J(\theta)$ is a scalar performance measure of the current policy
- $\mu_\pi(s)$ is the distribution of states s over policy π
- $q_\pi(s,a)$ is the action value function evaluated at state s and action a. This is the estimated return G (see section 2.1.4) if action a is taken. This is described further in Sutton and Barto [32].
- $\pi(a|s,\theta)$ is the policy, evaluated at state s and action a, and parameterized by θ

This definition of the policy gradient theorem is the same formulation that appears and is proved in Sutton and Barto [32]. The text also presents the REINFORCE algorithm. Incidentally, REINFORCE is presented in Sutton and Barto as a single algorithm rather than a family. This algorithm does indeed fit into Williams' family definition and can be seen as a representative instance of this family. In this work, 'REINFORCE' refers to the specific implementation found in Sutton and Barto, which was re-implemented and used for these experiments.

The policy gradient theorem states that for several formulations of a performance metric $J(\theta)$, i.e. a metric that provides a quantitative indication of how well or poorly the policy is performing, the gradient of this performance metric, which typically cannot be determined in practice, is proportional to the gradient of the policy itself, which often can be determined or approximated in practice. REINFORCE algorithms take advantage of this statement by following the gradient of the policy itself in order to incrementally update their policy and optimize this performance metric. In practice, the performance metric function is not actually defined; it is adequate just to have a method for approximating its gradient.

The policy gradient as it is defined above is not immediately useful. It requires knowledge of $\mu_\pi(s)$ and $q_\pi(s, a)$, neither of which can typically be determined in real-world applications. In order to make use of the policy gradient theorem it is usually reduced to the following equivalent formulation:

$$\nabla J(\theta) \propto E[G_t \nabla \ln \pi(A_t|S_t, \theta)]$$

This formulation reduces the policy gradient theorem from a general formula to a function of random variables that may be sampled in practice to approximate $\nabla J(\theta)$. The policy

$$\ln \pi(A_t|S_t, \theta)$$

can be implemented as a differentiable linear function or neural network using autodifferentiation libraries such as TensorFlow [24] or Torch [9] making this formulation tractable in practical implementations of REINFORCE.

2.5.2 REINFORCE

REINFORCE algorithms as defined by Williams [34] are those that follow the update rule

$$\theta \leftarrow \theta + \alpha \nabla J(\theta)$$

or

$$\theta \leftarrow \theta + \alpha G_t \nabla \ln \pi(A_t|S_t, \theta)$$

Sutton and Barto more specifically define REINFORCE as a complete Monte Carlo reinforcement learning algorithm, re-printed here in listing 15 from Sutton and Barto [32]. This listing shows that this formulation of REINFORCE uses the update rule described by Williams.

Algorithm 15: REINFORCE

Input: A differentiable policy parameterization $\pi(a|s,\theta)$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^d$

for *each episode* **do**

  Generate an episode $S_0, A_0, R_1, ...S_{T-1}, A_{T-1}, R_T,\ following\pi(\cdot|\cdot,\theta)$

  for *each step of the episode* $t = 0, 1, ..., T-1$ **do**

    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla ln\pi(A_t|S_t,\theta)$

As described in the previous section, practical use of the policy gradient theorem requires that G_t and $\ln \pi(A_t|S_t)$ be sampled in order to perform policy updates. In this formulation of REINFORCE, entire episodes are allowed to play out under a given policy $\pi(A_t|S_t,\theta)$ during which G_t and $\nabla \ln \pi(A_t|S_t,\theta)$ are sampled at each timestep. After an episode terminates, the weights θ may be updated according to the rule defined above. The outer loop may be terminated after either a fixed number of episodes or when the policy achieves some pre-determined minimum performance requirement. In this work, REINFORCE was set to run for a fixed number of episodes, though this number varied across experiments.

Algorithm 15 modifies this update rule to include the *discount factor* γ. This is a scaling factor added to reduce the value of rewards received at timesteps $t+T$ when viewed at time t, where T is any integer number of timesteps. This is incidental to this work; more details may be found in Sutton and Barto [32].

2.5.3 Action Formulations

REINFORCE requires that the policy used is differentiable and parameterized over some weight collection θ. As long as these requirements are met, REINFORCE allows for a variety of policy configurations which in turn allow for a variety of action formulations. This includes real-valued actions, which are the focus of this work.

Rather than learning a probability distribution over a finite set of discrete actions, REINFORCE may instead learn statistics about a continuous probability distribution from which real-valued actions may be sampled. As described in Sutton and Barto [32], this is achieved by re-formulating the policy as a normal distribution

$$\pi(a|s,\theta) \equiv \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2})$$

with mean $\mu(s,\theta)$ and standard deviation $\sigma(s,\theta)$ parameterized over the collection of weights θ. Because this is still in accordance with the policy gradient theorem as described previously, the same algorithm may be used to generate policy updates. Whereas previously, updates had the effect of modifying the relative probabilities of each action, updates now modify a single normal distribution from which new actions are sampled.

In this work, the simplification was made that $\sigma(s,\theta)$ was set to a fixed value of 0.1 and only the mean $\mu(s,\theta)$ was parameterized on θ and updated. This simplified the implementation without hindering the algorithm. It also avoided errors seen in early experiments where both the mean and standard deviation were parameterized functions as described above: because there is nothing under this formulation that directly enforces a minimum standard deviation (i.e. exploration is not directly encouraged), degenerate behaviour could often drive $\sigma(s,\theta)$ to output 0.0 or nearly 0.0, resulting in divide-by-zero errors.

2.6 CartPole Environment

This work uses only a single reinforcement learning environment, commonly known as the CartPole environment [4]. This environment was selected because it is a widely-known benchmark that can also be readily expressed in both a discrete action and a real-valued action without changing any other property of the task. This environment is one of several that has been re-implemented in Python and made available through Python's OpenAI Gym library [8].

CartPole is a 2D environment in which the task at hand is to learn how to balance a pole that is attached to a moving cart at a single pivot point. The OpenAI Gym library optionally allows the environment to be rendered. Figure 2.5 shows a screen shot taken from this render.

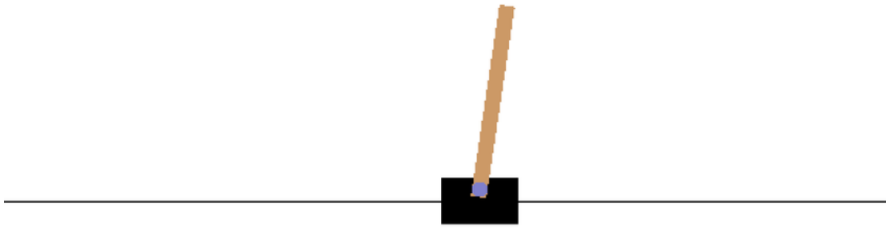

Figure 2.5: Sample rendering of the CartPole environment [8].

The cart has full freedom to slide along the track from side to side and the pole can rotate freely. Both act according to classical physics and the pole will fall to one side or the other if the cart is not acted upon. Upon resetting the environment, the cart and pole are randomly positioned within a range such that the environment can possibly be solved, i.e. the cart and pole cannot start such that failure is guaranteed. At every timestep, the state presented by the environment, $\vec{s}(t)$, is a vector of four floating-point values indicating the cart's position along the track (x), the cart's sideways velocity ($\dot{x}$), the angle between the pole and cart (θ), and the pole's angular velocity ($\dot{\theta}$).

In its classic formulation, the set of valid actions includes a discrete push of the cart to the left or a discrete push of the cart to the right. Taking no action at all is not a valid action in this environment. A discrete instance of a push applies 10 Newtons of force to the cart. At every timestep, the agent receives a reward of +1. The episode ends when:

- The agent has received a cumulative return of 500
- The angle between the pole and cart drops below 12 degrees, i.e. the pole falls over
- The cart reaches the boundaries of the viewing area

The goal of the learning agent is to maximize the cumulative return.

2.6.1 Continuous Actions

This work used both the classical environment that accepts discrete actions from the set { left push (-10.0N), right push (+10.0N) } as well as a custom modified version that accepts real-valued actions in the range [-1.0, 1.0]. The push applied to the cart is then the real-valued action multiplied by 50.0N. All other behaviour is the same.

Chapter 3

Motivation and Approach

As currently implemented, TPG inherently supports only discrete actions, which significantly limits the scope of environments to which TPG can be applied. This limitation is the result of actions taking the form of the fixed scalar action value assigned to each learner on creation and possibly modified via mutation, where this scalar value is sampled from the set of the environment's valid atomic actions. This effectively assigns each learner a 'type' and so promotes specialization, as discussed in section 2.3.1 on Bid-GP. For example, in the CartPole environment, this promotes the creation of learners who learn when to assert a push to the left and those who learn when to assert a push to the right.

This limitation reduces TPG's usefulness in practical applications. Many real-world environments in which a learning algorithm may be applied do not lend themselves to discrete action definitions. For example, a self-driving car necessarily needs to learn concepts such as: how far to turn the steering wheel, how much acceleration to apply, and how hard to apply the brakes. As mentioned in section 2.1 on reinforcement learning, a policy that provides control of a robotic arm must learn concepts such as how much pressure to apply during a grasping action, as well as how many degrees to rotate each joint. All of these examples are best represented as real-valued actions and so in both of these examples, TPG would not be applicable. There is therefore strong motivation to introduce a new action mechanism to TPG that would allow it to be applied to a broader variety of reinforcement learning tasks.

3.1 Obstacles in Extending TPG's Action Mechanism

Given that the limitation to discrete actions is the bidding/action formulation described in previous sections and above, it may appear that the path to real-valued actions lies in the reformulation of this mechanism. Several candidate reformulations that may appear obvious at first are discussed below, as well as the drawbacks to each

that have thus far led to them being left unexplored. This is presented to further motivate the path chosen in this work.

The first possible approach to modifying the bidding/action mechanism to produce real-valued actions is dropping the assigned scalar action entirely and using the value in the winning learner's R[0] as the real-valued action. This method would seem to be hinted at in the original formulation of standard linear genetic programming, in which the output is indeed directly interpreted from the learner's register values. However, the introduction of populations and/or teams to the evolution and solution formulation precludes this approach. In canonical linear GP, there is no notion of competition applied to the values in the learners' registers. However, in population and teaming approaches, these values are used directly to define a hierarchy of learners, where the register values correspond to an agent's self-proclaimed usefulness. Thus, during evolution, learners can separate the tasks of learning to operate in the environment, i.e. learning what action to apply, with learning when to assert their action, i.e. bidding behaviour. However, by conflating the bid value and action value, learners can no longer separate the tasks of learning what to do with learning when to do it: to successfully assert yourself as the most useful learner, you must also suggest a high-valued action. This approach is highly unlikely to produce successful solutions.

To address the conflation of bid values with action values, a possible modification of the previous approach would be to continue to use register 0 as the bid value, while using another register as the action value. Indeed, the separation of bid from action would likely lead to improved performance over the first approach. However, a similar conflating effect would still arise under this approach, where each learner would effectively have to learn two programs: one for calculating a bid value and one for calculating a real-valued action. These programs would need to execute simultaneously in the same register space without conflict: the instruction list would need to be capable of performing both functions, a substantial addition to the workload foisted on the evolutionary algorithm. The combined instructions mean that learners would need to evolve both capabilities simultaneously, since mutation could not discriminate between disrupting the development of bidding behaviour and the development of action calculation. The combined effect is an exponential increase in the difficulty of searching the space of solutions and is highly unlikely to produce successful agents.

A final approach that may seem apparent on identifying the scalar action as the limiting factor would be to expand valid scalar action values to include all real values within the range accepted by the environment. Unlike the approaches discussed previously, it is not obvious that this approach would fail to produce successful agents. However, this approach also sacrifices much of TPG's efficiency in searching the space of candidate solutions and, similarly to the previous approach, would likely dampen the evolutionary algorithm's selective pressure toward successful solutions. This negative effect comes from the loss of learner specialization that would accompany the modification of the scalar action values. In its current formulation, TPG is able to exploit learner specialization by dividing up the solution space into discrete subspaces in which solutions are simplified to binary decision-makers. As described in section 2.3.1 on Bid-GP, learner specialization has been shown to increase the performance of solutions, speed up time to working solution, and reduce the complexity of discovered solutions. Modifying the scalar action values such that they sample from a continuous range of values rather than assert one of a class of actions would result in the loss of these subspaces and likely in the loss of the benefits just described. Additionally, not only would the space of solutions lose the natural divisions described above, it would grow exponentially in size. The larger subspace combined with the decreased search efficiency are strong factors discouraging the use of this method for effective evolution of policies capable of producing useful real-valued actions..

3.2 Proposed Solutions

This work proposes that the solution, in general, is the combination of TPG and a decision-making network acting on state information contained entirely within TPG's graph structure. This decision-making network is chosen to be REINFORCE in this work, which is not only capable of producing one or more real-valued actions from state input, but is capable of optimizing its policy even after TPG has finished evolving. This work explores two variations of this approach with the final aim of producing one or more successful proof-of-concept algorithms that validate the approach in general.

3.2.1 Register Contents as State

The two variations differ primarily in their formulation of state. In the first variation, state is extracted from TPG's register values. Under TPG's original definition, where no additional memory vector is used, actions are based on the values held in a candidate's learners' registers, specifically on the values contained in each learner's register 0. Noting this, TPG can be reframed as a combination of both a feature- or state-engineering mechanism and a policy that operates on that state, where the state comprises the values held in the collection of R[0] values. Because a TPG solution capable of performing well in a given environment must necessarily hold actionable information in these registers, the bidding component of TPG execution is necessarily simultaneously a feature engineering algorithm. The first variation of TPG designed to produce real-valued actions through the addition of a decision-making network consists of attaching REINFORCE directly to an evolved TPG solution by feeding that solution's register 0 collection into REINFORCE as state at each timestep and allowing REINFORCE to determine the next action.

This variation has several requirements that preclude its use as a complete reinforcement learning algorithm and it is proposed exclusively to determine whether or not TPG is currently performing useful feature/state engineering. A simplification of the proposed algorithm applied to the CartPole environment is as follows:

1. Evolve a TPG solution capable of performing well in the discrete-action CartPole environment, where performing well may be defined as achieving a sufficiently high average score in post-evolution test runs (i.e. TPG policies with an average score of 100%). Note that the discrete-action CartPole environment is here necessitated by TPG's inherent limits as discrete-action algorithm.

2. Train REINFORCE in the continuous-action CartPole environment using the following action-taking mechanism:

   (a) At each timestep, feed TPG the environment's state (the state formulation in both the discrete and continuous CartPole environments is identical) and allow it to execute its bidding algorithm.

   (b) Feed the set of values found in TPG's collection of register 0 values after bidding to REINFORCE and allow REINFORCE to calculate an action.

(c) Feed REINFORCE's action back to the environment, i.e. take this action.

A successful policy generated through this approach would be evidence that:

- TPG is indeed engineering useful state representations internally, where 'useful' means that the state can be acted on by non-TPG policies to generate useful behaviour.
- The combination of TPG with a linear decision-making network is a viable method for producing real-valued actions.

Evidence of the first item in the preceding list would not only reveal more usefulness in current TPG solutions than was previously understood, but would also validate the re-framing of TPG as a state-engineering algorithm.

While this algorithm would be useful in validating this approach to real-valued action production, it suffers from two limitations that prevent its use as a complete reinforcement learning approach. First, it requires both a discrete and continuous version of the same environment. This is unlikely to be the case in practical applications in which the desired outcome is an agent capable of producing real-valued actions. In these cases the environment presumably requires real-valued actions. While a discrete version of the environment could potentially be simulated or contrived for the first phase of the algorithm in some cases, it is not necessarily true that this would be possible in all cases. Even when possible, its simulation may be non-trivial. An ideal algorithm would be useful in cases where only the continuous-action environment is available and the algorithm can still be used 'out-of-the-box'.

A second factor deterring this formulation is REINFORCE's requirement that the size of the state variable serving as its input must be known ahead of time and cannot be changed later. Because the total number of registers in a TPG solution can vary during its evolution, this means that the TPG solution serving as the feature engineer must be fully evolved in isolation and frozen prior to being fed to REINFORCE so that the number of registers will be fixed. This means that TPG must be frozen before being able to evaluate the usefulness of its state representation and cannot be updated if this representation is found to be inadequate. Additionally, because the TPG solution is evolved in isolation prior to the introduction of REINFORCE, TPG does

not 'know' during evolution how its state representation will ultimately be used: TPG will optimize the state representation in its register 0 collection to best serve its own policy, after which time REINFORCE will be tasked with optimizing its policy based on features that were engineered for another algorithm entirely. This disassociation of the two algorithms introduces a cap on the capabilities of REINFORCE in that TPG may have been more effective in its feature engineering had it received feedback during evolution informing it of the usefulness of its state as seen by REINFORCE. The following variation on the solution is introduced as a response to these limitations.

3.2.2 External State Representation

As noted, a successful result under the first variation of the TPG and decision-making network combination would provide evidence of TPG's capabilities as a feature engineer. The second variant described here is devised as an attempt to direct this capability deliberately and to have TPG create a better-behaved state solely for the purpose of feeding the decision-making network. In this case, 'better-behaved' means that the state has a pre-determined size and is intentionally leveraged as input to the decision-making network.

This variation builds on top of discoveries discussed earlier in section 2.4.4, in which TPG was found to be capable of using a memory vector to store register values and make use of them at a later point. When formulated as a memory, this vector allowed TPG to achieve a form of long-term planning. However, a wider applicability of the external vector is suggested by re-framing the vector not as simply a memory device but as a general-purpose state representation, albeit (or additionally) one capable of persisting state over multiple episodes. Just as the collection of values held in each learner's register 0 served as a state representation from which TPG derived bidding and action behaviour, the collection of values held in the external memory serves as a state representation from which TPG can build more complex bidding and action behaviour, including long-term planning. This ability to evolve more complex behaviour came, in this case, from the state representation's ability to persist state, an additional complexity not found in the original representation. However, beyond its ultimate use as a memory vector, the external vector serves as further evidence of TPG's ability to act as a feature/state engineer, and additionally

suggests that TPG can be made to generate this state in an arbitrary location as best serves the application.

The latter point suggests the use of an external state representation as input to the decision making network as the natural next step in this proof-of-concept. Here, the memory can be used nearly as described in section 2.4.4 and Smith's original paper [31]. This vector can be fed to the decision-making network at each timestep in order to select an action. This variation of the TPG and decision-making network combination overcomes several of the limitations of the previously-described variation. Because the external vector's size and existence can be known prior to evolution, it would no longer be necessary to generate a fully-evolved TPG solution before introducing the decision-making network. This also removes the necessity of the discrete-action version of the environment that is being solved. Rather, the combined TPG and decision-making network policy can be brought to bear on the continuous-action environment immediately, with TPG's own final action suggestions ignored in favour of the decision-making network's action suggestions. Additionally, TPG is fully 'aware' that its performance is based on the values it learns to write to this vector: the selective pressure is on the generation of an optimal state representation for the decision-making network from the outset. A final incidental benefit of this variation is that this algorithm also allows TPG to derive benefits from the external vector's facility as a memory device.

Though it is mitigated, the de-coupling of the two learning algorithms exists in this variation as well. Just as the de-coupling required the first variation to produce a fully-evolved TPG instance to bootstrap the final training algorithm, it now forces the second variation to choose one of the two components (either the TPG instance or the network weights) as fixed before it can train or evolve the other. This results in one of the two learning algorithms being entirely unintelligent for the first phase of training. However, because further evolution of the TPG instance has no effect on the shape of the external state vector, this variation allows for either algorithm to be re-trained or re-evolved additional times, allowing for multiple phases of alternating TPG evolution and network training. In this work, the environment was sufficiently simple that multiple phases were not required and in fact the implementation of the second variation did not require the decision-network to be trained at all.

Chapter 4

Experiments

Three groups of experiments were carried out in this work. One experiment, a precursor to the attempts to combine TPG with decision-making networks, was carried out to compare the effect of two different instruction sets on the frequency and range of values found in TPG registers. The remaining experiments focused on validating the algorithms under test and the viability of producing real-valued actions.

Efforts to combine TPG and REINFORCE were initially limited by what turned out to be the volatility of the register values found in trained TPG instances. Specifically, the original instruction set suggested by Banzhaf [7] and used by Kelly [17] in previous implementations of TPG, resulted in register values that span the entire range of valid 32-bit floating-point numbers.[1] The result was that multiplicative weighting calculations performed during REINFORCE training would often attempt to multiply extremely large (positively or negatively) floating-point values. Given that TPG was operating as a binary left/right controller, these values are never 'seen' from the application perspective. However, when attempting to apply gradient algorithms such as REINFORCE to state information from TPG, no effective scaling heuristic could be found. Ultimately, the research of this thesis resolved the issue by modifying the instruction set in an attempt to reduce the volatility of register values. The first experiment, comprising the first experiment group, compared the effect on register values of the instruction set used in canonical linear genetic programming with that of a modified set that was ultimately used for the remainder of this work.

After modifying the instruction set to accommodate the addition of REINFORCE, the remainder of the experiments were carried out. The two experiments comprising the second experiment group were designed to evaluate the viability of the combination of TPG and REINFORCE. These were intended to explore whether or not it is reasonable or even possible to generate a full reinforcement learning solution

[1] Register values in TPG as implemented in this work were 32-bit floating-point values, specifically the Numpy type 'float32' [28]

by feeding REINFORCE the contents of a TPG instance's register 0 collection as REINFORCE's state input. The underlying observation that guides this approach is that REINFORCE need not be any more complex than a linear model, but naturally returns real-valued actions. TPG in this case is pre-trained using a version of the task using discrete actions alone. REINFORCE then attempts to use the real-valued bid values from TPG to define state. Evaluation was performed using the well known CartPole task, a benchmark reinforcement learning task that can be defined equally well in terms of discrete and real-valued actions.

The remaining experiments, comprising the final set, were designed to evaluate the viability of directly evolving a TPG solution to produce a useful state representation for an external fixed-size vector of real-values via memory. In this context, the insight is to assume that the values TPG writes to memory can be directly transformed (using a static linear mapping) to a real-valued action. This was evaluated as a method for producing a solution to both the standard discrete-action CartPole environment as well as the modified continuous-action CartPole environment. Success in this configuration would demonstrate that TPG can produce real-valued actions without any additional learning algorithm.

4.1 Register Space Evaluation

In this experiment, two versions of TPG were compared, each of which adhered to the implementation described in section 2.4 on TPG in all aspects except the instruction set: one version implemented the instruction set described in section 2.4, while the other implemented the instruction described by Banzhaf [7] and used in previous implementations of TPG [17]. Both instruction sets are shown in the tables 4.1 and 2.1. In table 4.1, f_{max} and f_{min} refer to the maximum and minimum possible 32-bit floating-point values.

In both instruction sets, each assignment operation (i.e. the $\leftarrow$ operation) includes a check for NaN as well as a clamping operation to keep values between the minimum and maximum allowable values, which were the minimum and maximum 32-bit floating-point values. The latter check is necessitated by the Numpy type 'Inf' i.e. infinity.

Op-code	Operation
Addition	$R[i] \leftarrow R[i] + V[j]$
Subtraction	$R[i] \leftarrow R[i] - V[j]$
Multiplication	$R[i] \leftarrow R[i] * V[j]$
Division	$V[j] = 0.0\ ?\ R[i] \leftarrow f_{max} : R[i] \leftarrow R[i]/V[j]$
Cosine	$R[i] \leftarrow cos(V[j])$
Logarithmn	$V[j] < 0.0\ ?\ R[i] \leftarrow f_{min} : R[i] \leftarrow log(V[j])$
Exponential	$R[i] \leftarrow exp(V[j])$
Conditional	$R[i] < V[j]\ ?\ R[i] \leftarrow -R[i] : no - op$

Table 4.1: Instruction set used by canonical linear genetic programs

4.1.1 Instruction Set Differences

There are several notable differences in the instruction sets that can reasonably be expected to have a large impact on the final frequency and range of register values encountered during program execution.

The instruction set implemented in canonical linear GP and in prior implementations of TPG include both (natural) logarithm and exponential operations. Both of these instructions have behaviour likely to produce values at or near the range of valid 32-bit floating-point values: ln is asymptotic and approaches negative infinity as its input approaches 0.0, and exp easily produces large values from relatively low magnitude input. Hence, an input of 89.0 to the exponential operation will produce a value greater than the maximum positive 32-bit floating-point value. Additionally, the conditional check on the input to ln enables the production of the maximum negative 32-bit floating-point value if presented with any negative input. These operations alone significantly increase the likelihood that a linear genetic program instance's registers may contain values across the entire range of valid 32-bit floating-point values.

The instruction sets also differ in their implementations of the multiplication and division operations. The instruction set used in this work implements multiplication and division as doubling and halving operations respectively while the instruction set used in canonical linear GP allows the multiplication and division of any arbitrary register values. The former implementation has the effect of limiting the amount of change any register might see during a single multiplication or division instruction execution. For example a register holding a value of 10.0 cannot hold anything

larger than 20.0 or smaller than 5.0 after a single multiplication or division operation. This means several multiplications or divisions in a row (from the perspective of that register) are required to affect a large change to a register. In the canonical implementation, however, a single multiplication or division operation can result in a large change to a registers value. For example, if two registers hold 10.0, then a single multiplication instruction could increase one of them to 100.0 (and then 1000.0, then 10000.0 and so on). Correspondingly, the division operator is also potentially volatile: if a register holding a small value between 0 and 1 is used as the divisor in a division operation, a single division instruction may increase a register's value drastically. Both of these instructions can increase or decrease a register's value by orders of magnitude and so these operations also increase the liklihood that a linear genetic program's registers may contain values across the entire range of valid 32-bit floating-point values.

4.1.2 Test Procedure

To evaluate the effect of each instruction set on the range and frequency of values found in the registers of TPG instances, multiple TPG agents were evolved and tested using each instruction set. During testing, all register 0 values for the instance under test were recorded. The complete testing algorithm is as follows:

1. Perform the following n times:

    (a) Evolve a TPG agent in the discrete CartPole environment for $r_{generations}$ generations. Each agent was executed over $r_{episodes}$ episodes per generation, and surviving agents were allowed to skip re-evaluation for up to r_{skip} generations.

    (b) Test the highest-performing agent from evolution for $r_{test\ episodes}$ episodes. At each timestep during testing, collect and record the values stored in each learner's register 0.

This was repeated for both instruction sets. The values used are defined in table 4.2.

Parameter	Value
n	10
$r_{generations}$	20
$r_{episodes}$	3
r_{skip}	2
$r_{test\ episodes}$	10

Table 4.2: Parameters used in instruction set comparison experiment

4.2 TPG and REINFORCE

Two experiments were performed to evaluate the viability of the combination of TPG and REINFORCE as an approach to producing complete reinforcement learning solutions in general, and as an approach to producing reinforcement learning solutions capable of performing real-valued actions in particular. In general, this set of experiments set out to evaluate the approach described in section 3.2.1. The first experiment used only the discrete-action CartPole environment and was intended to evaluate whether the pairing of TPG and REINFORCE could produce functioning agents. The second experiment allowed TPG to evolve in the discrete-action CartPole environment, while REINFORCE was trained in the continuous-action CartPole environment to determine whether useful real-valued actions could be produced.

4.2.1 Discrete-Action Solution

In the first experiment in this set, TPG with no additional memory vector was allowed to evolve in the discrete-action CartPole environment. A TPG agent evolved in this step was intended to be used as a feature engineer for REINFORCE. After a TPG agent was evolved, a REINFORCE agent was trained which used that TPG agent's register 0 collection as its state input. This second step was performed multiple times, i.e. multiple REINFORCE agents were trained on the same TPG agent. This was done to account for the possibility that the gradient-based REINFORCE may become stuck in a local maximum due to unfortunate random initialization of its weights. A failure of REINFORCE to produce a working agent after only a single training attempt should not be considered proof that the approach is not viable.

In this and all work using REINFORCE, the policy network was a single linear

network with one output unit fed into one of two non-linearities. This architecture is shown in figure 4.1. In this figure, the linear network is shown between the Memory Matrix and Non-linearity components. Note that these units receive the register 0 values unaltered before applying the network weights, i.e. the first set of arrows represent direct transfer of values whereas the second set represent multiplicative weighting.

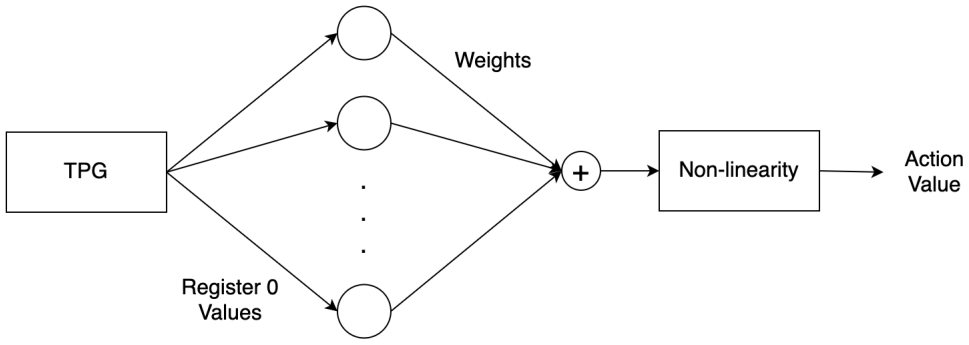

Figure 4.1: Architecture of the hybrid algorithm combining TPG, via its register 0 values, and REINFORCE, used to perform gradient ascent to update the weight vector.

Which non-linearity was used depends on whether REINFORCE was being used to produce discrete and real-valued actions: for discrete actions, the output was passed through a sigmoid function[2] and rounded to 0.0 or 1.0, the two actions accepted by the discrete-action CartPole environment; for continuous actions, the results were passed through a tanh function to produce a real value between -1.0 and +1.0, and an action was sampled from a normal distribution using this value as the mean. In short, the 'non-linear' element provides an asymptotic mapping from the TPG register values (which might still span a large dynamic range) to either the [0.0, ..., 1.0] interval or the [1.0, ... -1.0] interval. The non-linear function is essentially linear ($y = x$) around the origin. However, as the magnitude of the dependent variable (x) increases the corresponding limits are asymptotically approached. A high-level diagram of this mechanism is shown in figure 4.2. In this diagram, S_t represents the environment state and S'_t represents TPG's internal state representation, contained in its register 0 values. The full algorithm used in this experiment is as follows:

[2]Sigmoid function: $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

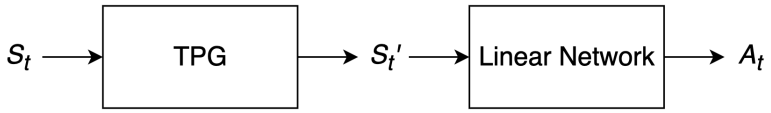

Figure 4.2: Action selection diagram of the hybrid algorithm combining TPG, via its register 0 values, and REINFORCE.

1. Repeat the following n_{runs} times:

   (a) Evolve a TPG agent in the discrete-action CartPole environment for $r_{generations,TPG}$ generations. Each agent was executed over $r_{train\ episodes,TPG}$ episodes per generation, and surviving agents were allowed to skip re-evaluation for up to $r_{skip,TPG}$ generations. At each generation the top agent score was recorded.

   (b) Test the highest-performing agent from evolution for $r_{test\ episodes,TPG}$ episodes. After each episode, the agent's final score was recorded.

   (c) Repeat the following $n_{training\ sessions,REINFORCE}$ times:

      i. Train REINFORCE in the discrete-action CartPole environment for $r_{train\ episodes,REINFORCE}$ where at each timestep:

         - TPG receives the state of the CartPole environment.
         - TPG executes its bidding/action mechanism.
         - The values in each learner's register 0 are collected into a single vector (the order in which learners' registers are checked is always the same).
         - REINFORCE receives this vector at its state input and selects an action. When the entire episode is complete, REINFORCE's score is recorded.

      ii. Test the resulting REINFORCE agent for $r_{test\ episodes,REINFORCE}$ episodes. These episodes' initial conditions are identical to the initial conditions seen by TGP agents during testing. At each episode, the agent's score is recorded.

The parameters described above are defined in table 4.3.

Parameter	Value
n_{runs}	20
$r_{generations,TPG}$	20
$r_{training\ episodes,TPG}$	25
$r_{test\ episodes,TPG}$	1000
r_{skip}	2
$r_{training\ sessions,REINFORCE}$	10
$r_{training\ episodes,REINFORCE}$	5000
$r_{test\ episodes,REINFORCE}$	1000

Table 4.3: Parameters used in discrete-action TPG + REINFORCE experiment

4.2.2 Continuous-Action Solution

The second experiment in this set was designed to determine if REINFORCE was capable of producing real-valued actions using the state engineered by TPG. This experiment was identical to the one described previously, with the modification that REINFORCE was trained and tested in the continuous-action CartPole environment. The architecture and action-selection mechanisms are as described in the previous section.

4.3 TPG and External State Vector

The final set of experiments were performed to determine whether a decision-making network could produce useful solutions to the discrete-action and continuous-action CartPole environments using an external memory vector component of TPG as state input, as described in section 3.2.2. The first experiment used only the discrete-action CartPole environment, while the second used only the continuous-action CartPole environment.

For these experiments, TPG was modified to re-introduce an external memory matrix, as used by Smith and Heywood and described in section 2.4.4. The trainable REINFORCE was dropped in favour of a simple linear decision network with fixed weights. This network was equivalent to the network used in the previous experiments with REINFORCE with all weights initialized to 0.1 and never updated. This leaves open the possibility that the network could be trained in future work. Decisions were made by feeding the external memory vector values into this network. This

architecture is shown in figure 4.3. In this figure, the linear network is shown between the Memory Matrix and Non-linearity components. Note that these units receive the register 0 values unaltered before applying the network weights, i.e. the first set of arrows represent direct transfer of values whereas the second set represent multiplicative weighting.

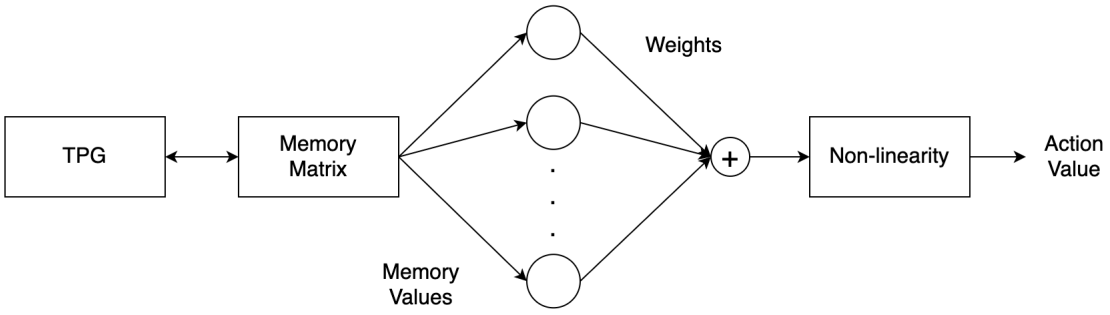

Figure 4.3: Architecture of the hybrid algorithm combining TPG, its external memory matrix, and a linear decision network.

Actions were selected identically to the methods used by REINFORCE: for discrete actions, the results was passed through a sigmoid function and rounded to 0.0 or 1.0, the two actions accepted by the discrete-action CartPole environment; for continuous actions, the results were passed through a tanh function to produce a real value between -1.0 and +1.0, and an action was sampled from a normal distribution using this value as the mean. A high-level diagram of this mechanism is shown in figure 4.4. In this diagram, S_t represents the environment state and S'_t represents TPG's internal state representation, contained in its memory matrix.

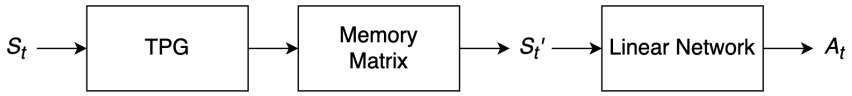

Figure 4.4: Action selection diagram of the hybrid algorithm combining TPG, its external memory matrix, and a linear decision network.

4.3.1 Memory Matrix Modifications

The memory matrix used in this work is modified from the version described in section 2.4.4. Specifically, the matrix is reduced from a size of 100×8 to 5×8. While

the larger, 800-entry matrix may encourage behaviour such as short-term and long-term memory, it generally produces sparse matrices. Moreover, the state space and duration of the reinforcement learning tasks used to demonstrate the original work involved $\approx 76,000$ attributes, programs consisting of 1000 instructions and thousands of temporal steps. Conversely, the state space of CartPole is only 4 and programs in this work are allowed a maximum of 128 instructions. With this in mind, a much smaller memory parameterization is adopted, as follows:

$$p = 0.7 - (0.2 * row)^2$$

over the 5 rather than 100 rows. This increases the maximum probability of any given write to 70% over the 25% used in the original formulation, which in turn reduces the likelihood of spare memory content. Sparsity is deemed potentially undesirable because it implies that any mapping to a real-valued action will be that much more sensitive to the few values written to memory.

4.3.2 Experiment

This experiment was performed two times: once in which the discrete-action CartPole environment was used, and a second time in which the continuous-action CartPole environment was used. The full algorithm used in this experiment was as follows:

1. Repeat the following n_{runs} times:
   (a) Evolve a TPG agent in the CartPole environment for $r_{generations}$ generations. Execute each agent over $r_{train\ episodes}$ episodes per generation. Surviving agents may skip re-evaluation for up to r_{skip} subsequent generations. At each generation, record the top agent score.
   (b) Test the highest-performing agent from evolution for $r_{test\ episodes}$ episodes. At each episode, record the agent's final score.

At each timestep, action selection was performed as follows:

1. TPG receives the state of the CartPole environment.
2. TPG executes its bidding/action mechanism, which may write values to the external memory vector.

3. The memory vector is flattened and fed to the decision-making network.

4. The decision making network produces an action suggestion as described above. This action is fed back into the environment.

The parameters described above are defined in table 4.4.

Parameter	Value
n_{runs}	20
$r_{generations}$	100 in discrete-action CartPole, 50 in continuous-action CartPole
$r_{episodes}$	5
r_{test}	1000
r_{skip}	2

Table 4.4: Parameters used in discrete-action TPG + REINFORCE experiment

Chapter 5

Results

5.1 Comparison of Instruction Sets

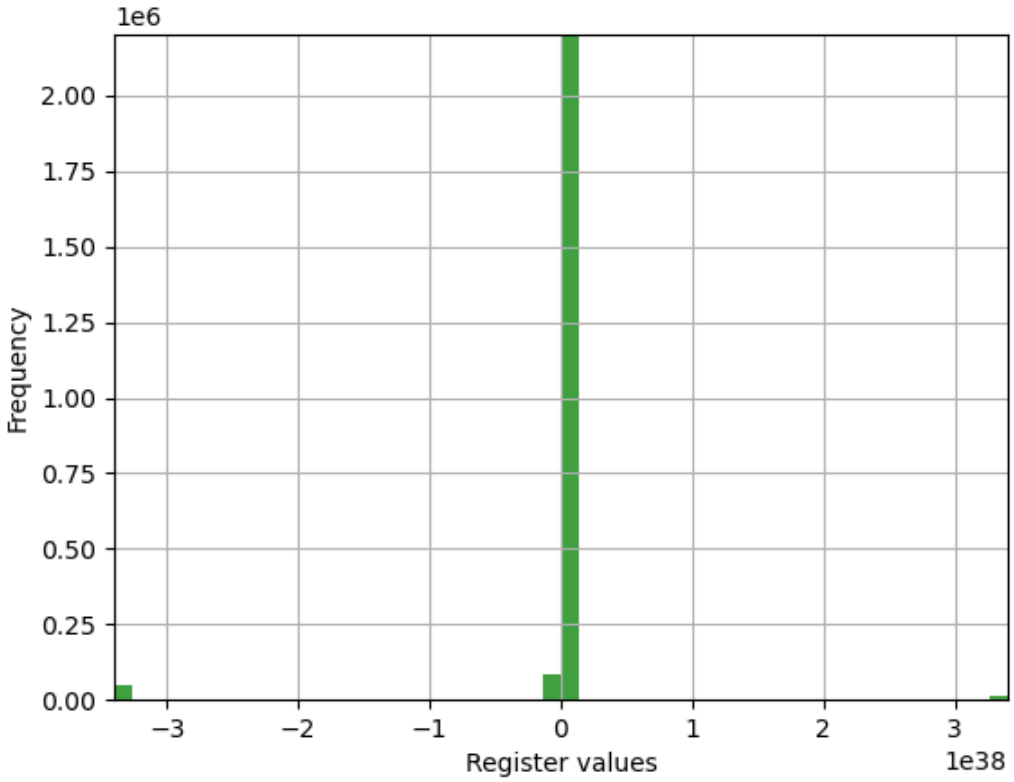

Figure 5.1: Histogram showing the distribution of register values, sampled from TPG agents using the canonical instruction set

All agents used in the comparison of the effects of the instruction set on register values were evolved to be highly performant: the average score of both sets of agents (those using the canonical instruction set and those using the modified instruction set) over 10 test episodes in the discrete CartPole environment was above 480.0 (out of a maximum of 500.0). These agents are therefore expected to be representative of

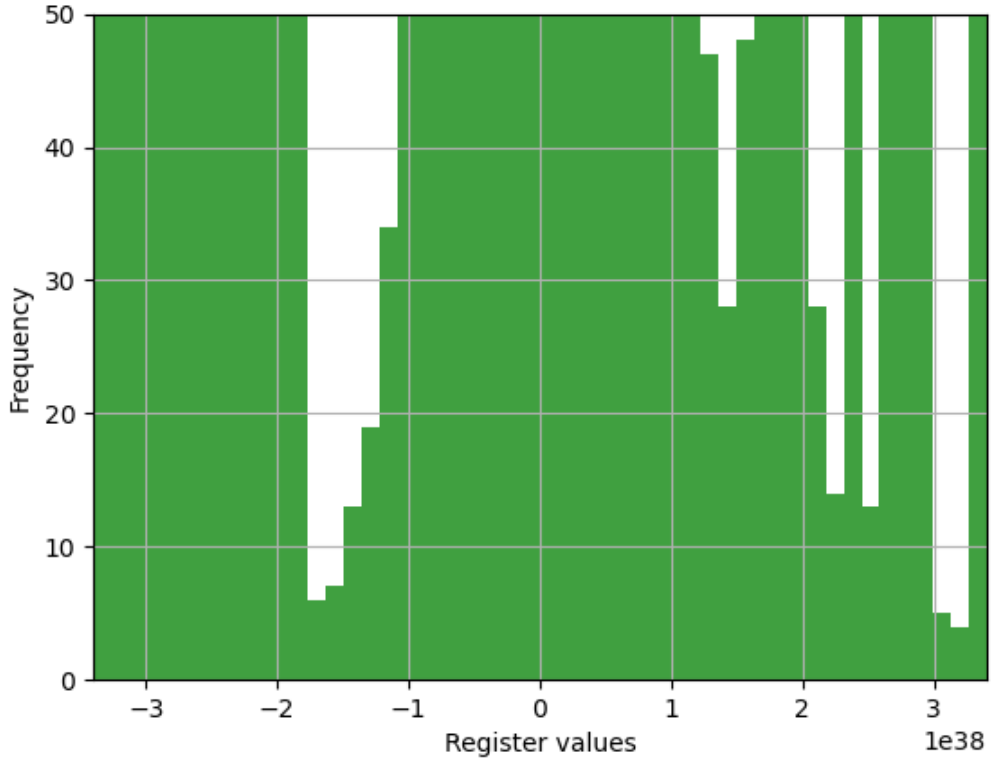

Figure 5.2: Histogram showing the distribution of register values, sampled from TPG agents using the canonical instruction set, cut off to better display the spread of values.

typical fully-evolved CartPole agents.

Figures 5.1 and 5.2 show the results of sampling register 0 values from the TPG agents that used the canonical instruction set over 10 test episodes each. Register values are put into one of 50 bins. Figure 5.1 displays the histogram with its bounds set based on the minimum and maximum bin values as well as bin height, while figure 5.2 displays the same results with the bin heights artificially set to 50 to better display the results in those bins with relatively few entries. As shown, the register values do indeed span the range of 32-bit floating-point values, where the maximum magnitude of a floating-point value is $\approx 3.4 \times 10^{38}$. In fact, the groupings shown in figure 5.1 suggest that TPG makes significant use of both the positive and negative maximum value; though most values are within the two bins adjacent to and including 0, the relative height of the bins including the positive and negative maximum value indicate

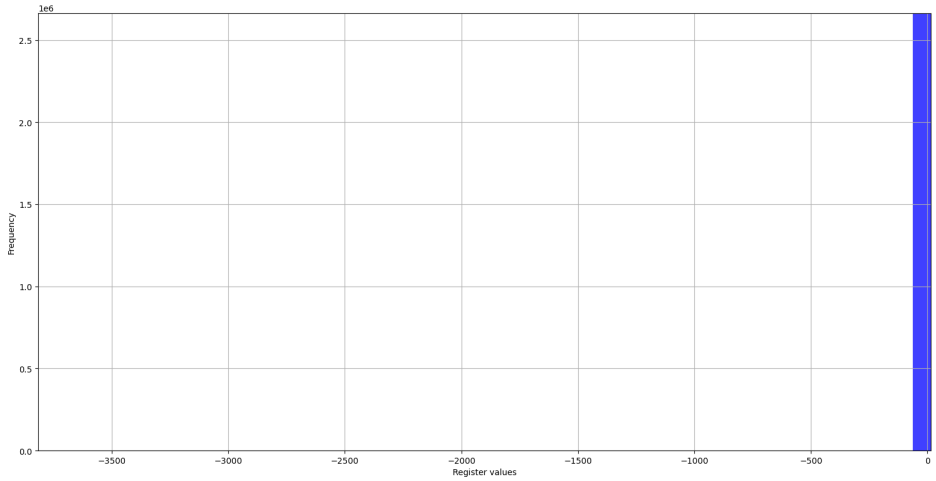

Figure 5.3: Histogram showing the distribution of register values, sampled from TPG agents using the modifed instruction set.

that most registers containing a value outside the two 0-adjacent bins contain either the positive or negative maximum floating-point value.

Figure 5.2 indicates that indeed all bins contain values, though some contain a relatively small number. Therefore, for at least those agents sampled, the entire range of 32-bit floating-point values is used by TPG agents.

Figures 5.3 and 5.4 show the same results of sampling register 0 values over 10 test episodes, this time from the TPG agents that used the modified instruction set. Register values were again put into one of 50 bins. As shown in figure 5.3, most register 0 values fall within a relatively small range around 0 (relative when compared to the same data taken from the canonical instruction set agents). After artificially cutting the bin height at 50 again, figure 5.4 shows a similar grouping effect to that seen in the canonical instruction set agents' register 0 values. These groupings may suggest that TPG makes use of these groups to represent coarse information. However, in agents using the modified instruction set, these groupings span only a fraction of the available 32-bit floating-point values.

These results suggest that the modified instruction set encourages TPG to store the same state information as agents using the canonical set in a smaller value space. This instruction set was used in all the following experiments in this work on the

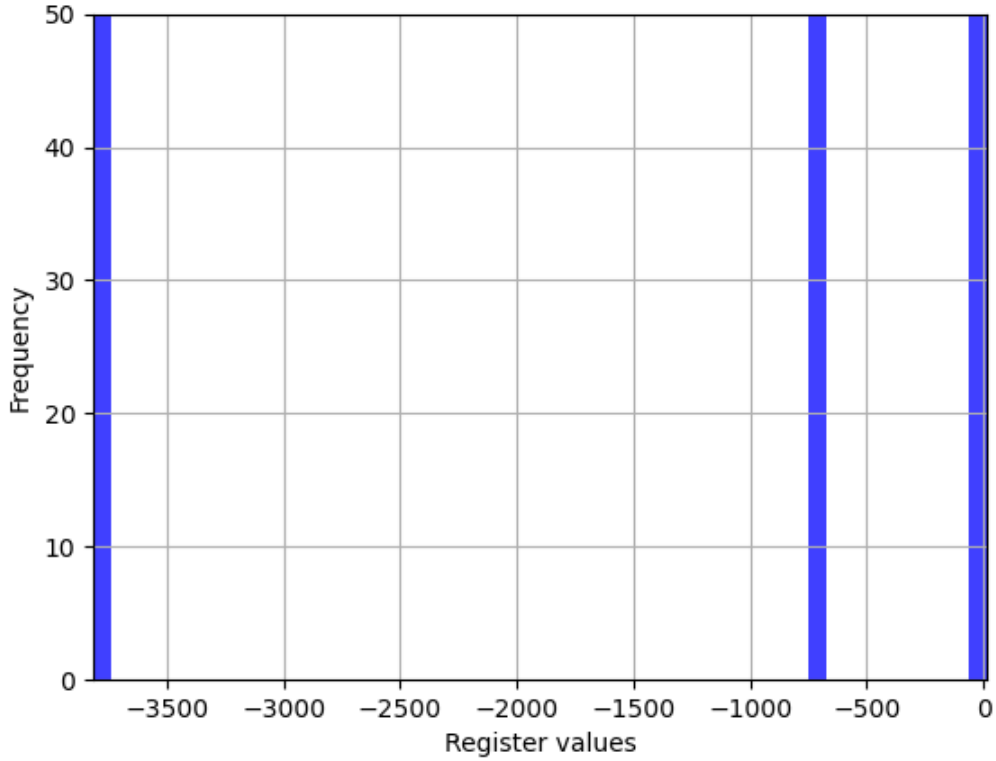

Figure 5.4: Histogram showing the distribution of register values, sampled from TPG agents using the modifed instruction set, cut off to better display the spread of values.

basis that the smaller range of values would be more useful to decision networks that typically benefit from input normalization. Input normalization to neural networks typically leads to faster training with higher learning rates and less sensitivity to weight initialization [13]; therefore this modified instruction set will likely aid REINFORCE and any other gradient methods used in the future. Additionally, the smaller range of register values seen should minimize the loss of information when register values were clipped to between -20.0 and +20.0 before being fed to the decision networks, which was performed in all experiments in this work. Normalization/standardization is typically applied to neural network attributes, but is only effective when the numerical values do not include 'special values' (e.g. under/overflow limits) since these preclude the use any normalizing operation.

5.2 TPG and REINFORCE

5.2.1 Discrete Action REINFORCE and CartPole

For this experiment, 20 TPG agents were evolved over 20 generations. Figure 5.5, 5.6 and 5.7 provide an overview of the relative complexity of the TPG agents produced during evolution. It is apparent that for the most part agents consisted of less than 10 teams and less than 40 programs. No preference for a particular instruction type was observed.

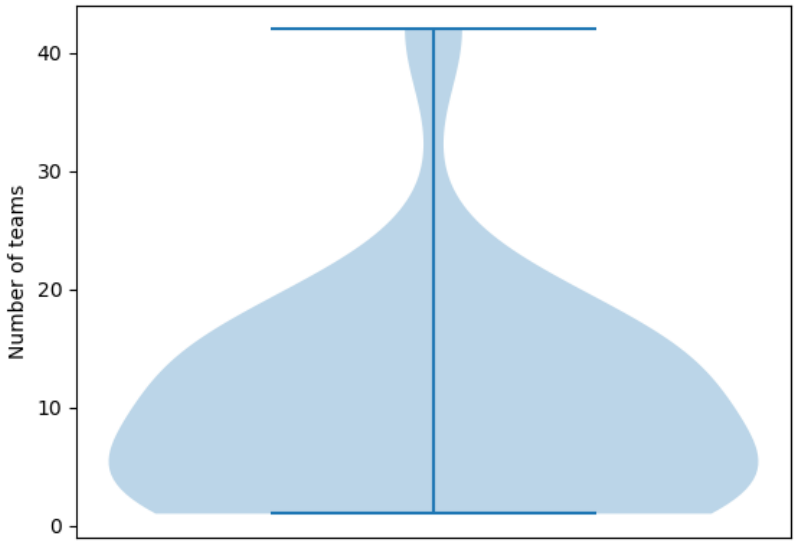

Figure 5.5: Distribution of total Team counts across TPG agents used in the discrete-action REINFORCE experiment.

As each agent was evolved, the top-scoring agent's score at each generation was recorded. After evolution, a 20×20 matrix (20 runs of evolution, each including 20 generations) of these recorded training scores was collected. To review the performance of these agents during evolution, the maximum, median, and minimum top score at each generation was extracted. Figure 5.8 shows these results. It is worth noting that the procedure just described for extracting scores does not extract scores of individual agents, but rather produces a representative curve of what the best, worst and 'average' agents' performance was during evolution. As shown in

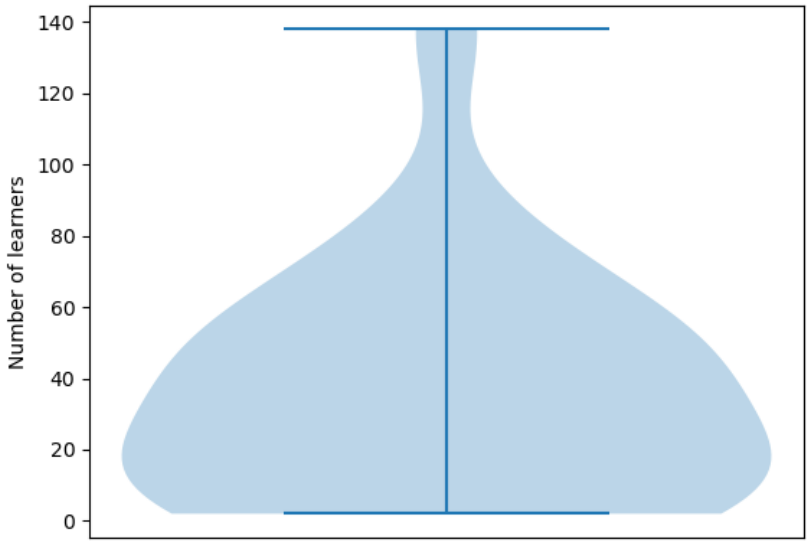

Figure 5.6: Distribution of total Learner counts across TPG agents used in the discrete-action REINFORCE experiment.

figure 5.8, even the representative median agent achieves high performance after 20 generations, scoring 100% for the last several generations of evolution.

Each of these agents was tested over 1000 episodes, each with different starting configurations (though each agent saw the same sequence of 1000 different starting configurations). Figure 5.9 shows the distribution of TPG agents' test scores across the 1000 test episodes. The mean agent performance was 486.46 out of 500.0 with a standard deviation of 41.22. The top-scoring agent achieved an average score of 500 over all 1000 episodes.

The training curves and test results suggest that TPG is capable of producing high-performing agents in the discrete CartPole environment. As shown in figure 5.9, 8 out of 20 agents scored 500.0/500.0 on all 1000 test episodes, while most others show a tight cluster of high scores near the 500.0 mark. Based on this, it can be assumed that REINFORCE is operating with high-quality TPG agents as its input in most cases.

For each TPG agent produced, a discrete-action REINFORCE agent[1] was trained

[1]The architecture for a REINFORCE agent is always a linear perceptron. Inputs are the R[0]

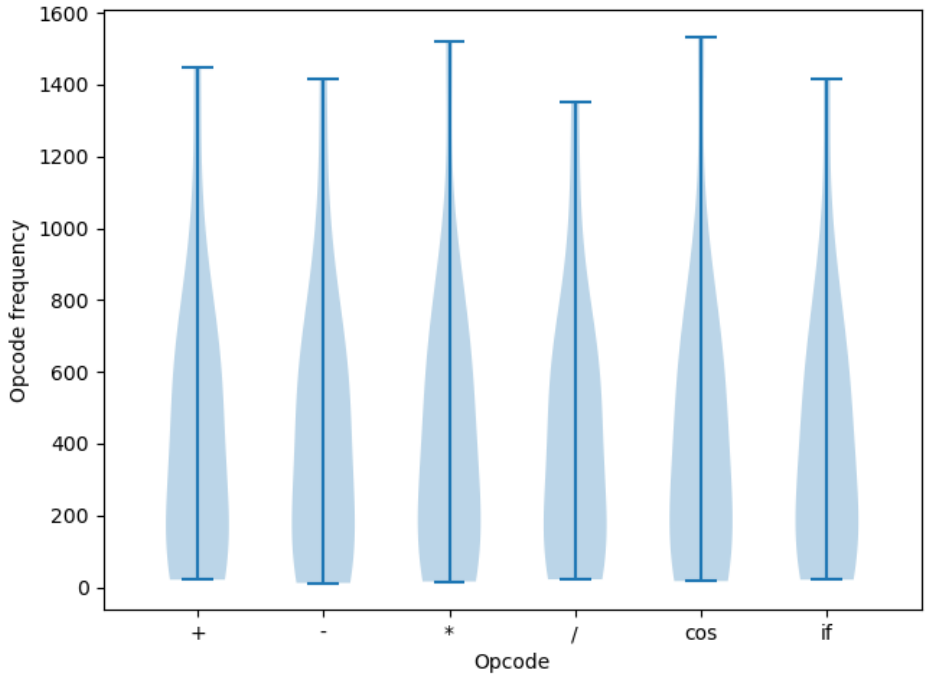

Figure 5.7: Distribution of op-code counts across TPG agents used in the discrete-action REINFORCE experiment.

over 5000 episodes using that agent as its input. Each REINFORCE agent was given 10 opportunities to re-train from a random initialization so that 'unlucky' initialization could be ruled out as a likely cause for any poor performance seen by REINFORCE. Each re-training from the same TPG agent is referred to below as a REINFORCE *session*. The task of settling on a final REINFORCE agent out of all 10 session candidates per TPG agent was done by calculating the mean test score for each session candidate and selecting the candidate with the highest mean test score. The training scores for all 20 of these top-performing candidates were collected into a 20×5000 matrix. Similarly to TPG, the maximum, median, and minimum score at each episode was extracted. Figure 5.10 shows these results, with each curve smoothed using a 50-episode rolling average. As shown in figure 5.10, even the representative median REINFORCE agent achieves a high score after 5000 episodes. However, unlike TPG, the representative worst-performing agent effectively

registers from each program comprising a TPG solution. Output is the force applied to the cart, as described in section 2.6.

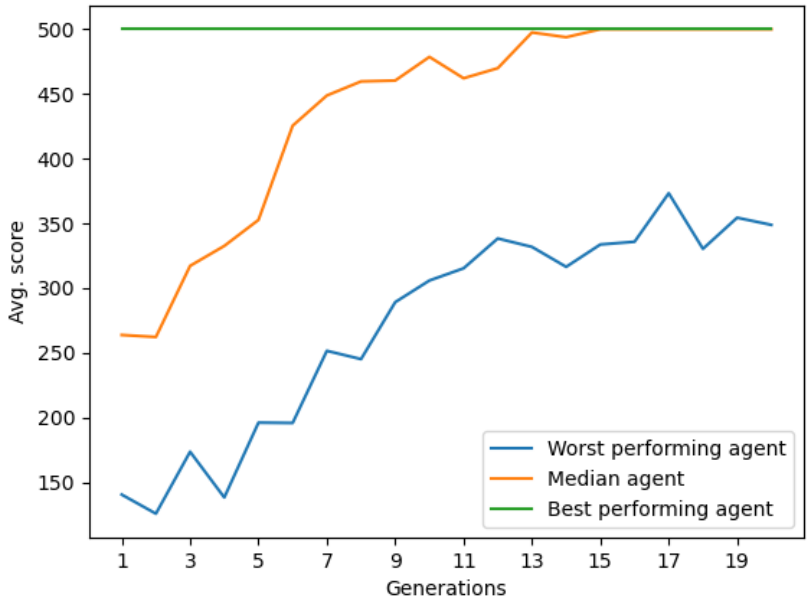

Figure 5.8: TPG training curve showing the top agent scores per generation for TPG agents used in the discrete-action REINFORCE experiment.

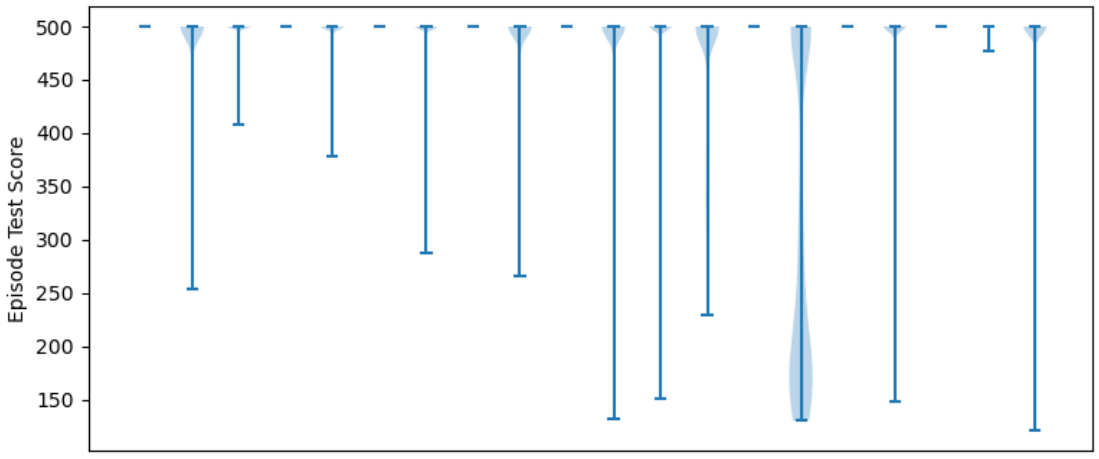

Figure 5.9: Distribution of TPG agents' test scores across 1000 test episodes for TPG agents used in the discrete-action REINFORCE experiment. Each violin represents the distribution of a single TPG agent's 1000 test scores.

achieves a performance of nearly 0 on average. This indicates that REINFORCE is not guaranteed to converge on useful behaviour.

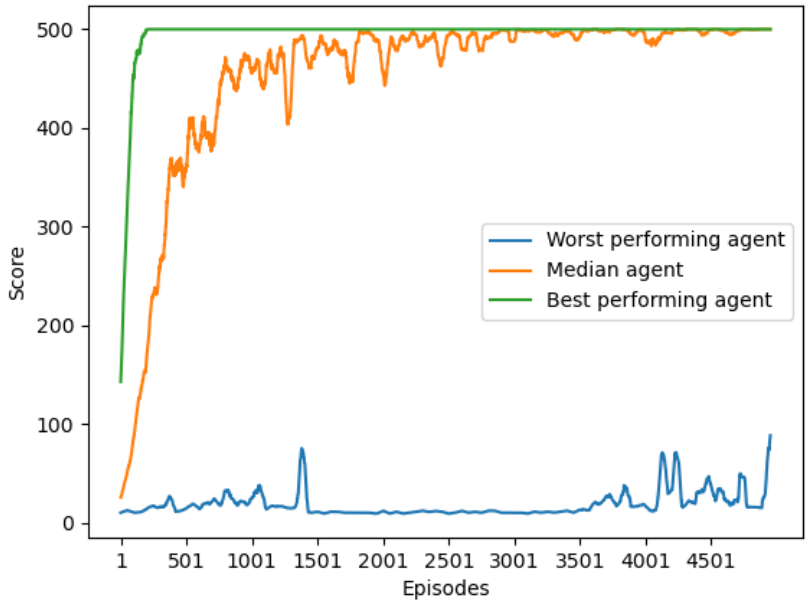

Figure 5.10: Discrete-action REINFORCE training curve showing the top agent scores per episode.

After each session, REINFORCE was presented with the same 1000 test episode configurations presented to TPG. REINFORCE's score was recorded after each test episode. Figure 5.11 show the distribution of test scores over all 20 finalized REINFORCE agents. The mean agent performance was 473.90 out of 500.0 with a standard deviation of 83.44. The top-scoring agent also achieved an average score of 500.0/500.0 over all 1000 episodes. The distribution shows strong clustering of results around 100% for all but one of the finalized REINFORCE agents, though the wide distributions show that even a highly capable agent, i.e. one with a high average test score, occasionally scores very poorly, consistent with TPG.

These results are strong evidence for the validity of this method of producing a reinforcement learning agent by feeding TPG's register 0 collection into REINFORCE as a state vector. The consistently-rising median and top training curves and high test performance are strong indications that REINFORCE is capable of making use of TPG's register 0 collection and therefore that TPG is producing a useful state representation in its register banks that can be interpreted by other algorithms.

Despite the strong indication that this is a valid approach, the higher standard

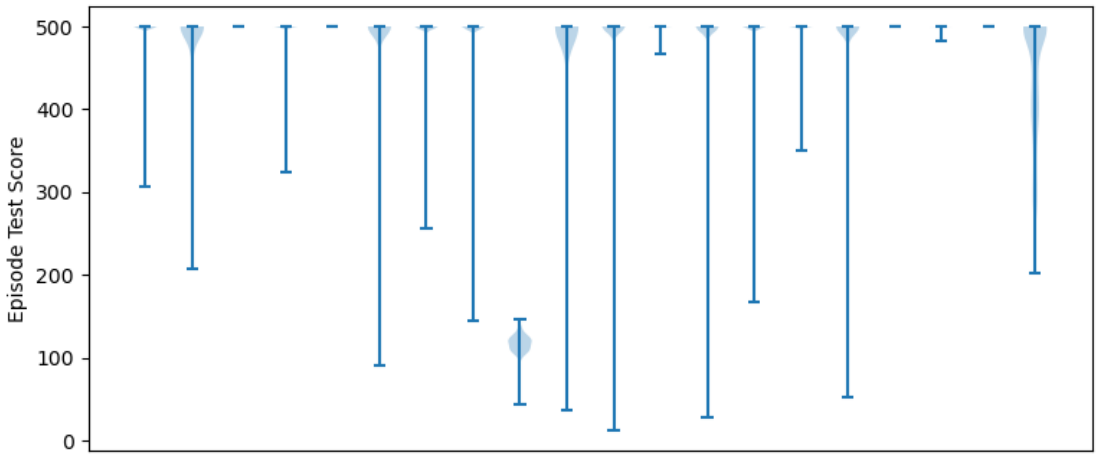

Figure 5.11: Distribution of discrete-action REINFORCE agents' test scores across 1000 test episodes. Each violin represents the distribution of a single REINFORCE agent's 1000 test scores.

deviation of the combination of REINFORCE and TPG compared to TPG alone as well as the extremely poor performance of the representative worst-performing agent suggest that this method may not be robust: it is evidently possible to run the algorithm as described without necessarily producing a useful agent. As noted, REINFORCE is given multiple opportunities (sessions) to re-train with a new initialization of its network parameters. If the poor performance is still due to bad luck in its network initialization then the only solution would be to increase the number of allowed sessions. This quickly gets expensive in terms of both time and computation, since REINFORCE needs to train from scratch on each new initialization attempt.

5.2.2 Continuous REINFORCE and CartPole

The same results were collected for the case of continuous-action REINFORCE agents trained in the continuous-action CartPole environment, based on TPG agents trained in the discrete-action CartPole environment. Figure 5.12 shows the training curve for the 20 TPG agents used in this experiment. Figures 5.13, 5.14 and 5.15 show the relative complexity of the TPG agents produced. These agents demonstrate similar complexity and training performance to those produced for the previous experiment.

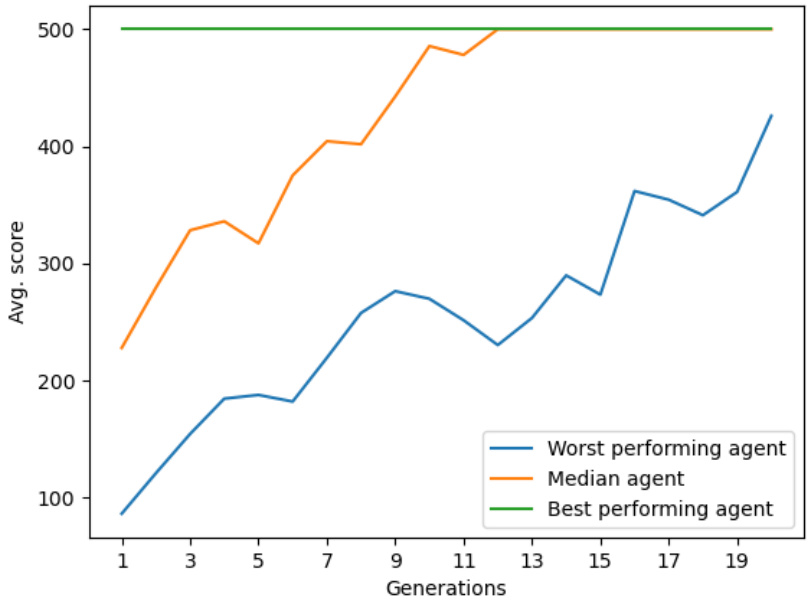

Figure 5.12: TPG training curve showing the top agent scores per generation for TPG agents used in the continuous-action REINFORCE experiment.

This is as expected since both sets of TPG agents were produced in the same discrete-action CartPole environment.

Each TPG agent was tested under the same conditions as those used in the discrete-action REINFORCE experiment, described in section 5.2.1. Figure 5.16 shows the distribution of TPG test scores. These agents achieved an average score of 489.11 out of 500.0 with a standard deviation of 17.56. Again, 8 out of 20 agents achieved a mean score of 100%. Note that in this experiment TPG was still evolved in the discrete-action CartPole environment. These results indicate that these agents performed similarly to those used in the previous experiment.

In this experiment, REINFORCE was trained over 25000 generations rather than 5000. Otherwise, the procedure was identical to that used for discrete-action REINFORCE in section 5.2.1. Figure 5.17 shows the results of training REINFORCE in this environment. As shown, the representative median-performing agent only achieves high-performance at the end of the 25000 available training episodes, already 5 times the number of episodes required by the median-performing agent in the discrete-action CartPole environment. This indicates that the continuous-action

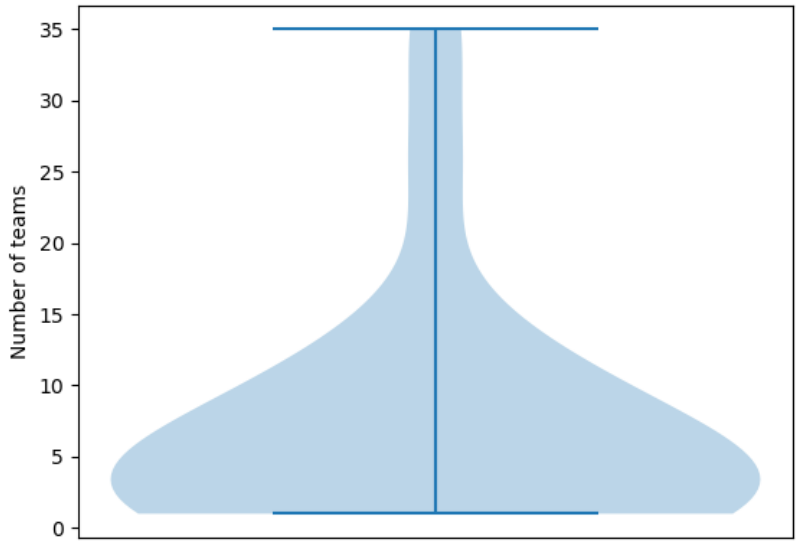

Figure 5.13: Distribution of total Team counts across TPG agents used in the continuous-action REINFORCE experiment.

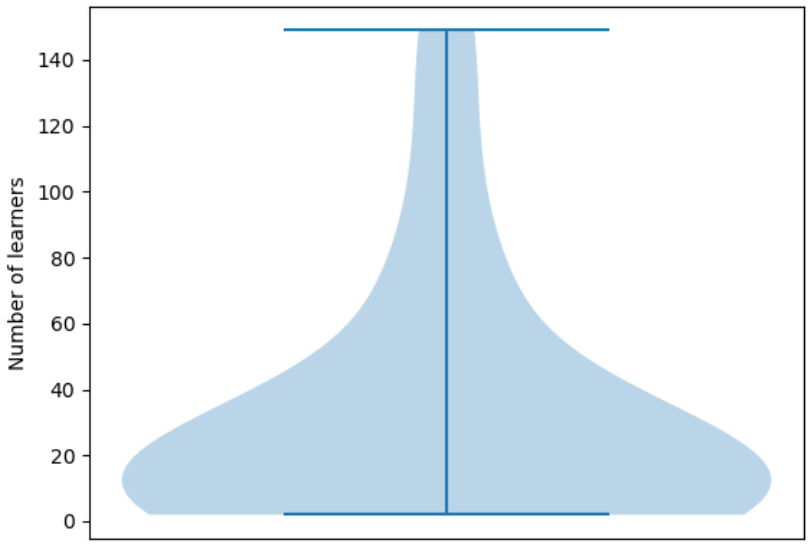

Figure 5.14: Distribution of total Learner counts across TPG agents used in the continuous-action REINFORCE experiment.

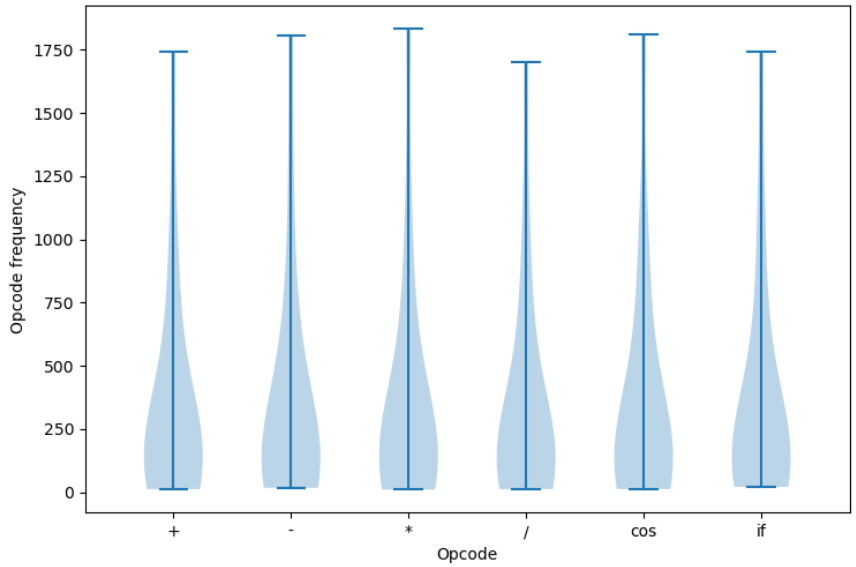

Figure 5.15: Distribution of op-code counts across TPG agents used in the continuous-action REINFORCE experiment.

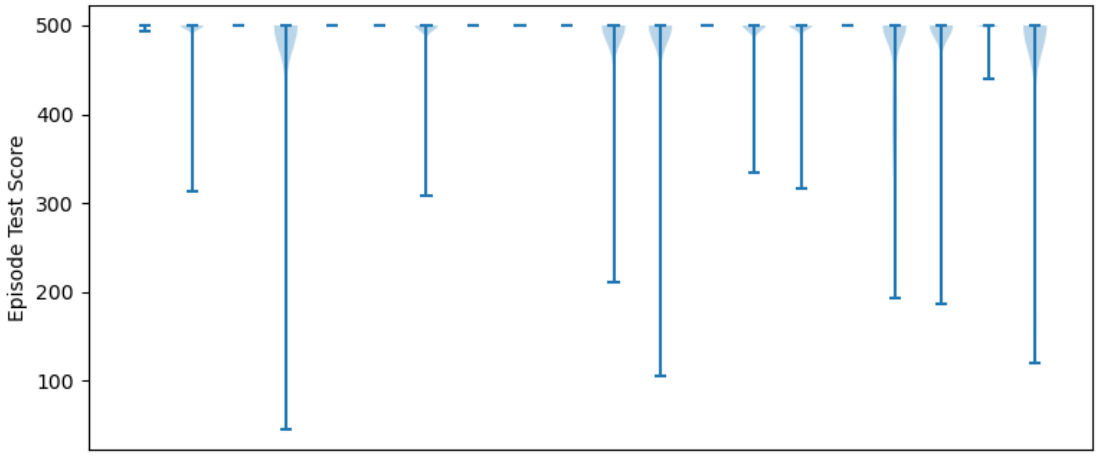

Figure 5.16: Distribution of TPG agents' test scores across 1000 test episodes for TPG agents used in the continuous-action REINFORCE experiment. Each violin represents the distribution of a single TPG agent's 1000 test scores.

environment is significantly harder to navigate.

Tests were also performed just as described in section 5.2.1. Figure 5.18 shows

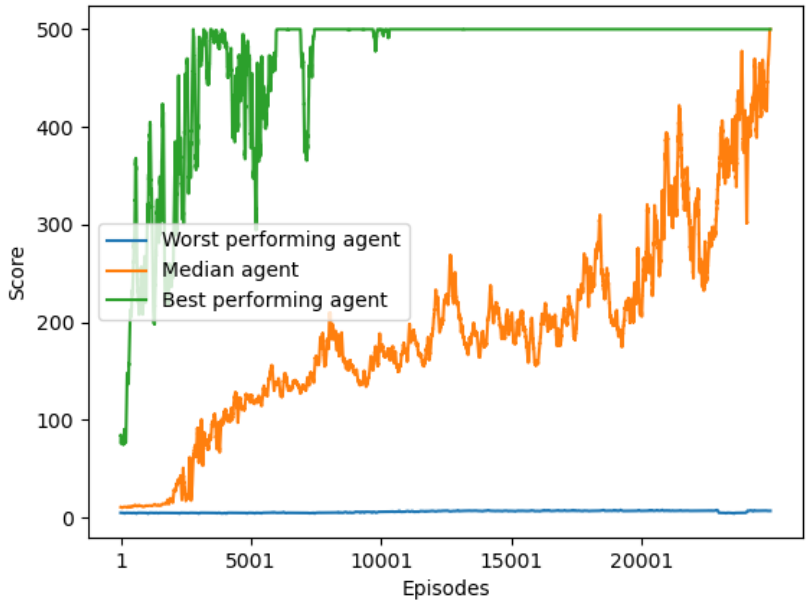

Figure 5.17: Continuous REINFORCE training curve showing the top agent scores per episode.

the distribution of test scores. The majority of agents still show a tight distribution around 100%, though a higher number of agents show a maximum score well below 100% when compared to those produced in the discrete-action environment. The mean agent performance was 401.68 out of 500.0 with a standard deviation of 151.03. This represents a drop in performance, evident in the lower mean score, as well as an indication of increased volatility, evident in the increased standard deviation. The higher volatility and increased number of sub-optimal agents (those whose maximum score is less than 100%) shows that this algorithm is not guaranteed to produce a useful agent. However, high-performing agents (those whose test scores show a tight cluster around 100%) are still the majority. The top-scoring agents achieved an average score of 500.0 over all 1000 episodes.

To further characterize REINFORCE's behaviour, an additional REINFORCE agent was trained and tested as described above. This agent achieved a mean score of 480.78 over 100 test episodes and so can be roughly considered to be a well-behaved agent. This agent's actions were recorded over a single test episode in which the agent scored 500.0. The recorded action values are shown in figure 5.19. Figure 5.20 shows

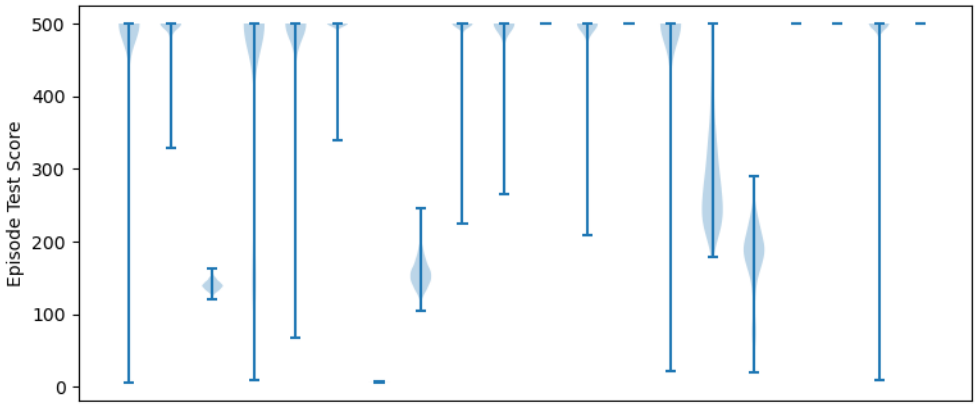

Figure 5.18: Distribution of continuous-action REINFORCE agents' test scores across 1000 test episodes. Each violin represents the distribution of a single REINFORCE agent's 1000 test scores.

a restricted portion of the same plot for better visibility into the oscillations of the actions. These plots suggest that REINFORCE is not simply re-learning something equivalent to discrete-action behaviour, e.g. learning to produce either of two large and opposing values at each timestep. Rather, REINFORCE is learning a policy in which action values are selected deliberately from within the allowed range of values.

Despite the drop in performance when compared to the case of REINFORCE trained in the discrete-action CartPole environment, the results of this experiment are encouraging. These results are strong evidence for the validity of this method of producing a TPG-based reinforcement learning agent capable of operating in a continuous-action environment. The training curves show that REINFORCE is still typically able to make use of TPG's register 0 collection to derive actions. Even more encouraging, REINFORCE is capable of using TPG's ad-hoc state vector (its register 0 collection) to produce continuous actions, despite TPG having only learned to navigate the discrete-action environment. This suggests that TPG is forming a relatively robust state representation in its register values. It should also be noted that REINFORCE was halted artificially after 25000 episodes based on time constraints during experiments and it is possible that a higher performance may have been observed had REINFORCE been afforded more episodes during training. This is suggested by the median training curve (figure 5.17), which indicates that the median agent learned to

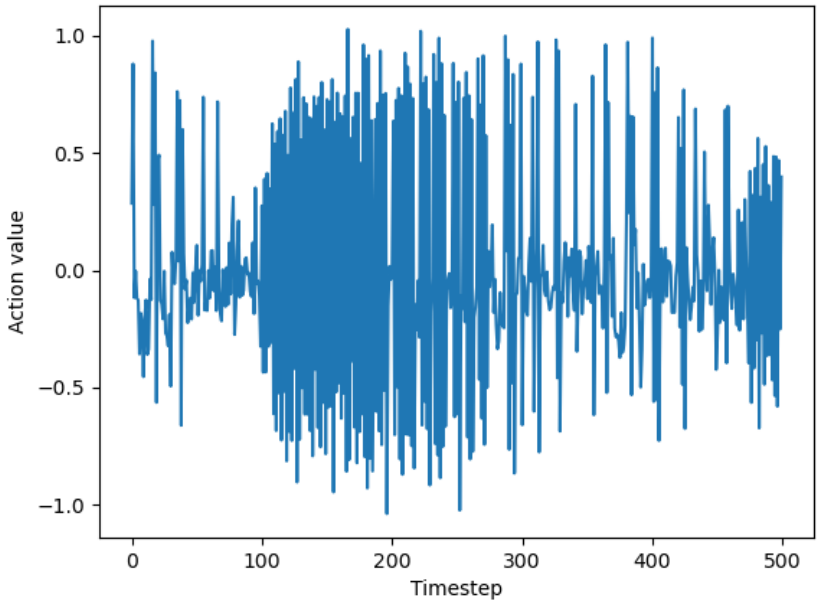

Figure 5.19: Recorded actions of a continuous-action REINFORCE agent.

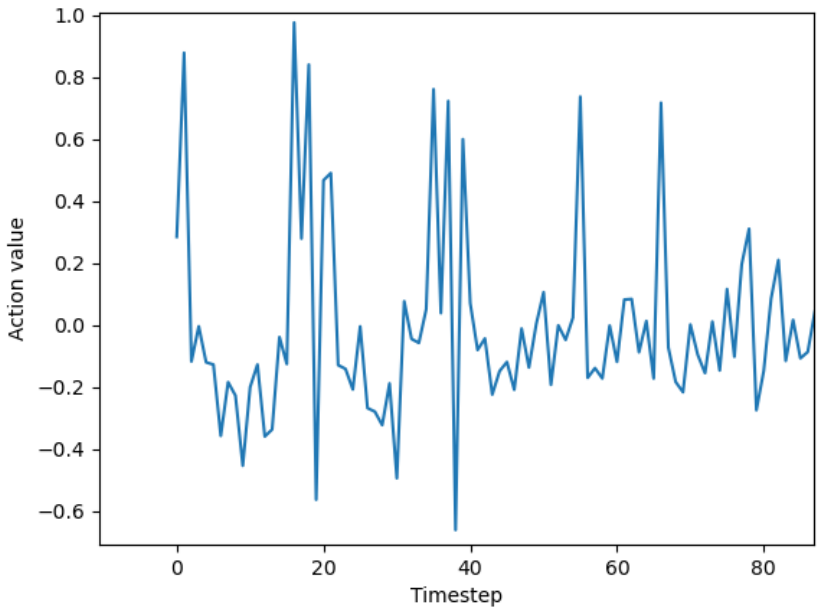

Figure 5.20: Crop of recorded actions of a continuous-action REINFORCE agent.

navigate the environment 'just in time'.

5.3 TPG and External State Vector

5.3.1 Discrete-action Environment

For this experiment 20 TPG agents with an external memory vector were evolved over 100 generations in the discrete-action CartPole environment. Actions were derived from the external vector as described in section 4.3.2, i.e. via a linear perceptron with fixed weights that receive the external state vector as input, feed the output to a sigmoid function and round the result to either 0 or 1. Results were collected using the same method for recording TPG scores used in the experiments with REINFORCE, described in section 5.2. Figures 5.21, 5.22 and 5.15 show the relative complexity of the TPG agents produced. The average team count, learner count and opcode counts all approximately double when compared to the agents produced when no external memory vector is used and actions are derived as per canonical TPG. This may be ascribed to the doubling of the number of generations assumed for training relative to the previous experiments.

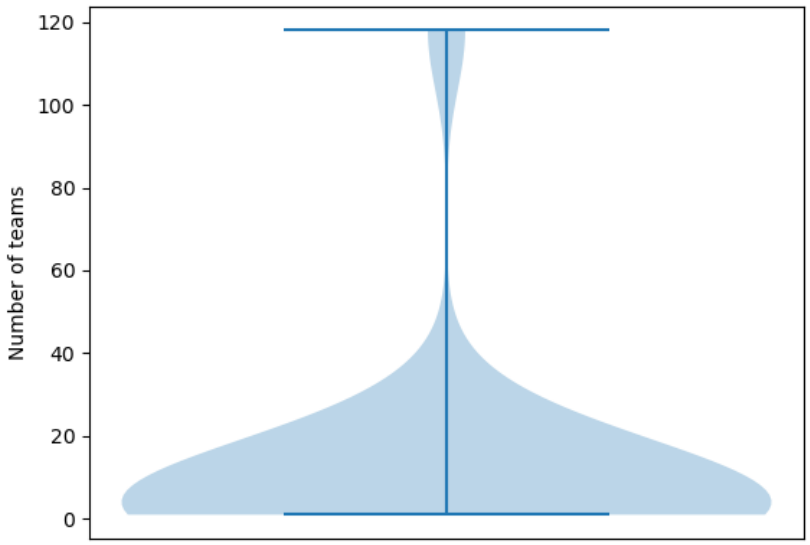

Figure 5.21: Distribution of total Team counts across TPG agents used in the discrete-action external memory vector experiment.

Figure 5.24 shows the training curves collected during evolution. As shown, the

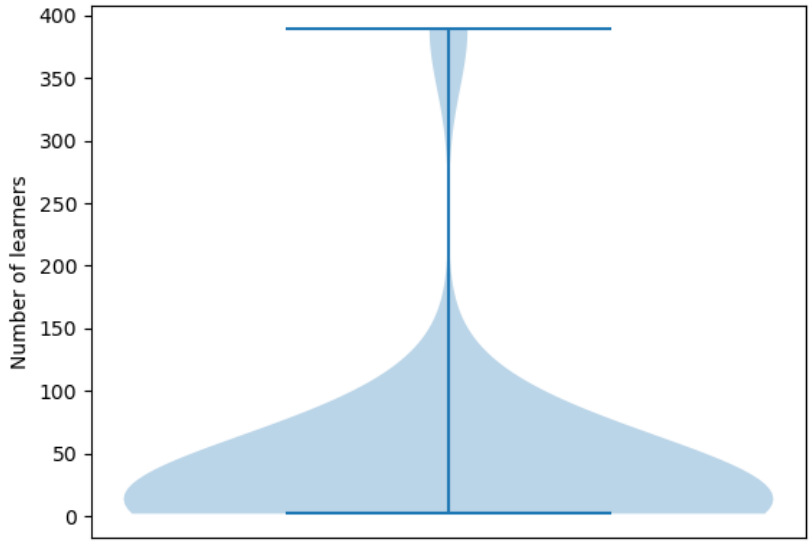

Figure 5.22: Distribution of total Learner counts across TPG agents used in the discrete-action external memory vector experiment.

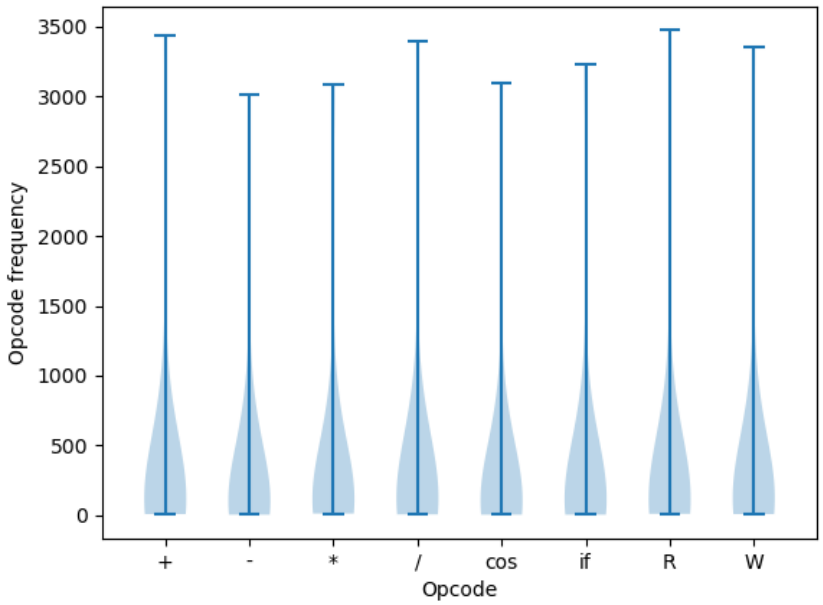

Figure 5.23: Distribution of op-code counts across TPG agents used in the discrete-action external memory vector experiment.

top- and median-performing agents quickly achieve a score of 100% in less than 50 generations. Even the worst-performing agent displays some level of capability and improvement during evolution, hovering around the 300.0 mark.

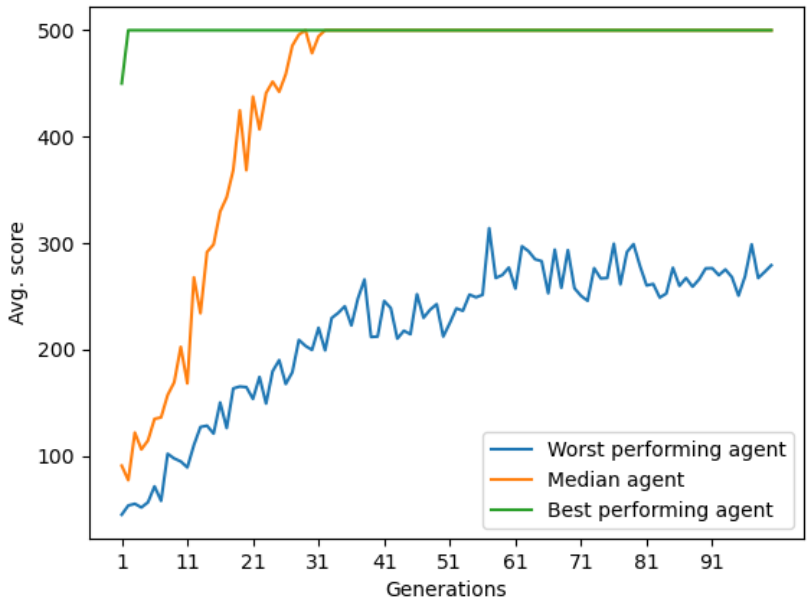

Figure 5.24: TPG training curve showing the top agent scores per generation for TPG agents used in the discrete-action external memory vector experiment.

These agents were tested exactly as described in section 5.2. Figure 5.25 shows the distribution of test scores across all agents. The mean agent performance was 483.64 out of 500.0 with a standard deviation of 71.87. The top-scoring agents also achieved an average score of 500 over all 1000 episodes, which occurred in 9 out of 20 agents. Notably, the distributions of test scores are tighter than those measured during the discrete-action REINFORCE evaluations, showing that the same effect or better can be achieved by TPG alone without the addition of a gradient method.

These results indicate that TPG can be taught to produce a useful state representation in a dedicated location, here the external memory vector, and that a successful agent can be produced that derives actions exclusively from this state representation. This is encouraging given that the decision network was never updated or modified during evolution; rather, TPG learned to cater the state representation to the fixed decision network.

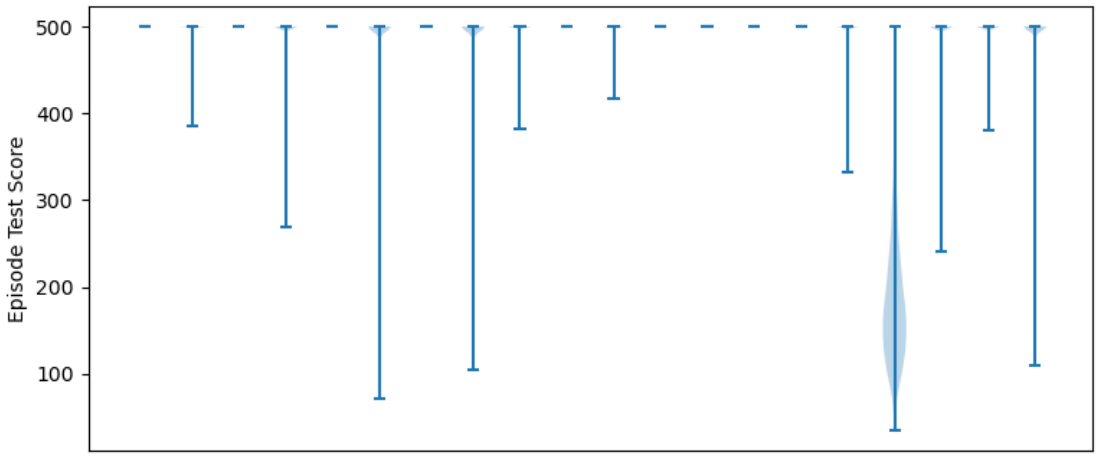

Figure 5.25: Distribution of TPG agents' test scores across 1000 test episodes for TPG agents used in the discrete-action external memory vector experiment. Each violin represents the distribution of a single TPG agent's 1000 test scores.

5.3.2 Continuous-action Environment

The same experiment was performed in the continuous-action CartPole environment and with TPG only given 50 generations to produce a useful agent. This was done in the interest of time after the previous experiment suggested that 100 generations was possibly excessive. Figures 5.26, 5.27 and 5.28 show the relative complexity of the TPG agents produced. As shown, these agents do not appear significantly more complex than those produced in the case of the discrete-action environment and in fact show reduced opcode counts in comparison. This suggests that the move to continuous actions from discrete actions, derived from the external memory vector, does not necessarily come at an increased cost of complexity in the TPG agents.

Figure 5.29 shows the training curves collected during evolution. As shown, the top- and median-performing agent both achieve a score of 100% in less than 40 generations. Similarly to the previous experiment, even the worst-performing representative agent achieves some degree of capability.

These agents were tested exactly as described in section 5.2. Figure 5.30 shows the distribution of test scores across all agents. The mean agent performance was 457.41 out of 500.0 with a standard deviation of 73.19. The scores are typically tightly

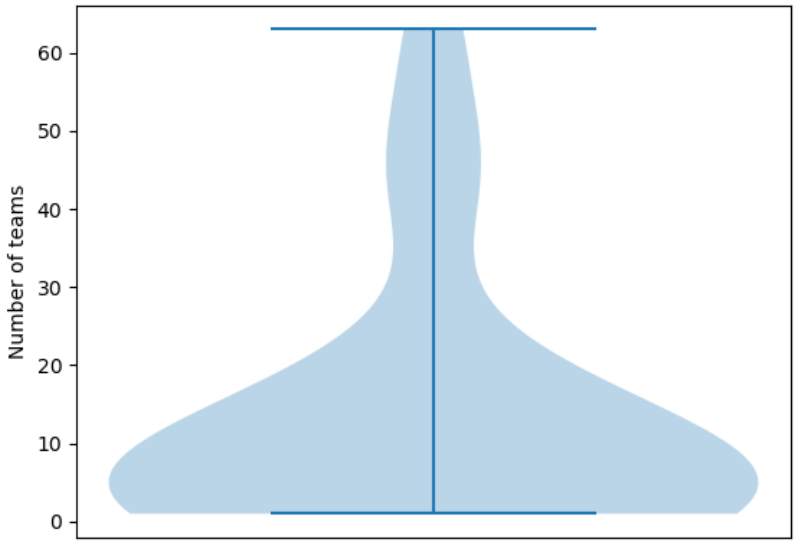

Figure 5.26: Distribution of total Team counts across TPG agents used in the continuous-action external memory vector experiment.

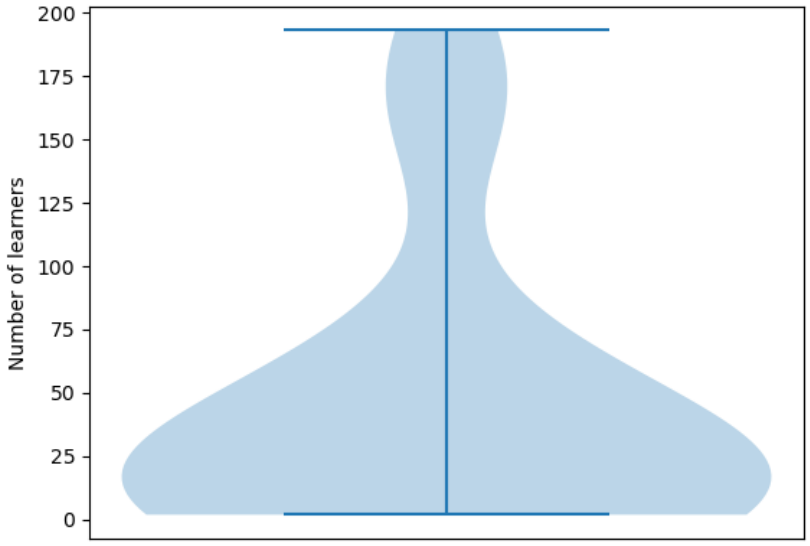

Figure 5.27: Distribution of total Learner counts across TPG agents used in the continuous-action external memory vector experiment.

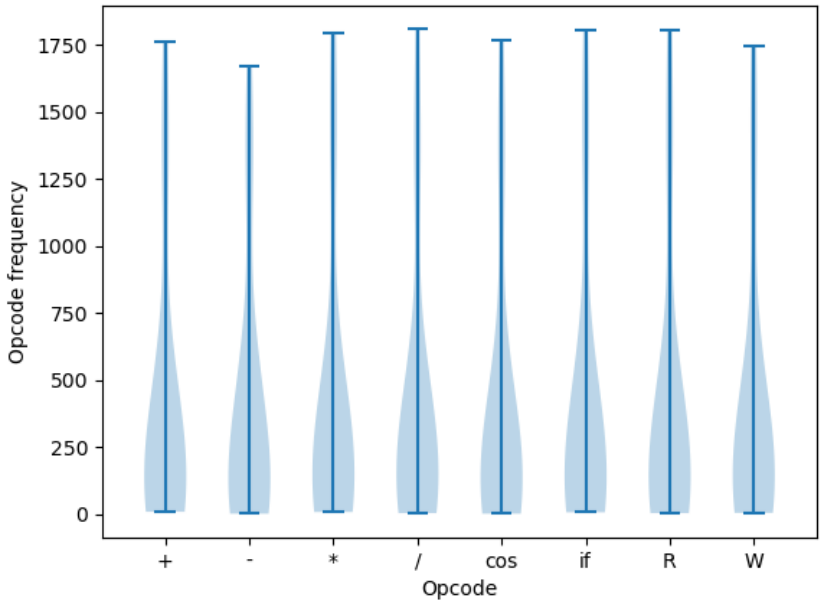

Figure 5.28: Distribution of op-code counts across TPG agents used in the continuous-action external memory vector experiment.

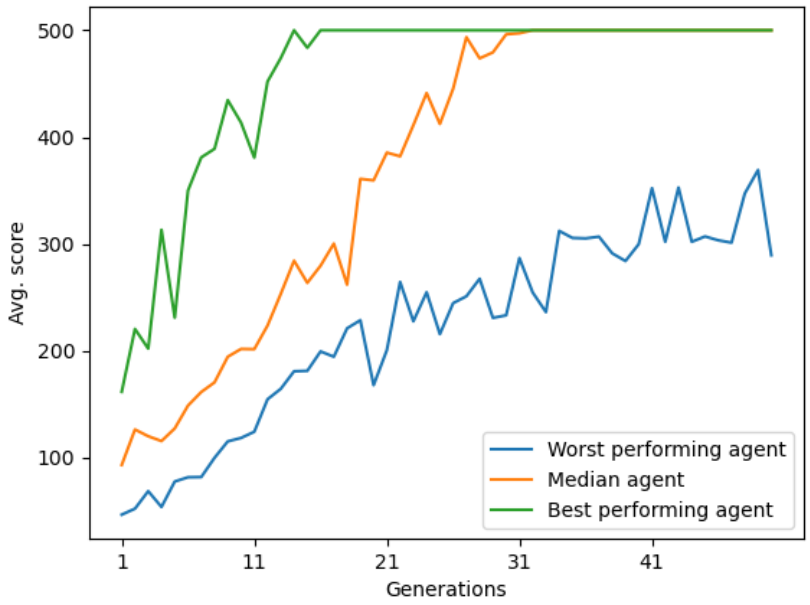

Figure 5.29: TPG training curve showing the top agent scores per generation for TPG agents used in the continuous-action external memory vector experiment.

clustered around the 100% mark across all but two agents. However the clustering is not as tight as in the case of agents operating in the discrete-action environment and a much lower number of agents achieve 100% in all test episodes. Rather, the majority of the agents tested show a minimum score below 100.0, further indicating that the continuous-action environment is more difficult to navigate.

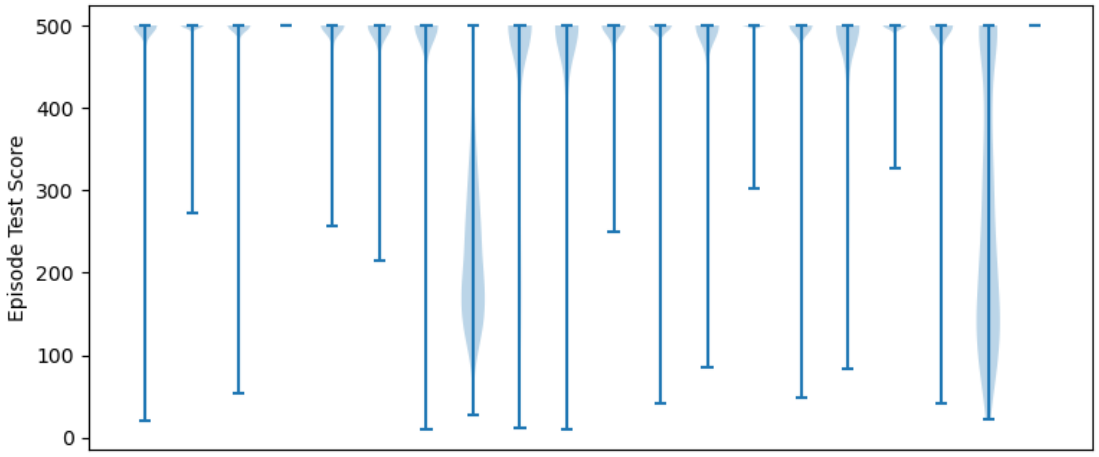

Figure 5.30: Distribution of TPG agents' test scores across 1000 test episodes for TPG agents used in the continuous-action external memory vector experiment. Each violin represents the distribution of a single TPG agent's 1000 test scores.

The first agent to achieve an average score of 100% over its test episodes was further analyzed to again characterize the selection of action values. This agent was tested in a single episode and its action values recorded. The agent scored 500.0 in this test episode. The recorded action values are shown in figure 5.19. Figure 5.20 shows a restricted portion of the same plot for better visibility into the oscillations of the actions. These plots suggest that, similarly to REINFORCE, TPG is also learning to deliberately select actions from within the allowed range of values.

Despite the drop in performance when compared to agents trained in the discrete-action environment, these results show that TPG is capable of tailoring a state representation in an external memory vector to a fixed decision network and is capable of producing useful agents that exhibit high performance in a continuous-action environment by deriving actions from this state representation.

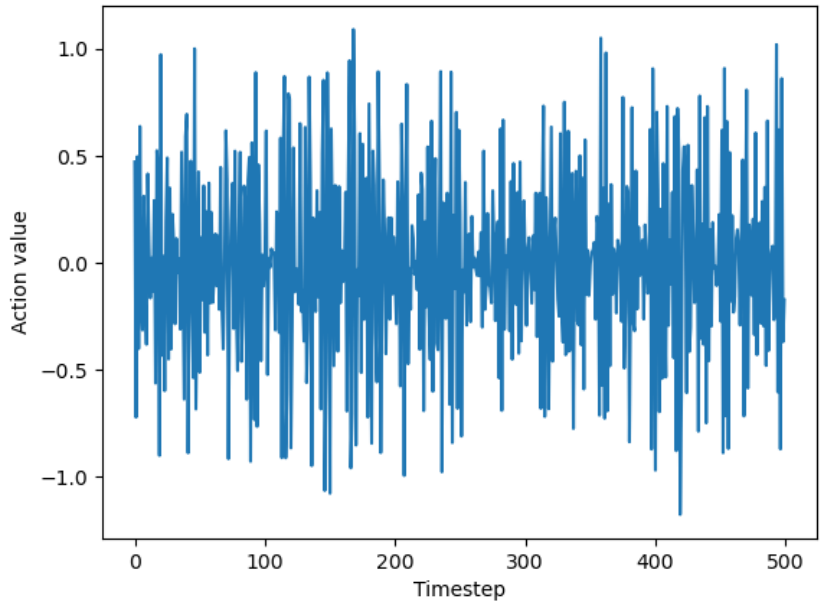

Figure 5.31: Recorded actions of a continuous-action TPG agent.

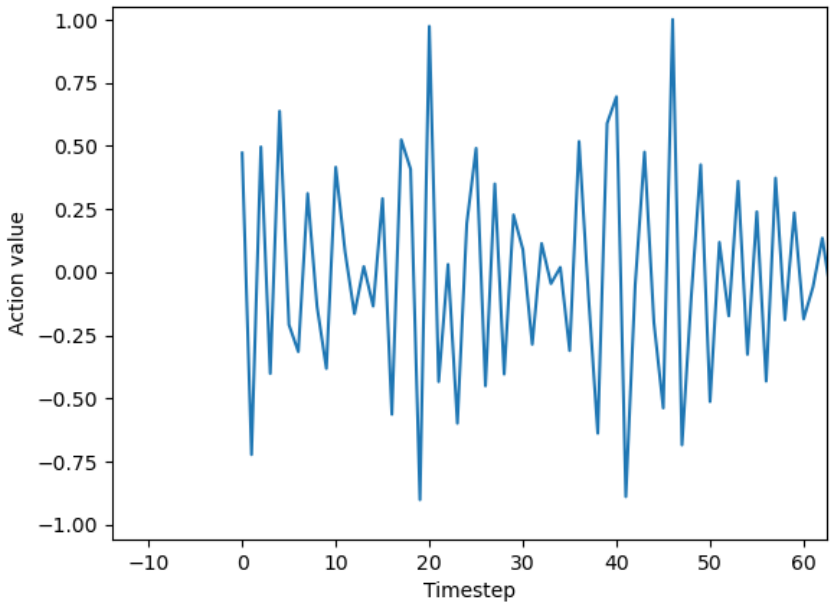

Figure 5.32: Crop of recorded actions of a continuous-action TPG agent.

Chapter 6

Conclusion

This thesis addresses the specific question of how to generalize the operation of the Tangled Program Graph framework to real-valued actions under a reinforcement learning context. To do so, two proposals are made. The first augments a TPG solution with a single weight vector that is used to derive a real-valued action using the REINFORCE learning algorithm. This implies that TPG is first trained on a discrete version of the task before REINFORCE can be deployed. In the second proposal, TPG is trained with 'indexed memory' (a representation of internal state) and a fixed weight vector assumed to directly derive a corresponding real-valued action. This means that there is only a single cycle of training and the task can be directly solved using a real-valued action. The results of both the attempt to derive actions from TPG's incidentally-produced state representation (register 0 values) via REINFORCE and the attempt to derive actions from its intentionally-produced state representation (external memory vector) successfully produced agents capable of solving the real-valued action configuration of the CartPole environment. This validates the re-framing of TPG as a combination of both a feature-engineering algorithm and an action-selection algorithm, with the output of the former being fed to the latter. It further indicates that the first component can be decoupled from the second and is capable of producing a robust state representation useful to multiple action-selection mechanisms. The success of both approaches in producing solutions to the real-valued action CartPole environment and the validation of the reframing described above confirm that TPG can be extended to perform real-valued actions and provide a method through which TPG can be modularized to accommodate a variety of action-selection algorithms.

While REINFORCE's performance provided the initial evidence to validate this approach to deriving actions, the algorithm using a combination of TPG and REINFORCE is too cumbersome and constrained to be used in real-world reinforcement

learning scenarios, due largely to its requirement that a discrete-action version of the environment be available. However, the final algorithm in which actions are calculated from TPG's memory vector has not only been shown to be effective, but is usable 'out-of-the-box'. The algorithm as implemented in this work does not suffer any additional constraints when compared to canonical TPG, requiring only the addition of the simple decision-making network. Additionally, though this was not a goal of this work, the results also suggest that the penalty in increased complexity of the resulting agents when using this algorithm is relatively minor.

Though encouraging, this work serves largely as a proof-of-concept and would benefit from future experiments to further investigate the external state-based algorithm's usefulness as well as perhaps extend its applicability. This future work might include:

- Evaluating the algorithm in more difficult environments. The algorithms in this work were presented with a readily tractable environment and their capabilities in more complex environment when compared to canonical TPG is unknown.

- Extending the decision-making network to multiple simultaneous actions. Many environments allow for or require a combination of multiple discrete and/or real-valued actions to be taken at each timestep. This could be achieved by further dividing up the memory matrix such that different sections of memory are fed into different action-selection networks.

- Extending the memory-based algorithm so that the decision network might be improved using gradient methods. This would effectively be a combination of both of the algorithms described in this work and would entail REINFORCE (or any gradient method) being applied to the external memory vector so that after TPG had settled on an architecture, the final decision-making network might be improved.

- Evaluate different gradient-based methods for improving the decision-making network (e.g. TD methods such as the Actor-Critic).

Regardless of work to be done in the future, the result of these experiments is a working, practical algorithm for deriving high-performance TPG agents capable of operating in a continuous-action environment. In terms of related algorithms, as noted in

the introduction, TPG represents a framework that produces emergent solutions with comparable quality to those identified using deep learning under visual reinforcement learning benchmarks such as ALE [17] [16] or VizDoom [30]. Neural-evolutionary methods such as NEAT and HyperNEAT are capable of providing real-valued outputs. However, when empirically evaluated under the suite of 50 ALE game titles [14], they have not scaled as well as TPG [16]. Future research will continue to characterize the problem characteristics that discriminate between each representation.

Bibliography

[1] Darrell Whitley et al. “Genetic Reinforcement Learning for Neurocontrol Problems”. In: *Machine Learning* 13 (1993), pp. 259–284.

[2] Thomas Bäck, T. Fogel, and D. Michalewicz. “Evolutionary Computation 1 (Basic Algorithms and Operators)”. In: (Jan. 2000).

[3] A. G. Barto, R. S. Sutton, and C. W. Anderson. “Neuronlike adaptive elements that can solve difficult learning control problems”. In: *IEEE transactions on Systems, Man, & Cybernetics* 13.5 (1983), pp. 834–846.

[4] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems”. In: *Artificial Neural Networks: Concept Learning*. IEEE Press, 1990, pp. 81–93. ISBN: 0818620153.

[5] M. G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. URL: http://dx.doi.org/10.1613/jair.3912.

[6] C. Boutilier, T. Dean, and S. Hanks. “Decision-Theoretic Planning: Structural Assumptions and Computational Leverage”. In: *Journal of Artificial Intelligence Research* 11 (July 1999), pp. 1–94. ISSN: 1076-9757. DOI: 10.1613/jair.575. URL: http://dx.doi.org/10.1613/jair.575.

[7] M. Brameier and W. Banzhaf. “A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining”. In: *Trans. Evol. Comp* 5.1 (Feb. 2001), pp. 17–26. ISSN: 1089-778X. DOI: 10.1109/4235.910462. URL: https://doi.org/10.1109/4235.910462.

[8] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[9] R. Collobert, K. Kavukcuoglu, and C. Farabet. “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn, NIPS Workshop*. 2011.

[10] Kenneth De Jong, David Fogel, and Hans-Paul Schwefel. "A history of evolutionary computation". In: Jan. 1997, A2.3:1–12.

[11] John Doucette, Peter Lichodzijewski, and Malcolm Heywood. "Hierarchical Task Decomposition through Symbiosis in Reinforcement Learning". In: *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation* (July 2012). DOI: 10.1145/2330163.2330178.

[12] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. "Accelerated Neural Evolution through Cooperatively Coevolved Synapses". In: *J. Mach. Learn. Res.* 9 (June 2008), pp. 937–965. ISSN: 1532-4435.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[14] Matthew Hausknecht et al. "A Neuroevolution Approach to General Atari Game Playing". In: *IEEE Trans. Comput. Intell. AI Games* 6.4 (Dec. 2014), pp. 355–366. DOI: 10.1109/tciaig.2013.2294713. URL: http://dx.doi.org/10.1109/TCIAIG.2013.2294713.

[15] Jeff Heaton. *An Empirical Analysis of Feature Engineering for Predictive Modeling*. 2017. arXiv: 1701.07852 [cs.LG].

[16] Stephen Kelly and Malcolm Heywood. "Emergent Tangled Graph Representations for Atari Game Playing Agents". In: Mar. 2017, pp. 64–79. DOI: 10.1007/978-3-319-55696-3_5.

[17] Stephen Kelly and Malcolm I. Heywood. "Emergent Solutions to High-Dimensional Multitask Reinforcement Learning". In: *Evol. Comput.* 26.3 (Sept. 2018), pp. 347–380. ISSN: 1063-6560. DOI: 10.1162/evco_a_00232. URL: https://doi.org/10.1162/evco_a_00232.

[18] M. Kempka et al. "ViZDoom: A Doom-based AI research platform for visual reinforcement learning". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. 2016, pp. 1–8.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262111705.

[20] Peter Lichodzijewski and Malcolm Heywood. "Coevolutionary bid-based genetic programming for problem decomposition in classification". In: *Genetic Programming and Evolvable Machines* 9 (Dec. 2008), pp. 331–365. DOI: 10.1007/s10710-008-9067-9.

[21] Peter Lichodzijewski and Malcolm Heywood. "Managing team-based problem solving with symbiotic bid-based genetic programming". In: Jan. 2008, pp. 363–370. DOI: 10.1145/1389095.1389162.

[22] Peter Lichodzijewski and Malcolm Heywood. "Pareto-coevolutionary genetic programming for problem decomposition in multi-class classification". In: Jan. 2007, pp. 464–471. DOI: 10.1145/1276958.1277058.

[23] Peter Lichodzijewski and Malcolm I. Heywood. "Symbiosis, Complexification and Simplicity under GP". In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: Association for Computing Machinery, 2010, pp. 853–860. ISBN: 9781450300728. DOI: 10.1145/1830483.1830640. URL: https://doi.org/10.1145/1830483.1830640.

[24] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[25] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[26] David E. Moriarty and Risto Miikkulainen. "Efficient Reinforcement Learning Through Symbiotic Evolution". In: *Machine Learning* AI94-224 (1996). Ed. by Leslie Pack Kaelbling, pp. 11–32. URL: http://nn.cs.utexas.edu/?moriarty:mlj96.

[27] Miguel Nicolau, Marc Schoenauer, and W. Banzhaf. *Evolving Genes to Balance a Pole*. 2010. arXiv: 1005.2815 [cs.AI].

[28] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[29] Liviu Panait, Sean Luke, and R. Paul Wiegand. “Biasing Coevolutionary Search for Optimal Multiagent Behaviors”. In: *IEEE Trans. Evol. Comput.* 10.6 (2006), pp. 629–645.

[30] Robert J. Smith and Malcolm I. Heywood. “Scaling Tangled Program Graphs to Visual Reinforcement Learning in ViZDoom”. In: *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*. Vol. 10781. Lecture Notes in Computer Science. Springer, 2018, pp. 135–150.

[31] Robert Smith and Malcolm Heywood. “A Model of External Memory for Navigation in Partially Observable Visual Reinforcement Learning Tasks”. In: Mar. 2019, pp. 162–177. ISBN: 978-3-030-14811-9. DOI: 10.1007/978-3-030-16670-0_11.

[32] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[33] Richard S. Sutton et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS’99. Denver, CO: MIT Press, 1999, pp. 1057–1063.

[34] Ronald J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.