Chapter 1
Evolving Many-Model Agents with Vector and Matrix Operations in Tangled Program Graphs

Tanya Djavaherpour, Ali Naqvi, Eddie Zhuang, Stephen Kelly

Abstract Tangled Program Graphs (TPGs) are highly modular, hierarchical representations for genetic programming that are well-suited to multitask learning in temporal sequence prediction tasks such as control and time series forecasting. In this work, we expand the simple scalar register machines traditionally used in TPGs to include vector and matrix memory and operations. This helps TPGs evolve versatile agents that are capable of solving partially-observable control and forecasting problems simultaneously. A single agent can predict actions in discrete and continuous control tasks, as well as perform generative time-series prediction.

1.1 Introduction

Life presents an endless stream of unique problems that require multi-faceted solutions. We find several examples of emergent systems that generate multi-faceted problem solvers in nature and nature-inspired computation. A common theme in these biological and artificial life systems is the emergence of *many-model* solvers, in which general-purpose structural components are repeated and repurposed many times in unique, specialized contexts. We see this, for example, in the many-model structure of the neocortex (in reality and simulation), where tens of thousands of models interact and compete to organize perception, thought, and action [1,6,10,22]. Evolution and plasticity of mental models are fundamental to the emergence of the many-model brains in nature. It stands to reason that artificial life systems incorporating these processes (e.g. [23], [21]) show the greatest potential to reproduce the brain's efficiency, speed of learning new tasks, and general problem solving power.

TPGs were initially designed to solve visual reinforcement learning problems [13] and have since been extended for supervised image classification [26], time series prediction [17], radio fingerprint identification [5], large-scale parallelisa-

Department of Computing and Software, McMaster University, Canada e-mail: {djavahet, naqvia18, zhuane4, spkelly}@mcmaster.ca

tion [8], and ultra-fast inference on embedded hardware through the generation of optimised standalone C code [7]. Automatic problem decomposition has been a theme in all these works, with the modular structure of TPGs lending themselves well to multitask learning problems which can benefit from the ability to discover multiple predictive models that are specialized for particular tasks, and automatically organize them into a single agent. Switching multiple models in and out of the decision control flow and runtime leads to efficient inference and helps avoid common problems associated with continual and multitask learning such as catastrophic forgetting [19] and the distraction dilemma [34].

Many recent works also explore TPGs with a focus on utilizing temporal memory [17, 18, 27, 28]. This is a natural application, since the fundamental building blocks in TPGs are Linear Genetic Programs (LGPs) [2], in which a sequence of instructions operate on input data and store intermediate values in memory registers. When registers are configured to maintain state in temporal sequence tasks, they act as a built-in temporal memory mechanism which allows agents to build and maintain a mental model of the environment on-the-fly. This a requirement for operation in environments that are only partially observable [3, 31].

Traditionally, register machines in TPGs have been limited to scalar memory and a minimal set of operations, e.g. $\{+,-,\times,\div,log,exp,cos\}$. These constraints have led to efficient, compact, and interpretable solutions [13, 28]. However, we are increasingly interested in evolving brain-inspired computational models. To do so, we study the types of memory and operations used in computational neuroscience (e.g. [6], [32]) and evaluate how TPGs might be extended to support these data structures and operations. Indeed, systems based on Linear Genetic Programming lend themselves well to neurocomputing models because there are no constraints on the types of memory structures or operations used. Multiple systems have emerged in recent years that evolve programs using the types of scalar, vector, and matrix manipulations common in modern artificial neural networks [9,24,25]. In this initial expansion of TPGs, we test the utility of vector and matrix memory and operations, and show how they improve the search for many-model problem solvers.

1.2 Multitask Learning

The objective in this work is to discover programs that can operate two partially-observable classic control systems (Acrobot and Pendulum) [4], and also perform recursive time series forecasting on two unrelated time series; Laser [11] (a real-world recording) and Mackey-Glass [20] (a chaotic series generated from a parameterized function), Figure 1.1. All 4 tasks can be characterized as temporal sequence learning problems. However, each task has unique dynamics and objectives, making it exceedingly difficult to build a single agent capable of solving them all. Addressing these unique challenges simultaneously highlights the power of many-models systems.

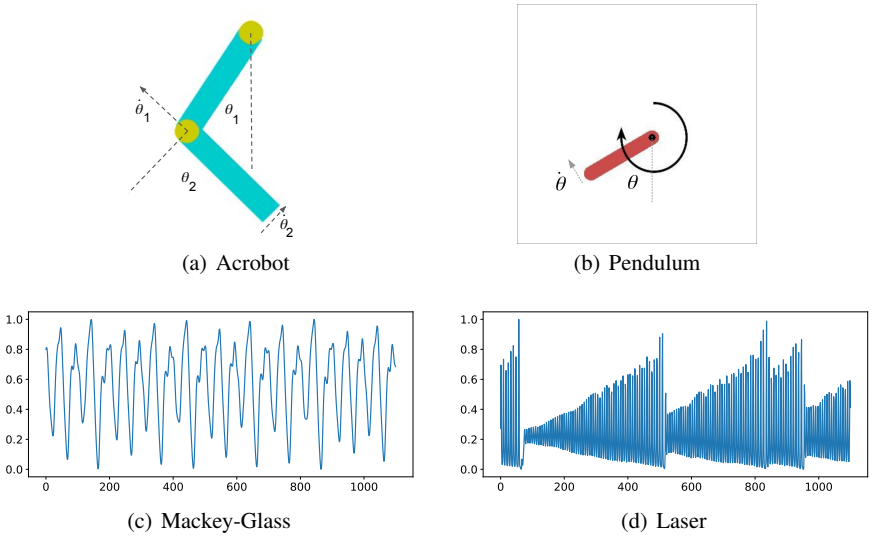
1.2.1 Reinforcement Learning Problems

Reinforcement Learning (RL) is a type of machine learning problem in which computational agents learn through trial-and-error interaction with the problem environment over time [30]. At each timestep, the agent observes its environment through sensor inputs $\mathbf{obs}(t)$, takes an action that changes the system state, and receives a reward signal that describes the desirability of its current situation. The goal is to develop behaviours that map observations to actions such that the summed reward over all timesteps is maximized. RL is characterized by a myriad of learning challenges including partial observability of state, delayed rewards, dynamic and multitask environments, and mixed discrete and continuous action spaces [16].

The unique challenge in this work requires an agent to operate in partially-observable versions of two RL benchmarks from the classic control literature: the discrete-action Acrobot and continuous-action Pendulum, Figure 1.1. In the Acrobot task, two links form a double pendulum with one end fixed and the other end free, both connected by an actuated joint. The state of the system at each timestep, $\mathbf{obs}(t)$, is described by 2 state variables including angles of the first and second joint, θ_1 and θ_2. Angular velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$, are not observable to the agent. The system is controlled by applying a torque of +1, -1, or 0 to the joint between the links. The objective is to swing the free end of the lower link to reach a predetermined height in the fewest possible steps. Each timestep that fails to meet this target incurs a penalty of -1, while successfully reaching the height concludes the task and grants a final reward of 0. The episode ends after 200 timesteps. In the Pendulum task, a pole is attached at one end to a fixed point and the other end is free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position and balance it with its center of gravity directly above the fixed point. The task ends after 300 timesteps. Both control tasks are episodic, with the agent's fitness equal to its average reward over 20 episodes with unique initial conditions. A post-training test procedure measures the agent's average reward over 100 episodes with start conditions not seen during training. Reward functions are shown in Table 1.1.

Table 1.1 Definition of rewards for control tasks, provided to the agent when an episode ends due to success, failure, or a time constraint. t_{max} is the max timesteps per episode. In the Pendulum task, $\dot{\theta}$ is the angular velocity, $Torque \in (-1,1)$ is the torque applied to the joint, and ϕ is a function to normalize the pole angle: $\phi(\theta) = ((\theta+\pi) \mod (2\times\pi)) - \pi$.

Task	Episode Reward	t_{max}
Acrobot	$\sum_{t=1}^{t_{end}} -1.0$	200
Pendulum	$\sum_{t=1}^{t_{max}} -(\phi(\theta)^2 + 0.1\times\dot{\theta}^2 + 0.001\times Torque^2$	300



(a) Acrobot (b) Pendulum

(c) Mackey-Glass (d) Laser

Fig. 1.1 Problems environments used in this work. For complete details on control tasks see [4] and for time series tasks see [11, 20].

1.2.2 Recursive Time Series Forecasting Problems

The goal of time series forecasting is to predict (unseen) future values based on previously observed values. To do so, the agent is fed individual samples in order from series $x()$ and, given sample $x(t)$, must predict the value of $x(t+1)$. Our choice of datasets and methodology was inspired by [33] and this study follows prior evaluations of TPGs in forecasting tasks [17]. The Laser and Mackey-glass datasets are univariate and contain 1100 samples normalized to the interval $[0,1]$.

Before predictions begin at $x(t)$, the agent is executed for each input sample in series $x(t-50)$, $x(t-49)$, ..., $x(t-1)$ and the agent's output values are ignored. When this *priming* phase is complete, recursive forecasting is used to predict future values. That is, after $x(t-1)$, true samples from $x()$ are no longer used as input. Instead, the agent's predictions are fed back as input to predict future values. For example, the model's output for $x(t)$ becomes its input observation at $t+1$, and so on. Recursive forecasting allows predictions to any horizon.

Our configuration for training evaluation uses slices of the dataset as training episodes. Beginning at t_0, agents are *primed* with 50 samples, and the fitness function measures how well they recursively predict the next fifty samples, repeating with start points from $t_{50}, t_{100}, ..., t_{950}$. Thus, solution fitness is the MSE over 950 predictions in total. In our setup, $\mathbf{obs}(t)$ stores the 32 most recent input values in a first-in-first-out buffer initialized with zeroes.

A validation procedure measures how well each agent recursively forecasts beyond the horizon used during training. Specifically, agents are used to predict the

next 100 samples starting from $t_{100}, t_{200}, ...t_{900}$. MSE over the last 50 predictions from each validation set (a total of 450 predictions) is used as the validation score. To obtain a final test score, the single program graph with the best validation score is used to predict the next 100 samples from t_{1000} (i.e, the model is primed with samples $\mathbf{s}(t_{950}...t_{999})$).

Note that $\mathbf{obs}(t)$ never contains a task-label input to identify which task is currently being experienced by the agent. As such, successful multitask agents must 1) identify the task by tracking how observation variables change over time, and 2) predict the best task-specific action/output at each timestep.

1.2.3 Multitask Fitness

During evolution, each agent in the population is evaluated on each task independently, resulting in 4 task-specific *raw fitness* scores. Agents are then assigned a *multitask* fitness for each set $s \in \mathbb{P}$ (the power set of 4 task combinations). For single task sets, multitask fitness is simply the the training fitness in that task. For multitask sets, fitness captures how well an agent performs on multiple problems by ranking each agent by their weakest performance in the problem set. To achieve this, every agent's mean reward on each task is normalized relative to the rest of the current population. Normalized score for agent a_i on task t_j is calculated as:

$$sc^{nrm}(a_i,t_j) = (sc(a_i,t_j) - sc_{min}(t_j))/(sc_{max}(t_j) - sc_{min}(t_j)) \tag{1.1}$$

where $sc(a_i,t_j)$ is the mean score for agent a_i on task t_j and $sc_{min,max}(t_j)$ are the population-wide min and max mean scores for task t_j. Fitness for agent a_i is then $min(sc^{nrm}(a_i,t_{\{1..n\}})$, or the minimum normalized score for agent a_i over all problems. n denotes the number of problems. Thus, *champions* are the agents with the highest minimum normalized fitness over all problems in each set. Normalizing rewards is a critical part of quantifying multitask fitness and mitigates the *distraction dilemma* [34], a common issue in multitask learning in which the differing magnitude of reward signals may make certain tasks appear more salient.

1.3 Tangled Program Graphs

Tangled Program Graphs (TPGs) are modular, hierarchical representations for Genetic Programming that are particularly well-suited to multitask learning in partially-observable visual reinforcement learning and time series prediction tasks [17]. In short, TPGs excel at tasks which require automatic problem decomposition and temporal memory. The big idea is to construct many-model prediction machines from the bottom up, adaptively building programs, teams of programs, and graphs of teams of programs through an emergent and open-ended evolutionary search, Figure 1.2. The following subsections describe each component in detail and outline

the fundamental rules that govern their interaction with each other and the external environment.

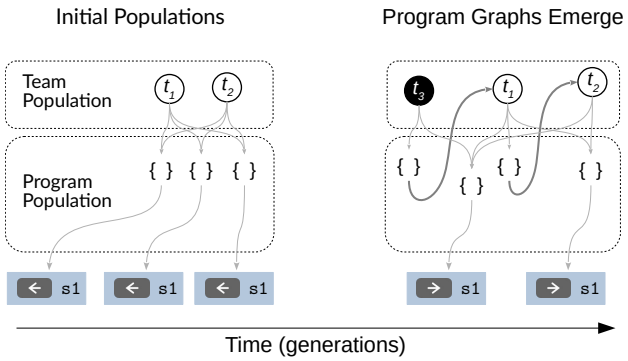


Fig. 1.2 Illustration of the relationship between teams and programs in TPG. Initially, all programs are leaf nodes. Over time, program action pointers may be modified to refer to other teams and *program graphs* emerge. When a team is subsumed into a program graph, it is cloned and the clone (t_{2c}) becomes an internal node. See Section 1.3.3 for details.

1.3.1 Programs

The foundational representation in TPGs are linear genetic programs, or register machines [2]. Register machines execute a sequence of operations which transform data stored in memory, Algorithm 1. Register memory may be organized in virtually any data structure, where the types of data structures used will define the set of operations a program may use to manipulate that data. Inspired by AutoML-Zero [25], this work supports scalar, vector, and matrix memory structures, and a wide array of potential operations, Table 1.3. Three properties of the register machine representation are particularly important for TPGs:

1. Since programs operate on memory structures, all intermediate values are stored in memory, and are therefore accessible to any other instruction in the program. In temporal sequence tasks, memory structures can maintain state between each step in the sequence. This *stateful* property allows the agent to store and integrate data over time, forming the basis of a dynamic *mental model*.
2. Any individual memory structure, or set of structures, may be interpreted as the program's *output* post-execution. As such, single or multiple outputs may be defined, and outputs may take the form of any data structures supported by the machine.
3. Instructions that have no effect on the value of output registers can be identified prior to execution and skipped during evaluation of the agent. A significant

proportion of instructions may be ineffective in this respect. Given that each evaluation in temporal sequence tasks may require hundreds of program executions, skipping these *intron* instructions can have a large impact on the speed and efficiency of agents.

Algorithm 1 Example program (register machine). Each program contains 8 instances of each memory type, scalar `s`, vector `v`, and matrix `m`. All integers in the program (e.g. numbers following `s,v,m`) are evolved constants used for memory indexing. m_w is an evolved integer constant, unique to each program, which specifies the dimensionality of vector and matrix memory, sized $(m_w, 1)$ and (m_w, m_w) respectively. o_i is an evolved integer constant specifying this program's observation offset index. $\mathbf{obs}(t)$ represents the observable state at time t. Python code in lines 1-3 copy $\mathbf{obs}(t)$ to `v0` and `m0`, making it accessible to vector and matrix operations later in the program (See Figure 1.3 for an example). Scalar operations can access $\mathbf{obs}(t)$ directly (e.g. line 7). All memories are stateful, meaning their contents are reset to zero at the beginning of each agent evaluation in a given task, and thereafter left to accumulate intermediate values over time. Programs have two return values (line 11) which are interpreted as the weight (confidence) value and action output. In this example line 8 does not affect the final value of `s0` or `s1`. Ineffective instructions are useful for the evolutionary search [35], but for efficiency they can easily be identified and skipped during program execution [2]. A complete list of operations and instruction formats appears in Table 1.3.

```
1: v0 = numpy.roll(obs(t), -o_i)[:m_w]          ▷ Copy observation to vector memory
2: vi = numpy.roll(obs(t), -o_i)[:m_w*m_w]      ▷ Copy observation to temporary vector vi
3: m0 = vi.reshape(m_w,m_w)                     ▷ Copy observation to matrix memory
4: v3 = s0*vi                                   ▷ Program execution begins
5: v3 = s0*vi
6: s0 = mean(v3)
7: s0 = s0/vi[3]
8: s3 = norm(mi)                                ▷ Intron
9: s1 = cos(s0)
10: if (s0 < vi[2]): s0 = -s0
11: return s0,s1                                ▷ weight, continuous action
```

Programs represent the smallest distinct instance of a predictive *model*, Algorithm 1. In this work, they have two return values which are interpreted in unique ways. Scalar `s0` can be characterized as a *confidence* value which predicts the suitability of this program's other return values relative to the current state of the system. Programs also return scalar `s1`, which can be used as a continuous-valued *action* in control tasks, or a prediction variable in time series forecasting. Finally, to support discrete control tasks, programs are also associated with one task-specific discrete action pointer, See Figure 1.2. In short, the role of programs is to learn the appropriate *contexts* in which their continuous output register (`s1`) and/or discrete action should be deployed to the environment.

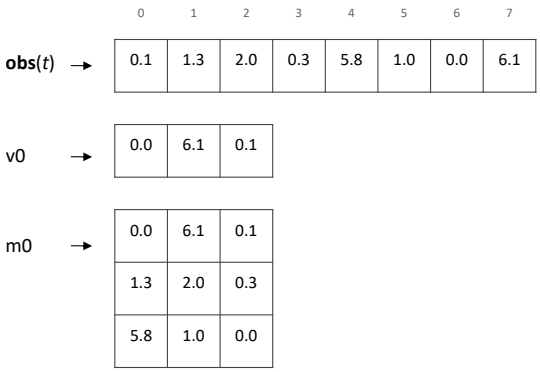


Fig. 1.3 Illustration of how vector observation data $\mathbf{obs}(t)$ is copied to vector and matrix program memory, lines 1-3 in Algorithm 1. In this example, $\mathbf{obs}(t)$ is size 8, the program's evolved memory dimensionality m_w is 3, and its evolved observation offset index o_i is 6. This methodology allows for observation data to be represented as any size vector or matrix. Since programs have their own mutable memory dimensionality and offset, they are free to access any part of the observation through a *window* of this size.

1.3.2 Teams of Programs

In order to explicitly encourage problem decomposition, individual programs do not work alone. Instead, a *team of programs* is the basic representation for a stand-alone agent in TPGs. Each team is represented as a list of programs that *collectively* map input observations $\mathbf{obs}(t)$ to a pair of discrete and continuous actions. Teams can be thought of as vertices in a computational graph where the programs represent directed, dynamically-weighted *edges*. For example (See Figure 1.2), given observation $\mathbf{obs}(t)$, all programs in team t_1 will be executed in order. The contents of each program's `s0` is interpreted as its weight. The edge with the largest weight defines the agent's output at time t. In this work, we require a discrete action (for discrete control tasks such as Acrobot) and a single continuous value (for continuous control tasks such as Pendulum and time series prediction). As such the *atomic* (i.e. terminal) program returns its discrete action pointer and the value stored in `s1`. Note that any type of memory may be used for output, implying this framework is extensible to various task types (e.g. returning a vector memory instead of a scalar would allow this representation to be used in multivariate time series prediction or robot control requiring multiple continuous control variables [15].

1.3.3 Graphs of Teams of Programs

Programs and Teams are stored in separate populations and coevolved, Figure 1.2. Initially, each team is composed of 3 unique programs and all programs are leaf nodes, as in the left-hand side of Figure 1.2. Evolution is driven by a generational

Genetic Algorithm (GA) in which parent selection identifies the most promising root teams [1], which are then modified by crossover and mutation operators.

Team variation operators may change the program order or add, remove, and mutate programs in the team. Team complement as well as program length and content are all adapted properties. When a program is modified by variation operators, it will remain a leaf with probability p_{atomic}, and will otherwise connect to one team from the set of teams present from any previous generation. These connection mutations are the mechanism by which TPGs support compositional evolution, adaptively recombining multiple (previously independent) teams into graph structures, or *program graphs*, Right-hand side of Figure 1.2.[2]

Execution of a program graph begins at the root team (t_3 in Figure 1.2), where all programs in the team will execute in order. Graph traversal then follows the program with the largest weight, repeating the execution process at every team along the path until a leaf node is reached. Thus, the graph computes one path from root to leaf at each timestep, where only a subset of programs in the graph (those in teams along the path) require execution. This process is illustrated in Algorithm 2.

Algorithm 2 Selecting an action through traversal of a program graph. P is the current program population. tm_i is the current team (initially a root node). $\mathbf{obs}(t)$ is the state observation. V is the set of teams visited throughout *this* traversal (initially empty). First, all programs in tm_i are executed relative to the current state $\mathbf{obs}(t)$ (Lines 4,5). The algorithm then considers each program in order of bid (highest to lowest, Line 7). If the program is a terminal node, its discrete and continuous action pair is returned (Line 8). Otherwise, if the program's action points to a team that has not yet been visited, the procedure is called recursively on that team. Thus, while a program graph may contain cycles, they are not followed during traversal. To ensure an atomic is always found, team variation operators are constrained such that each team maintains at least one program that has an atomic action.

1: **procedure SelectAction**(tm_i, $\mathbf{obs}(t)$, V)
2: $A = \{x \in P : x \text{ has atomic action}\}$
3: $V = V \cup tm_i$ ▷ add tm_i to visited teams
4: **for all** $p_i \in tm_i$ **do**
5: $bid(p_i) = exec(p_i, \mathbf{obs}(t))$ ▷ run prog. on $\mathbf{obs}(t)$ and save result
6: $tm_i' = sort(tm_i)$ ▷ sort progs by bid, highest to lowest
7: **for all** $p_i \in tm_i'$ **do**
8: **if** $p_i \in A$ **then return** $action(p_i)$ ▷ atomic reached
9: **else if** $action(p_i) \notin V$ **then**
10: **return** $SelectAction(action(p_i), \mathbf{obs}(t), V)$ ▷ delegate action (graph edge)

[1] During selection, fitness ties are broken by comparing the inference complexity of agents, with lower complexity preferred. This helps control bloat by reducing the likelihood of agents complexifying without any associated fitness gain (See Section 1.4.2).

[2] Complete details are available in [12, 13, 18].

The hierarchical inter-dependency between teams emerges entirely through interaction with the task environment. The subset of teams/programs that require execution is dynamically selected at run-time as a function of the agent's input observation and the state of its mental model (i.e. memory). This many-model representation allows evolution to build agents for multiple problems simultaneously [18].

1.4 Experimental Results

Experiments in this work are primarily designed to benchmark the utility of vector and matrix operations in TPGs. Results suggest that these extensions can significantly improve multitask sequence learning in the challenging 4-task scenario described in Section 1.2. This section discusses these findings in terms of problem solving ability and inference complexity of the resulting multitask agents.

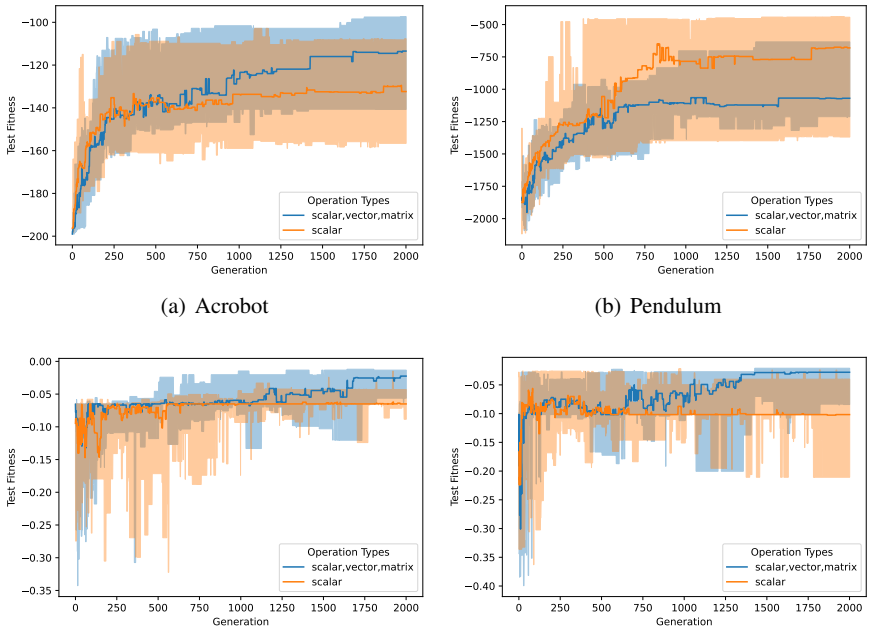
1.4.1 Problem Solving Ability

Figure 1.4 reports test fitness at each generation for multitask champions interacting with each problem. Two independent experimental cases are shown, scalar-only TPGs and TPGs extended with vector and matrix operations. We run 50 repeats of each experiment, with hyperparameters listed in Table 1.2. In the Mackey-Glass and Laser tasks, TPGs with vector and matrix operations are consistently outperforming those without. Figure 1.4 shows the median fitness (solid line) and min/max fitness (shaded area) for the single best multitask agent in each repeat. The Mann-Whitney U test is applied to compare the final test fitness for the pair of experiments in each plot. The *scalar, vector, matrix* case is significantly better in Mackey-Glass and Laser ($p < 0.05$), while the *scalar* case is better in Pendulum ($p < 0.05$), and no significant difference is found in Acrobot. Overall, only the *scalar, vector, matrix* case is able to discover a single multitask agent capable of solving all 4 tasks. That is, the agent can swing up the Acrobot, swing-up and balance the Pendulum, and achieves test MSE in the recursive forecasting tasks which matches the best results achieved by single-task agents in previous work [3] [14]. This agent's prediction accuracy in recursive forecasting is shown in Figure 1.5.

1.4.2 Inference Complexity

Figure 1.6 provides the average number of program instructions executed per prediction by the best agent as it interacts with each task. *complexification* [29] is clearly

[3] The best *scalar,vector,matrix* agent in Figure 1.4 achieves a test score of 0.0144 and 0.0127 for Mackey-Glass and Laser, respectively

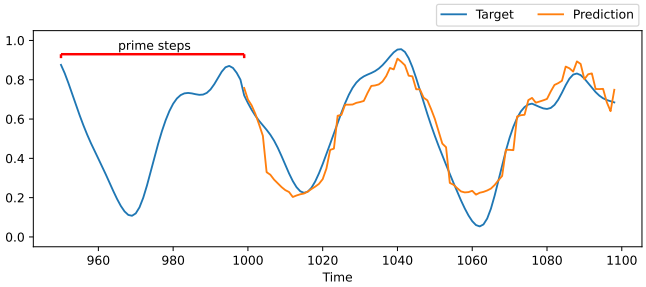


(a) Acrobot (b) Pendulum

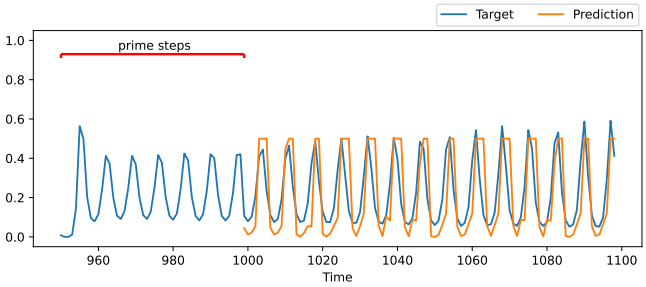
(c) Mackey-Glass (d) Laser

Fig. 1.4 Evolutionary progress in the multitask learning challenge. Plot shows the median test fitness (solid line) and min/max fitness (shaded area) for the single best multitask agent over 50 repeats. The fitness score at each generation is produced by the *same* multitask agent which solves all 4 problems relatively well. This agent is selected from the population using the multitask ranking metric described in Section 1.2.3.

visible in most cases. The agents start simple and gradually complexify through interaction with the problems. The only significant difference found between experimental cases is in the recursive forecasting tasks. In the scalar-only experiments, fitness tends to stagnate at generation ≈ 250. It seems that evolution does not discover ways to improve simple agents, and incremental code growth does not result in consistent fitness improvements. The population tends not to complexify as a result. On the other hand, the *scalar, vector, matrix* cases exhibit incremental growth *and* steady fitness improvements beyond generation 1000 in all tasks except Pendulum. It appears that the extended memory and operation set enhances the expressive power of TPGs in the recursive forecasting tasks. We leave it to future work to analyze which specific operations are important, and to interpret how they are used within linear genetic programs.



(a) Mackey-Glass

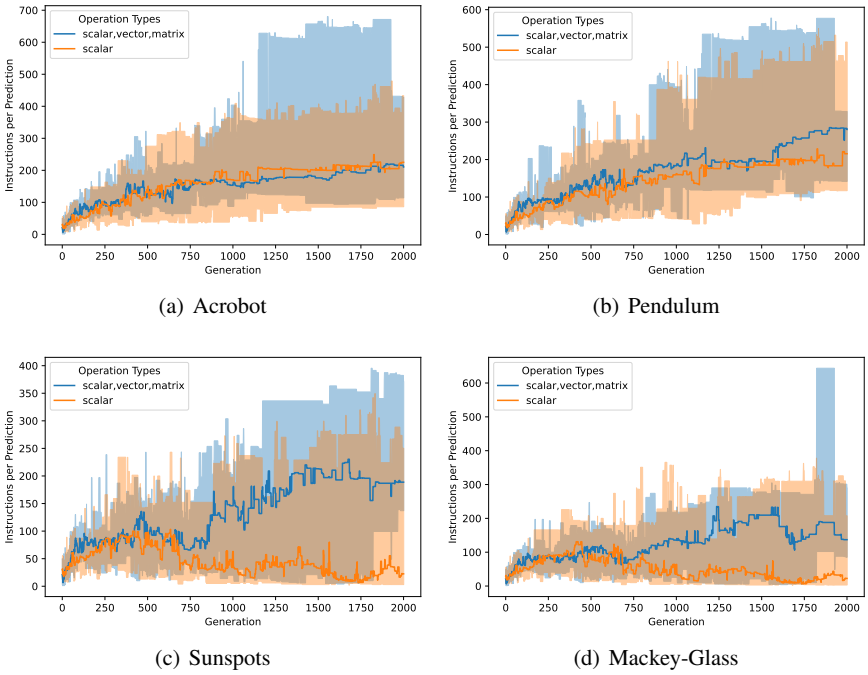


(b) Laser

Fig. 1.5 Example recursive forecasts produced by the best multitask agent with the full set of operations (See *scalar,vector,matrix* in Figure 1.4).

1.4.3 Evolved Many-Model Hierarchies

Figure 1.7 depicts an example program graph in which each node represents one team of programs. Node charts illustrate the proportion of timesteps in which each team was visited during test episodes in each task. For example, the root node is visited in every timestep, thus proportions are equal for Acrobot, Pendulum, Mackey-Glass, and Laser. It is currently unclear why long vertical paths appear traversing several teams with equivalent task distributions. This structure implies that all teams in such paths are visited together, and internal nodes are potentially redundant. In the current version of TPGs, variation only operates on root nodes, thus program graphs can only be constructed from the bottom up, and internal nodes are essentially frozen, protected from variation. This property is helpful in avoiding catastrophic forgetting in dynamic and multitask environments that require continual learning, but it comes with reduced ability for the search to fine tune behaviours by adjusting internal components of the agent's structure. Given that the example team in Figure 1.7 is competent in all tasks, we wonder if support for internal node deletion and/or variation could reduce the structural complexity of graphs without



(a) Acrobot (b) Pendulum

(c) Sunspots (d) Mackey-Glass

Fig. 1.6 Development of agent complexity. Plot shows the average number of program instructions executed per prediction by the best multitask agent as it interacts with each task. Median (solid line) and min/max (shaded area) over 50 independent repeats are shown. The Mann-Whitney U test is applied to compare final instruction counts for the pair of experiments in each plot. Significant difference in found in Mackey-Glass and Laser ($p < 0.05$). See Figure 1 for an illustrative example program.

having to *tear down* the agent from the root and build it back up. Answering this question presents an important opportunity for future work.

1.5 Discussion and Future Work

TPGs evolve many-model agents which reuse structurally-simple building blocks in complex hierarchical systems. These agents are evolved from scratch to solve several types of temporal sequence prediction problems simultaneously. We have shown that TPGs can discover multi-faceted solvers for a set partially observable problems including discrete control, continuous control, and two unrelated time series prediction tasks. For the first time, we have demonstrated the utility of vector and matrix operations in TPGs. These extensions are most useful in the forecasting tasks, which have larger observation spaces (control tasks have $\mathbf{obs}(t)$ of size 2, while recursive forecasting tasks have $\mathbf{obs}(t)$ of size 32). Furthermore, unlike the

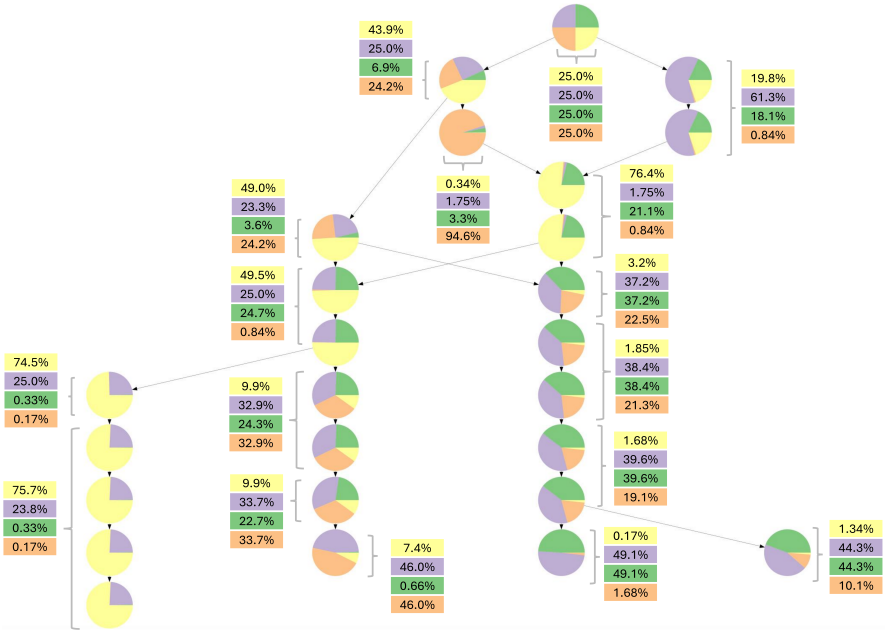


Fig. 1.7 Example program graph in which each node represents one team of programs. Node charts illustrate the proportion of timesteps in which each team was visited during test episodes in each task.

control tasks, forecasting tasks encode a time window in $\mathbf{obs}(t)$. These differences are likely the main reason why vector and matrix memory, and linear algebra operations, provides an advantage over the simple scalar register machines traditionally used in TPGs. However, this study is just the beginning. Future work will analyze which specific memories and operations are important, and interpret how they are used within TPGs. We are also interested modifying TPGs' graph-building operations in order to optimize many-model agents for efficiency. Broadly speaking, the long-term goal is to design a search space which includes the operations needed to discover new algorithms inspired by advanced neurocomputing models such as Hierarchical Temporal Memory (HTM) [6], and to define new operators which can efficiently guide evolution in this direction.

Acknowledgements This research was enabled in part by support provided by the Digital Research Alliance of Canada (alliancecan.ca) and the NSERC Discovery Grants program.

Table 1.2 Parameterization of team and program populations. n_{elite} is the number of agents to save in each generation. For the team population, p_x is the probability of crossover and p_{mx} denotes a mutation operator in which: $x \in \{d, a\}$ are the probability of deleting or adding a program respectively; $x \in \{m, n, s\}$ are the probability of creating a new program, changing a path-program's action pointer (leaf or team), and changing a program's shared memory pointer respectively. For the program population, p_x denotes a mutation operator in which $x \in \{delete, add, mutate, swap\}$ are the probability for deleting, adding, mutating, or reordering instructions within a program. m_w is the dimensionality of a program's vector and matrix memory. p_{mem} is the probability of changing m_w. p_{atomic} is the probability that a modified action-pointer will be atomic (terminal).

Team Population			
Parameter	Value	Parameter	Value
Agent (root team) population size	1000	p_{md}	0.5
n_{elite}	500	p_{ma}	0.4
Initial team size	10	p_{mn}, p_{ms}	0.1
Max team size	∞	p_{mm}	0.2
		p_x	0.5

Program Population			
Parameter	Value	Parameter	Value
Number of elements in `s`	8	*Initial* m_w	2
Number of elements in `m`	8	*Min* m_w	2
Number of elements in `v`	8	*Max* m_w	8
Initial program size	10	*Max program size*	∞
p_{delete}	0.5	p_{add}	0.4
p_{mutate}	1.0	p_{swap}	0.2
p_{atomic}	0.99	p_{mem}	0.2

Table 1.3 Set of operations available to programs. s, $\mathbf{v}$, and M denote a scalar, vector, and matrix respectively. Early-alphabet letters (a, b, etc) denote memory addresses. Mid-alphabet letters (e.g i, j, etc) denote indexes. In this work, the *scalar* experimental case is limited to OP0 - OP12, while all operations are available in the *scalar, vector, matrix* case. Operations are inspired by AutoML-Zero [25].

Op ID	Code Example	Input Args Addresses / types	Output Args Address	Description (see caption) / type
OP0	`s2=s3+s0`	a,b / scalars	c / scalar	$s_c = s_a + s_b$
OP1	`s4=s0-s1`	a,b / scalars	c / scalar	$s_c = s_a - s_b$
OP2	`s8=s5*s5`	a,b / scalars	c / scalar	$s_c = s_a\, s_b$
OP3	`s7=s5/s2`	a,b / scalars	c / scalar	$s_c = s_a / s_b$
OP4	`s8=abs(s0)`	a / scalar	b / scalar	$s_b = \lvert s_a \rvert$
OP5	`s4=1/s8`	a / scalar	b / scalar	$s_b = 1/s_a$
OP6	`s1=cos(s4)`	a / scalar	b / scalar	$s_b = \cos(s_a)$
OP7	`s1=exp(s2)`	a / scalar	b / scalar	$s_b = e^{s_a}$
OP8	`s0=log(s3)`	a / scalar	b / scalar	$s_b = \log s_a$
OP9	`s3=heaviside(s0)`	a / scalar	b / scalar	$s_b = \mathbb{1}_{\mathbb{R}^+}(s_a)$
OP10	`if(s3<s7):s2=-s2`	a,b / scalar	c / scalar	IF(s_a ¡ s_b) THEN s_c = -s_c
OP11	`s1=minimum(s2,s3)`	a,b / scalars	c / scalar	$s_c = \min(s_a, s_b)$
OP12	`s8=maximum(s3,s0)`	a,b / scalars	c / scalar	$s_c = \max(s_a, s_b)$
OP13	`v2=heaviside(v2)`	a / vector	b / vector	$\mathbf{v}_b^{(i)} = \mathbb{1}_{\mathbb{R}^+}(\mathbf{v}_a^{(i)}) \forall i$
OP14	`m7=heaviside(m3)`	a / matrix	b / matrix	$M_b^{(i,j)} = \mathbb{1}_{\mathbb{R}^+}(M_a^{(i,j)}) \forall i,j$
OP15	`v1=s7*v1`	a,b / sc,vec	c / vector	$\mathbf{v}_c = s_a\, \mathbf{v}_b$
OP16	`v1=bcast(s3)`	a / scalar	b / vector	$\mathbf{v}_b^{(i)} = s_a \;\; \forall i$
OP17	`v5=1/v7`	a / vector	b / vector	$\mathbf{v}_b^{(i)} = 1/\mathbf{v}_a^{(i)} \;\; \forall i$
OP18	`s0=norm(v3)`	a / scalar	b / vector	$s_b = \lvert \mathbf{v}_a \rvert$
OP19	`v3=abs(v3)`	a / vector	b / vector	$\mathbf{v}_b^{(i)} = \lvert \mathbf{v}_a^{(i)} \rvert \;\; \forall i$
OP20	`v5=v0+v9`	a,b / vectors	c / vector	$\mathbf{v}_c = \mathbf{v}_a + \mathbf{v}_b$
OP21	`v1=v0-v9`	a,b / vectors	c / vector	$\mathbf{v}_c = \mathbf{v}_a - \mathbf{v}_b$
OP21	`v8=v1*v9`	a,b / vectors	c / vector	$\mathbf{v}_c^{(i)} = \mathbf{v}_a^{(i)}\, \mathbf{v}_b^{(i)} \forall i$
OP22	`v9=v8/v2`	a,b / vectors	c / vector	$\mathbf{v}_c^{(i)} = \mathbf{v}_a^{(i)} / \mathbf{v}_b^{(i)} \forall i$
OP23	`s6=dot(v1,v5)`	a,b / vectors	c / scalar	$s_c = \mathbf{v}_a^T\, \mathbf{v}_b$
OP24	`v4=minimum(v3,v9)`	a,b / vectors	c / vector	$\mathbf{v}_c^{(i)} = \min(\mathbf{v}_a^{(i)}, \mathbf{v}_b^{(i)}) \forall i$
OP25	`v7=maximum(v3,v6)`	a,b / vectors	c / vector	$\mathbf{v}_c^{(i)} = \max(\mathbf{v}_a^{(i)}, \mathbf{v}_b^{(i)}) \forall i$
OP26	`s2=mean(v2)`	a / vector	b / scalar	$s_b = \text{mean}(\mathbf{v}_a)$
OP27	`s3=std(v3)`	a / vector	b / scalar	$s_b = \text{stdev}(\mathbf{v}_a)$

[Table continues on the next page.]

Table 1.3 Ops vocabulary (continued)

Op ID	Code Example	Input Args Addresses / types	Output Args Address / type	Description (see caption)
OP28	`m1=outer(v6,v5)`	*a*,*b* / vectors	*c* / matrix	$M_c = \mathbf{v}_a \mathbf{v}_b^T$
OP29	`m1=s4*m2`	*a*,*b* / sc/mat	*c* / matrix	$M_c = s_a M_b$
OP30	`m3=1/m0`	*a* / matrix	*b* / matrix	$M_b^{(i,j)} = 1/M_a^{(i,j)} \forall i,j$
OP31	`v6=dot(m1,v0)`	*a*,*b* / mat/vec	*c* / vector	$\mathbf{v}_c = M_a \mathbf{v}_b$
OP32	`m2=bcast(v0,axis=0)`	*a* / vector	*b* / matrix	$M_b^{(i,j)} = \mathbf{v}_a^{(i)} \forall i,j$
OP33	`m2=bcast(v0,axis=1)`	*a* / vector	*b* / matrix	$M_b^{(j,i)} = \mathbf{v}_a^{(i)} \forall i,j$
OP34	`s2=norm(m1)`	*a* / matrix	*b* / scalar	$s_b = \|\|M_a\|\|$
OP35	`v4=norm(m7,axis=0)`	*a* / matrix	*b* / vector	$\mathbf{v}_b^{(i)} = \|M_a^{(i,\cdot)}\| \forall i$
OP36	`v4=norm(m7,axis=1)`	*a* / matrix	*b* / vector	$\mathbf{v}_b^{(j)} = \|M_a^{(\cdot,j)}\| \forall j$
OP37	`m9=transpose(m3)`	*a* / matrix	*b* / matrix	$M_b = \|M_a^T\|$
OP38	`m1=abs(m8)`	*a* / matrix	- *b* / matrix	$M_b^{(i,j)} = \|M_a^{(i,j)}\| \forall i,j$
OP39	`m2=m2+m0`	*a*,*b* / matrixes	*c* / matrix	$M_c = M_a + M_b$
OP40	`m2=m3-m1`	*a*,*b* / matrixes	*c* / matrix	$M_c = M_a - M_b$
OP41	`m3=m2*m3`	*a*,*b* / matrixes	*c* / matrix	$M_c^{(i,j)} = M_a^{(i,j)} M_b^{(i,j)} \forall i,j$
OP42	`m4=m2/m4`	*a*,*b* / matrixes	*c* / matrix	$M_c^{(i,j)} = M_a^{(i,j)} / M_b^{(i,j)} \forall i,j$
OP43	`m5=matmul(m5,m7)`	*a*,*b* / matrixes	*c* / matrix	$M_c = M_a M_b$
OP44	`m2=minimum(m2,m1)`	*a*,*b* / matrixes	*c* / matrix	$M_c^{(i,j)} = \min(M_a^{(i,j)}, M_b^{(i,j)}) \forall i,j$
OP45	`m7=maximum(m1,m0)`	*a*,*b* / matrixes	*c* / matrix	$M_c^{(i,j)} = \max(M_a^{(i,j)}, M_b^{(i,j)}) \forall i,j$
OP46	`s2=mean(m8)`	*a* / matrix	*b* / scalar	$s_b = \text{mean}(M_a)$
OP47	`v1=mean(m2,axis=0)`	*a* / matrix	*b* / vector	$\mathbf{v}_b^{(i)} = \text{mean}(M_a^{(i,\cdot)}) \forall i$
OP48	`v3=std(m2,axis=0)`	*a* / matrix	*b* / vector	$\mathbf{v}_b^{(i)} = \text{stdev}(M_a^{(i,\cdot)}) \forall i$
OP49	`s4=std(m0)`	*a* / matrix	*b* / scalar	$s_b = \text{stdev}(M_a)$

References

1. Badcock, P.B., Friston, K.J., Ramstead, M.J.D., Ploeger, A., Hohwy, J.: The hierarchically mechanistic mind: An evolutionary systems theory of the human brain, cognition, and behavior. Cognitive, Affective, & Behavioral Neuroscience **19**(6), 1319–1351 (2019)
2. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer (2007)
3. Brave, S.: The evolution of memory and mental models using genetic programming. In: Proceedings of the 1st Annual Conference on Genetic Programming, pp. 261–266. MIT Press (1996)
4. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. arXiv **1606.01540** (2016)
5. Chillet, A., Boyer, B., Gerzaguet, R., Desnos, K., Gautier, M.: Tangled Program Graph for Radio-Frequency Fingerprint Identification. In: 2023 Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2023) (2023)
6. Cui, Y., Ahmad, S., Hawkins, J.: Continuous Online Sequence Learning with an Unsupervised Neural Network Model. Neural Computation **28**(11), 2474–2504 (2016)
7. Desnos, K., Bourgoin, T., Dardaillon, M., Sourbier, N., Gesny, O., Pelcat, M.: Ultra-fast machine learning inference through C code generation for tangled program graphs. In: 2022 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6 (2022)
8. Desnos, K., Sourbier, N., Raumer, P.Y., Gesny, O., Pelcat, M.: Gegelati: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs. In: Workshop on Design and Architectures for Signal and Image Processing (14th Edition), DASIP '21, p. 35–43. ACM, New York, NY, USA (2021)
9. Gligorovski, N., Zhong, J.: Lgp-vec: A vectorial linear genetic programming for symbolic regression. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion, p. 579–582. Association for Computing Machinery, New York, NY, USA (2023)
10. Hawkins, J., Ahmad, S., Cui, Y.: A Theory of How Columns in the Neocortex Enable Learning the Structure of the World. Frontiers in Neural Circuits **11**, 81 (2017)
11. Hübner, U., Abraham, N.B., Weiss, C.O.: Dimensions and entropies of chaotic intensity pulsations in a single-mode far-infrared nh_3 laser. Phys. Rev. A **40**, 6354–6365 (1989)
12. Kelly, S.: Scaling genetic programming to challenging reinforcement tasks through emergent modularity. Ph.D. thesis, Faculty of Computer Science, Dalhousie University (2018)
13. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. Evolutionary Computation **26**(3), 347–380 (2018)
14. Kelly, S., Newsted, J., Banzhaf, W., Gondro, C.: A modular memory framework for time series prediction. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20, p. 949–957. ACM, New York, NY, USA (2020)
15. Kelly, S., Park, D.S., Song, X., McIntire, M., Nashikkar, P., Guha, R., Banzhaf, W., Deb, K., Boddeti, V.N., Tan, J., Real, E.: Discovering adaptable symbolic algorithms from scratch. In: International Conference on Intelligent Robots and Systems (IROS) (2023)
16. Kelly, S., Schossau, J.: Evolutionary Computation and the Reinforcement Learning Problem, pp. 79–118. Springer Nature Singapore, Singapore (2024)
17. Kelly, S., Smith, R.J., Heywood, M.I., Banzhaf, W.: Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. ACM Trans. Evol. Learn. Optim. **1**(3) (2021)
18. Kelly, S., Voegerl, T., Banzhaf, W., Gondro, C.: Evolving hierarchical memory-prediction machines in multi-task reinforcement learning. Genetic Programming and Evolvable Machines **22**(4), 573–605 (2021)
19. Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A.A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., Hadsell, R.: Overcoming catastrophic forgetting in neural networks. Proceedings of the National Academy of Sciences **114**(13), 3521–3526 (2017)

20. Mackey, M.C., Glass, L.: Oscillation and chaos in physiological control systems. Science **197**(4300), 287–289 (1977)
21. Miller, J.F.: Evolving developmental neural networks to solve multiple problems. In: ALIFE 2020: The 2020 Conference on Artificial Life, Artificial Life Conference Proceedings, pp. 473–482. ASME (2020)
22. Mountcastle, V.B.: The columnar organization of the neocortex. Brain **120**, 701–722 (1997)
23. Najarro, E., Risi, S.: Meta-learning through hebbian plasticity in random networks. In: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 20,719–20,731. Curran Associates, Inc. (2020)
24. Praczyk, T., Szymkowiak, M.: Linear matrix genetic programming as a tool for data-driven black-box control-oriented modeling in conditions of limited access to training data. Scientific Reports **14**(1), 12,666 (2024)
25. Real, E., Liang, C., So, D.R., Le, Q.V.: AutoML-zero: Evolving machine learning algorithms from scratch. arXiv preprint arXiv:2003.03384 (2020)
26. Smith, R.J., Amaral, R., Heywood, M.I.: Evolving simple solutions to the cifar-10 benchmark using tangled program graphs. In: 2021 IEEE Congress on Evolutionary Computation (CEC), pp. 2061–2068 (2021)
27. Smith, R.J., Heywood, M.I.: Evolving Dota 2 Shadow Fiend Bots Using Genetic Programming with External Memory. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19, pp. 179–187. ACM, New York, NY, USA (2019)
28. Smith, R.J., Heywood, M.I.: Interpreting tangled program graphs under partially observable dota 2 invoker tasks. IEEE Transactions on Artificial Intelligence **5**(4), 1511–1524 (2024)
29. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. J. Artif. Int. Res. **21**(1), 63–100 (2004)
30. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA (2018)
31. Teller, A.: The Evolution of Mental Models. In: Advances in Genetic Programming, Volume 1. The MIT Press (1994)
32. Trappenberg, T.P.: Fundamentals of Computational Neuroscience. Oxford University Press UK (2002)
33. Turner, A.J., Miller, J.F.: Recurrent Cartesian Genetic Programming of Artificial Neural Networks. Genetic Programming and Evolvable Machines **18**(2), 185–212 (2017)
34. Vithayathil Varghese, N., Mahmoud, Q.H.: A survey of multi-task deep reinforcement learning. Electronics **9**(9) (2020)
35. Wineberg, M., Oppacher, F.: The benefits of computing with introns. In: Proceedings of the 1st Annual Conference on Genetic Programming, p. 410–415. MIT Press, Cambridge, MA, USA (1996)