Evolving Simple Solutions to the CIFAR-10 Benchmark using Tangled Program Graphs

Robert J. Smith
Faculty of Computer Science
Dalhousie University
Halifax, NS. Canada
robert.smith@dal.ca

Ryan Amaral
Faculty of Computer Science
Dalhousie University
Halifax, NS. Canada
ryan.amaral@dal.ca

Malcolm I. Heywood
Faculty of Computer Science
Dalhousie University
Halifax, NS. Canada
mheywood@cs.dal.ca

2021 IEEE Congress on Evolutionary Computation (CEC) | 978-1-7281-8393-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/CEC45853.2021.9504998

***Abstract*—The goal of the CIFAR-10 benchmark is recast from the perspective of discovering light-weight as well as accurate solutions. Specifically, the image data, on which CIFAR-10 is based, requires multiple practical issues to be addressed that are not often considered collectively when applying genetic programming to classification problems. Issues of particular interest include cardinality, multi-class classification and diversity maintenance. We demonstrate that diversity maintenance and cardinality can be approached simultaneously by adopting a data subset to compose pools of exemplars for lexicase selection. The issues of multi-class classification and solution simplicity are addressed by adopting the tangled program graph (TPG) approach to emergent modularity. In addition, the mutation operator is modified to ensure that class labels do not 'die out' during evolution. The resulting benchmarking study demonstrates solutions that are significantly more accurate than AutoML while providing comparable accuracies with solutions from unsupervised feature discovery, i.e. 70% accuracy. However, unlike the latter TPG solutions are several orders of magnitude simpler.**

***Index Terms*—CIFAR-10, Tangled program graphs, Modularity, Lexicase Selection, *k*-armed Bandit**

I. Introduction

Before deep learning frameworks became the norm for classification tasks, it was increasingly recognized that factors associated with unsupervised 'feature construction' – before the application of a classifier – had the most impact on classification performance (e.g. [1], [2]). Thus, factors such as receptive field size, the number of hidden nodes (features), stride between features, and the effect of feature whitening were investigated. This resulted in solutions to image recognition benchmarks such as CIFAR-10 that were based on thousands as opposed to millions of parameters. In this work, it is solutions at this level of complexity that we are interested in explicitly evolving using genetic programming. The point being that at this level of complexity, we are still able to do so without recourse to specialized hardware platforms, e.g. graphics processing units, hereafter a simple or light-weight solution. Our underlying objective is to simulate research into classification frameworks that might ultimately result in complimentary approaches to that of deep learning.

Research supported by NSERC Discovery Grant 45009

The starting hypothesis of this work is that in order to produce light-weight solutions to high-dimensional image classification tasks, we need to embrace categorical forms of modularity (reviewed in §II). This means that although a candidate solution might be composed from millions of instructions, only thousands of instructions are involved in making each prediction. In short, the path of execution through a candidate solution is contextually sensitive on the relationship between (code) modules comprising a solution.

For the purposes of this motivating study we concentrate on the CIFAR-10 dataset [3]. Making process in this task is a challenge because multiple factors have to be simultaneously addressed: cardinality (50,000 records in the training partition), multi-class classification (10 classes) and diversity maintenance. An underlying constraint that we assume is that a single champion classifier has to appear at the end of a trial capable of labelling the CIFAR-10 dataset without resorting to specialist hardware to accelerate training/ post training operation.

The remainder of the paper is developed as follows. Section II reviews modularity as used in this research. Modularity will be assumed to facilitate the incremental construction of programs, reuse and decomposition of the task. Section III summarizes the tangled program graph (TPG) framework assumed for supporting emergent modularity, where TPG has previously demonstrated an ability for discovering solutions to high-dimensional reinforcement problems [4]–[6] and a capacity for operation under modest computing platforms [7]. Section IV summarizes the set of approaches adopted for simultaneously addressing cardinality and diversity maintenance through the design of appropriate interfaces to the training partition. Section V presents results that indicate that TPG can achieve accuracies competitive with those from AutoML, with the conclusion and future work appearing in Section VI.

II. Categorical Modularity

Modularity appears in multiple guises in genetic programming (GP), with the most well known being that of subroutines (e.g., Automatically Defined Functions [8]). In this context modularity is synonymous with the decomposition of a task into distinct *functions* [9], i.e. as used in software engineering.

In contrast *categorical* modularity encourages multiple modules to exist with similar yet distinct properties [9]. In both cases, it is necessary to generate a sufficiently diverse set of modules while simultaneously learning how to deploy them. Scaling to more difficult tasks potentially appears through module reuse, but only if useful modules can be discovered. Similar issues appear in various forms of ensemble learning using GP [10], [11] and coevolution [9], [12].

In the context of this research, we are particularly interested in the case of categorical modularity in multi-class classification. One simple approach to enforcing modularity would be to assume a single module per class, as has been the case in approaches based on ensemble learning [11] or multi-trees [13]. However, such approaches often assume prior knowledge regarding ensemble size. Conversely, this research adopts the metaphor of teams of learners (e.g. [14], [15]) in which a learner is a program that represents action and context (for its action) independently. As a consequence, team composition is an emergent phenomena with different teams adopting different numbers of programs and distributions of terminal actions (class labels).

Switching between modules has also been articulated through the guise of diversity maintenance in (competitive) coevolution [9], [12]. In short, if an appropriate diversity maintenance scheme can be identified, then multiple specialists (aka modules) are simultaneously maintained in the population. Hypothetically, an appropriate team can then be composed by learning a 'gating function' that learns to switch between the relevant specialists under different stimuli. Neither [9], [12] were able to develop an appropriate gating function, but they were able to recommend specific types of diversity maintenance w.r.t. the iterated prisoner's dilemma task.

Recently, schemes have appeared for developing emergent modularity through 'teams-as-actions' using the Tangled Program Graph (TPG) framework [4], [5]. This enables a terminal action to instead reference another team. This is useful when a learner's context has insufficient resolution to distinguish between multiple actions. Instead, hierarchical relationships are developed that enable teams of programs with more specialist abilities to be referenced. This has the advantage of not disturbing the pattern of contexts currently developed at the calling team. The end result is a solution described by 'graphs of teams of programs' discovered in an entirely emergent manner [4], [5].

Another theme of significance when addressing practical classification tasks is that of (training) data cardinality. One approach might be to assume specialist computing support, such as cloud or parallel computing infrastructure [16], [17]. The end result again being multiple models that are combined using some form of voting. The approach adopted in this work is to assume that fitness evaluation is performed against a data subset (DS) of much lower cardinality than the training partition. Biases can then be introduced to shape how the records appearing in the DS are selected/retained/removed [14], [15], [18], i.e. membership of the DS changes to reflect developments in the performance of the classifiers. The specific mechanisms assumed for doing so are detailed in §IV.

III. Tangled Program Graphs

Tangled Program Graphs (TPG) represents a GP framework in which emergent modularity incrementally composes programs into teams of programs into graphs of teams of programs [4], [5], [19]. To date, the framework has been demonstrated on visual reinforcement learning tasks[1] as encountered in the Arcade Learning Environment [4], [5] or ViZDoom first person shooter [6], [20]. As such, it might be anticipated that TPG also represents a suitable starting point for multi-class classification tasks described by visual input data. Sections III-A through III-C summarize the TPG framework, interpreting operation from the perspective of multi-class classification.

A. Learners

A learner, $\mathcal{L}(i)$, defines an individual, *i*, in terms of a program, *prog*, and terminal action, a where $a \in \mathcal{C}$ is the set of classes in a multi-class classification problem, or $\mathcal{L}(i) = \langle prog_i, a_i \rangle$. A program only produces a single output, whether that be the root node of tree structured GP or register $R[0]$ in the case of linear GP. Actions are merely a scalar corresponding to a terminal action (these will later evolve to also encompass pointers to other teams §III-B). The purpose of a program is to define context for the corresponding action. The same program can appear in different learners if it is partnered with a different action. A learner on its own does not define anything useful. Learners only appear in the Learner population, $\mathcal{L}$.

B. Teams

An independent team population, $\mathcal{T}$, conducts a search for good combinations of learners to appear in teams using a variable length representation. The following conditions are enforced: 1) each team, $tm(j)$ must consist of a unique combination of learners; 2) the same learner, $\mathcal{L}(i)$, may appear in multiple teams, subject to condition 1; 3) there cannot be less than two learners in the same team; 4) there must be at least two different actions represented by the complement of learners within the same team.

In order to establish the output of team *tm*(*j*), all programs from learners within this team are evaluated on training record, p_k, or $\forall i \in tm(j) : y_i = prog_i(p_k)$. The program with maximum output on p_k is identified by $i^* = \arg\max_i(y_i)$. Such a program wins the right to suggest its corresponding terminal action, a_i^*, for comparison against the known class label, o_k. Thus, the outcome of the interaction between training record, p_k, and team *tm*(*j*) is,

$$G(p_k, tm(j)) = \begin{cases} 1, & \text{IF } o_k == a_i^* \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Variation operators assume that the worst *Gap* teams have been removed, or a breeder (§IV-A). Any learners that are not

[1]State defined in terms of pixel values from video input (e.g. 33,600 pixels in [19] and 76,800 pixels in [6], [20]).

associated with a team, are also deleted. The remaining pool (of teams) represent potential parents, of which *Gap* are selected and cloned. Only the cloned teams are modified through crossover and mutation. Crossover selects parents pairwise from the pool of surviving teams with uniform probability. The learners common to both appear in the offspring. Learners unique to each parent are selected to appear in an offspring with probability P_{cpy}. Let the result of this process be the set of Gap offspring teams, $\mathcal{L}'$.

Mutation takes the form of stochastically adding (P_a) or deleting (P_d) learners from the offspring pool, $\mathcal{L}'$ (subject to the above constraints). Variation is performed relative to learners indexed by teams from $\mathcal{L}'$ with probability P_m. Again, should a learner be selected for variation, it is first cloned. This means that only the offspring team inherits the modified learner, $\mathcal{L}'(i)$, not any of the $\mathcal{T}-Gap$ grandfathered teams that happened to use the same learner. Learner variation operators include: instruction delete (P_{del}), add (P_{add}), swap (P_{swp}) and choose a new terminal action (P_{mn}).

C. Graphs

Section III-A defined a learner as the smallest 'module' whereas Section III-B provided a mechanism for organizing learners into teams without prior parameterization for how many learners should appear in a team.[2] Different teams might excel at labelling different subsets of exemplars. Typically, it is assumed that cross-over will provide a sufficient mechanism for recombining the properties from different teams. However, an underlying assumption with such a process is that the learners are always able to identify unique conditions under which to out-bid other learners. TPG avoids this assumption by enabling a learner to instead reference a different team, thus devolving control to the referenced team under record p_k.

The key to this process is to provide two *types* of learner action mutation. At initialization all learners are initialized with terminal actions (corresponding to the available class labels). Thereafter, an action mutation consists of the following sequence of tests:

1) IF $rand > P_{mn}$ THEN no mutation
2) ELSE
    a) IF $rand > P_{action}$ THEN $a_i \leftarrow$ Choose($\mathcal{C}$)
    b) ELSE $a_i \leftarrow$ Choose($\mathcal{T}$)

where Step 1 represents the decision to modify an action. When true either the set of terminal actions, $\mathcal{C}$, is chosen (Step 2a) or a pointer to another team, $\mathcal{T}$, is established (Step 2b).

Two types of team are now recognized. Those that receive at least one reference from another team and those that do not; the latter define the set of 'root teams' $\mathcal{T}_{root}$. At initialization $\mathcal{T}_{root} = \mathcal{T}$. Evaluation may only commence from a root team. Team evaluation is unchanged Section III-B. Should the winning learner's action be a terminal action, then comparison against the class label is performed (Equ. (1)). Otherwise, the action is a pointer to another team and the process of determining the winning learner repeats at the identified team.

The set of eligible parents is also limited to the set of root teams, or $tm \in \mathcal{T}_{root}$. Thus, variation operators (§III-B) are *only* applied to root teams with ratio of root to non-root teams floating. Moreover, the non-root teams essentially behave as if they have been archived, unless at some point the variation operators remove all incoming references.

Naturally, it is also possible for loops to appear in the path of evaluation, i.e. the halting problem. TPG avoids this issue, by marking teams visited during the evaluation of a root team. Should a learner identify a previously visited team, then the learner with runner up bid is (recursively) selected. By enforcing the constraint that all teams have to have a minimum of one terminal action, TPG guarantees that loops cannot result. Further details of the TPG algorithm appear in tutorial form in [5], [19].

IV. Selection, Cardinality, and Diversity Maintenance

In this section, several forms of diversity maintenance are introduced relative to the concept of a data subset. The cardinality of the data subset is lower than that of the original training partition, hence both diversity maintenance and cardinality are addressed simultaneously. In the following, an 'individual' is synonymous with the TPG root team under evaluation (§III-C).

Algorithm 1 Balanced Random Subset Selection (BRSS). The loop of Step 1a retains $(1 - DS_{gap})\%$ of *DS* content between generations with uniform probability (Step 1(a)i). Each class then has DS_{gap} new content introduced (Step 1(a)ii). These class specific samples are then concatenated to form the data subset (Step 1b) used for fitness evaluation at generation *t*, Step 1c. Individuals from the team population can then be ranked (Step 1d) and the worst performing $Gap\%$ of individuals dropped (§III-B). Parents are sampled with uniform probability from the surviving individuals and variation operators applied to replace the $Gap\%$ of individuals deleted during Step 1e. The new population, $P(t+1)$, is defined by the survivors from fitness evaluation, $P(t)$, and the *offspring* from Step 1g.

1) for $(t = 0; t < MaxGen; t = t + 1)$
    a) for $(c = 0; c < |C|; c = c + 1)$
        i) $DS(c,t) = Cpy(DS(c,t), DS_{gap})$
        ii) $DS(c,t) = DS(c,t) \cup Sample(TP(c))$
    b) $DS(t) = DS(0,t) \cup ... \cup DS(C-1,t)$;
    c) EvalFitness($P(t), DS(t)$);
    d) Rank($P(t)$);
    e) Del($P(t), Gap$);
    f) *parents* = Select($P(t), Gap$);
    g) *offspring* = Variation($parents$);
    h) $P(t+1)$ = cat($P(t)$, *offspring*);

[2] Although a minimum of two learners (with different actions) is necessary to avoid defining a degenerate team §III-B.

A. Balanced Random Data Subset

Cardinality is addressed by sampling from the original training partition (*TP*) to construct a data subset (*DS*) where $|DS| << |TP|$ using Algorithm 1. Given that CIFAR-10 represents a balanced dataset consisting of 10 classes, the *DS* is formed by sampling each class equally, i.e. we assume that all classes are equally difficult to classify. This represents our baseline approach for addressing cardinality; hereafter Balanced Random (data) Subset Selection (BRSS). Algorithm 1 details the corresponding training cycle under a 'breeder' model of selection.

The fitness function assumed during Step 1c is merely the classification count:

$$f(j,t) = \frac{1}{|DS|} \sum_{\forall p_k \in DS(t)} G(p_k, tm(j)) \tag{2}$$

where $DS(t)$ is the data subset of training records at generation *t*; $G(p_k, tm(j))$ represents the binary outcome of (root) team '*j*' on record '*k*' as defined by Equ. (1). Note that as the content of the data subset at each generation, $DS(t)$, is always balanced, then fitness reflects performance over all classes. Individuals with larger $f(j,t)$ are preferred.

B. Fitness Sharing

Fitness sharing (FS) builds directly on BRSS (§IV-A) by introducing a weighting into fitness such that training exemplars that many other (few other) classifiers correctly label are worth less (more) [21]. Thus, in this work, the only difference between the algorithm for BRSS and fitness sharing is the definition for fitness. Equation (2) in Step 1c of Algorithm 1 is therefore replaced by:

$$f(j,t) = \sum_{\forall p_k \in DS(t)} \frac{G(p_k, tm(j))}{\sum_{\forall tm(i) \in \mathcal{T}_{root}} G(p_k, tm(i))} \tag{3}$$

where $G(p_k, tm(j))$ assumes the same definition as in Equ. (1). Naturally, individuals with larger $f(j,t)$ are still preferred during fitness ranking, but this time fitness has the potential for maintaining diversity in the population by holding on to 'specialist' individuals for longer than in BRSS.

C. The k-Armed Bandit

The question of which class to compose the data subset from can also be posed in the form of a *k*-Armed Bandit, one for each of the $\mathcal{C}$ classes. Each 'class bandit' models the return associated with choosing a record from class *c* from the data subset as an estimate of future rewards $Q(c)$. At generation $t = 0$ each 'class bandit' has the same return, $Q(c) = 0$. Thereafter, we use the average error on class '*c*' w.r.t. the champion team to define the reward, r_c received by class bandit '*c*' at generation, *t*. Thus, more difficult classes see a higher return, resulting in a proportionally higher number of selections of training records from that class. With this in mind, an ε-greedy method is assumed for class selection, as per Algorithm 2.

Algorithm 2 Bandit selection of training record class. Replaces Steps 1a through 1c in BRSS. Step 1 copies $(1 - DS_{gap})\%$ of records currently represented in *DS* into the next data subset (Step 2). *TS* is then incrementally sampled to provide DS_{gap} new training records, Step 3, under the direction of the Bandit's selection of classes (Step 3a). Step 4 returns the average error on class '*c*' records present in DS at generation '*t*' relative to the champion team. $Q(c)$ denotes the estimate of cumulative rewards (error) on class '*c*'. $Q(c) = 0$ at generation zero. $\alpha(c)$ is the class-wise step size parameter, initialized to $\frac{1}{n(c)=1}$ at generation 0. Thereafter $\alpha(c) = \frac{1}{n(c)+=1}$ with each action use. See also §2.4 of [22].

1) $\forall c \in C : DS(c,t) = Cpy(DS(c,t), DS_{gap})$
2) $DS(t) = DS(0,t) \cup ... \cup DS(C-1,t)$;
3) for $(k = 0; k < |DS(t)|; k = k + 1)$
   a) $c = \begin{cases} \arg\max_i Q(i), & \text{with probability } 1-\varepsilon \\ rand(0, C-1), & \text{with probability } \varepsilon \end{cases}$
   b) $DS(t) = DS(t) \cup$ Select$(DS(c))$;
4) EvalFitness$(P(t), DS(t))$;
5) for $(c = 0; c < |C|; c = c + 1)$
   a) $r_c =$ ClassError$(P(t), DS(c))$
   b) $Q(c) = Q(c) + \alpha(c)[r_c - Q(c)]$

Algorithm 3 Initializing Lexicase selection for classification. The objective is to select the $(P - Gap)\%$ individuals that *survive* from generation t to $t+1$. `pool` is the set of eligible parents at generation t, i.e. $\mathcal{T}_{root}(t)$ in the case of TPG. `instances` are the training cases used for fitness evaluation. $TP(c)$ represents the class specific ($c \in \{0, ..., C-1\}$) training partition selected at epoch *t*. Note that the order of `instances` is shuffled at each call.

1) *parents* = $\emptyset$;
2) $c = rand(0, C-1)$;
3) for $(i = 0; i < P - Gap; i = i + 1)$
   a) `pool` = $P(t)$;
   b) `instances` = *shuffled*$(TP(c), t)$;
   c) *parents* = *parents* $\cup$ Lexicase(`pool`, `instances`);

D. Lexicase Selection

Lexicase selection represents a model of selection in which individuals are 'not selected' as soon as they fail to provide the correct outcome [23], [24]. This means that fitness evaluation need not be performed across the entire training partition. Moreover, in comparison to other diversity mechanisms, Lexicase selection has demonstrated a better ability to maintain diverse populations of solutions [23]. In order to apply Lexicase selection to multi-classification tasks (Algorithm 3), we select a class from the training partition with uniform probability (Step 2) and initialize the class `instances` using a random shuffle (Step 3b). The same class is used for selection for τ generations. Algorithm 4 details the call to 'Lexicase' at Step 3c.

Algorithm 4 Lexicase selection function (called by Algorithm 3). Step 2a selects a training pattern from `instances`. Note that the order of training patterns was randomized in Algorithm 3. Step 2b identifies all individuals remaining in the 'pool'. Step 2(b)i represents a misclassification, so the individual is dropped from the pool of eligible individuals (Step 2(b)ii). The loop exits when either `pool` $= \emptyset$ or all individuals are tested against p_k. Step 2c tests for a single remaining individual in pool, which is then returned as a parent (Step 2d). Step 2e removes p_k from the patterns used at this round of selection. If there are no further training instances (Step 2f), then an individual is randomly selected from the individuals remaining in the pool (Step 2g).

1) $k = 0$
2) while (!**return**)
   a) Choose $p_k =$ `instances`(i)
   b) for all ($gp \in$ `pool`)
      i) IF $gp(p_k)! = o_k$
      ii) THEN `delete`(*gp*, `pool`)
   c) IF size_of(`pool`) == 1
   d) THEN **return** select(`pool`)
   e) delete(p_k, `instances`)
   f) IF size_of(`instances`) == 0
   g) THEN **return** select(rand(`pool`))
   h) $k = k + 1$

Relative to Algorithm 1 this means that Steps 1a through to 1e are replaced by Algorithm 3, i.e. Step 3c (Algorithm 3) returns the *surviving* programs at generation '*t*' from which Gap reproducing parents and offspring are defined.

V. Empirical Evaluation

A. Parameterization

Table I represents the common parameterization of TPG assumed throughout the empirical evaluation. No claims are made regarding its optimality, indeed the instruction set is very simplistic. Additional parameters specific to each of the four selection–diversity mechanisms from Section IV are detailed in Table II. The choice for the data subset size and 'gap' reflects a balance between a desire to minimize the cost of fitness evaluation while increasingly exposing surviving TPG individuals to different exemplars as the number of generations increases. The instruction set for learner programs is limited to: $\{+, -, \div 2, \times 2,$ `cond(a, b)` $\}$ where add and subtract take two arguments, divide and multiply take one and the last instruction is a two argument conditional of the form: `a > b ? a = a : a = -a`.[3]

All benchmarking will be performed using the CIFAR-10 dataset[4], which represents a 10 class image classification task. There is a standard partition into training and test (Table III) that will be assumed throughout. Given the size of the dataset, three independent runs are performed for each of the four selection / diversity heuristics (§IV) using a generation limit of $t_{max} = 10,000$; hereafter **TPG(short)**. Under this parameterization one run takes 1 to 1.5 weeks to complete (single thread) and potentially iterates through the entire training partition 10 times over the course of evolution. One further run is then performed under the preferred configuration with a generation $t_{max} = 50,000$, i.e. a run time in the order of 5 to 6 weeks (single thread); hereafter **TPG(long)**. The R-G-B three colour formate of the dataset is converted into a single 24-bit integer by first representing each colour as an 8-bit integer and then concatenating into a single 24-bit integer using 8-bit bit-shifts [6].

[3]Definitions assumed for division and multiplication were assumed to reduce the likelihood of extreme values being returned [25].

[4]https://www.cs.toronto.edu/~kriz/cifar.html

TABLE I
TPG Parameterization common to all experiments. For the most part this follows an earlier work deploying TPG in ViZDoom reinforcement learning tasks [6]. *Gap* is the % of Root teams replaced. ω is the number of Learners per Team at initialization

Team Population		Learner Population	
Parameter	Value	Parameter	Value
Pop. Size (P)	360	Max. Instructions	128
Gap	50%	Prob. Delete Instr. (P_{del})	0.5
ω	9	Prob. Add Instr. (P_{add})	0.5
P_d	0.7	Prob. Mutate Instr. (P_{mut})	1.0
P_a	0.7	Prob. Swap Instr. (P_{swp})	1.0
P_m	0.2	P_{mn}, P_{atomic}	0.2, 0.5

TABLE II
Parameters specific to the selection–diversity mechanisms (§IV)

BRSS, FS, Bandit	
Size of the Data subset (DS)	120
Number of data records replaced per generation (D_{gap})	50%
Lexicase	
Consecutive generations of class selection (τ)	10

B. Results

Section IV defined four configurations of TPG of increasing complexity to address the issues of selection, cardinality and diversity maintenance. In order to rank the training outcomes from the TPG(short) runs under each configuration, the top 10 agents from each run are identified on the training partition. These classifiers are then compared using a (two sample, unequal variance) *t*-test with the case of BRSS as the control. Table IV provides a summary of the resulting *p*-values, where

TABLE III
CIFAR-10 dataset properties. The 10 classes represent airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships and trucks.

Training Partition (per class)	Test Partition (per class)	Num. Classes (C)	Pixels per Image
5,000	1,000	10	$32 \times 32 = 1024$

TABLE IV
BRSS VERSUS EACH ADDITIONAL TPG CONFIGURATION. THE BONFERRONI-DONN POST HOC TEST FOR $\alpha = 0.05$ SETS THE THRESHOLD OF SIGNIFICANCE TO $\frac{\alpha}{k-1} = 0.0167$ WHERE $k = 4$ IS THE NUMBER OF CONFIGURATIONS.

BRSS versus FS	BRSS versus k-AB	BRSS versus Lexicase
1×10^{-4}	2.4×10^{-6}	2.7×10^{-10}

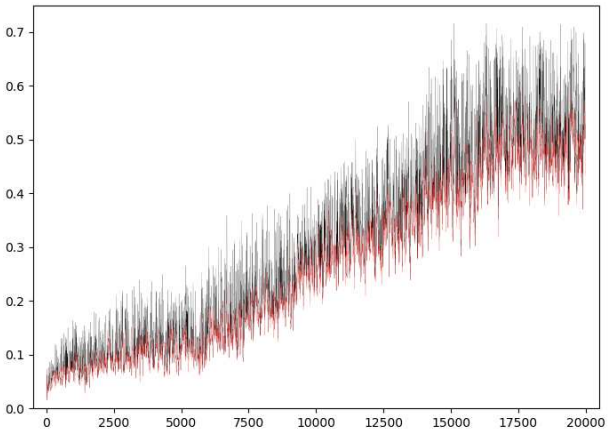

Fig. 1. Fitness of first 20,000 generations of TPG(long). Black indicates performance above average, and red the average.

this reflects the rejection of the null hypothesis that the other three methods are indistinguishable from the control. The p-values indicate that even with the Bonferroni-Donn post hoc correction, all three TPG configurations are better than the control. Relative to overall accuracy the methods were ranked as Lexicase $>$ k-armed Bandit $>$ Fitness sharing $>$ BRSS on both the training and test partition.

Figure 1 illustrates progress of fitness during the first 20,000 generations of TPG(long). A gradual improvement in the performance is observed in the best (black) versus average (red) performing individuals in the population. This contrasts to the sudden increases with flat plateau's often associated with genetic programming. We attribute this to the partial / non-stationary nature of fitness evaluation. That is to say, fitness evaluation is no longer relative to an entire (stationary) training partition, but now performed relative to some subset of training records at any point in time.

Table V summarizes the average *test* performance where it is apparent that the ranking identified under training is retained. Moreover, the TPG(long) run was able to improve on many of the class-wise accuracies. However, it also returned a particularly poor accuracy on classes 3 and 8. In order to gain more insight into this, the distribution of terminal actions across the TPG(long) run champion was queried (Table V). This indicates that the classes with low accuracy might also correspond to classes with a low number of learners (see class C3). This hypothesis is investigated further in Section V-C.

Table VI summarizes the complexity of the single best champion from the TPG(short) runs (columns 1 through 4). It appears that BRSS produced larger teams with less learners whereas the remaining configurations appear to employ smaller teams, but a higher diversity of learners. Note also that only a fraction of the teams are visited before a class label prediction is made, further decreasing the computational cost per decision. This issue will be revisited in §V-C when the matter of learner actions prematurely 'dying out' is addressed.

C. Balanced Action Mutation

Section V-B indicated a preference for lexicase selection but also hinted that learners representing specific classes might 'die out' during the course of a run (e.g. TPG(long) for class C3 in Table V). With this in mind, action mutation (Step 2a) from Section III-C is modified to introduce a bias to choose an action in proportion to $\frac{1}{f_a}$, i.e. the inverse of the frequency with which each action appears within the Root team under modification. The last row of Table V reports the average per class test partition accuracy of three runs performed under this new configuration. In short, the per class classification accuracy is now more balanced, resulting in the highest overall average accuracy and lowest standard deviation. The total spread between worst and best classified class (on average) is lower, implying good generalization across the classes. It is also apparent that the number of learners appearing in the champion solutions for each class is now also 'balanced', although no attempt is made to a priori specify what the number of learners should be per class.

Comparison is now performed against other sources of light-weight solutions for the CIFAR-10 benchmark. As noted in the introduction, prior to deep learning, there was considerable interest in the use of unsupervised methods for feature construction. Such methods provided the basis for insights used to deploy convolutional and max-pooling layers in deep learning frameworks. Once unsupervised learning had designed appropriate features, then a classifier is identified using supervised learning. Particular examples including restricted Boltzmann machines (RBM) and Support Vector Machines (SVM) as well as 'shallow' convolutional neural networks (CNN). More recently, there has also been a development towards 'AutoML' in which a pool of base classifiers are deployed with a Bayesian optimization routine to 'automate' the tuning of learning parameters for each base classifier [26], [27]. An ensemble is then constructed using the trained set of base classifiers.

Table VII summarizes test partition classification accuracies and solution complexities for comparator and TPG (lexicase selection and balanced action mutation). It is apparent that both the unsupervised feature constructors and TPG perform significantly better than AutoML, i.e. the best AutoML performance is no better than the worst performing TPG/ unsupervised feature constructor). It is not possible to say anything specific about the complexity of AutoML solutions as on the one hand there are tens of classification algorithms involved, on the other hand some form of attribute reduction takes place.

TABLE V

AVERAGE *test accuracy* OF TPG WITH EACH SELECTION–DIVERSITY SCHEME. BEST OF COLUMN HIGHLIGHTED IN BOLD. BEST OF TPG(SHORT) HIGHLIGHTED WITH AN UNDERLINE. C*x* IS THE AVERAGE ACCURACY ON CLASS *x*. 'AVG' IS THE AVERAGE ACCURACY ACROSS ALL 10 CLASSES. STDDEV IS THE STANDARD DEVIATION ACROSS ALL 10 CLASSES. #TERM. ACT. REFLECTS THE AVERAGE NUMBER OF TERMINAL ACTIONS OF EACH CLASS IN THE CHAMPIONS FROM TPG(LONG)

TPG(short) : $t_{max} = 10,000$												
Scheme	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Avg	SD
BRSS (§IV-A)	0.528	0.26	0.215	0.458	0.455	0.043	0.183	**0.941**	0.146	0.242	0.347	0.246
FS (§IV-B)	0.317	0.471	0.219	0.342	0.618	0.264	0.301	0.583	0.666	**0.817**	0.46	0.191
k-AB (§IV-C)	0.149	0.659	0.574	**0.881**	0.622	0.11	0.272	0.499	0.517	0.563	0.485	0.228
Lex (§IV-D)	0.618	0.549	0.575	0.129	0.699	0.348	0.347	0.497	0.516	0.782	0.506	0.18
TPG(long) : $t_{max} = 50,000$												
Lex (§IV-D)	0.705	**0.767**	0.001	0.707	0.752	0.572	0.557	0.232	**0.746**	0.75	0.579	0.247
#Term. Act.	21	42	3	16	96	49	29	44	8	74	38.2	29.5
TPG(long) with Balanced action mutation : $t_{max} = 50,000$												
Lex (§IV-D)	**0.711**	0.683	**0.655**	0.58	**0.851**	**0.582**	**0.713**	0.715	0.605	0.729	**0.683**	**0.082**
#Term. Act.	37	39	40	39	43	38	37	39	40	39	39	1.7

TABLE VI

COMPLEXITY OF CHAMPION. VALUES QUOTED FOR LEX(ICASE) SELECTION ARE FOR 'SHORT' (LEX(S)) AND 'LONG' (LEX(L)) RUN CHAMPIONS (LATTER WITH BALANCED ACTION MUTATION) RESPECTIVELY. #INSTRUCTIONS REPRESENTS THE *total* NUMBER OF INSTRUCTIONS ACROSS THE ENTIRE SOLUTION (WHERE FAR LESS ARE EXECUTED PER DECISION).

Scheme	BRSS	FS	*k*-AB	Lex(S)	Lex(L)
#Teams	21	14	14	15	81
#Learners	135	209	201	188	360
#Instructions	4,701	8,777	7,109	6,788	31,823
Avg #Learners/ Team	6.4	14.9	14.4	12.5	4.6
Avg #Instr./ Learner	34.8	42	35.4	36.1	83.0
Avg #Instr./ Team	223.9	626.9	507.8	452.5	371.9

TABLE VII

SUMMARY OF COMPARATOR ALGORITHM PERFORMANCE ON TEST PARTITION. †AUTO-WEKA BEGINS WITH 27 (10) BASE (META-) CLASSIFIERS. ‡AUTO-SKLEARN BEGINS WITH 15 BASE CLASSIFIERS. NOT CLEAR HOW MANY APPEAR IN THE SOLUTION. *K* IS THE NUMBER OF FEATURES WHERE EACH FEATURE HAS *N* PARAMETERS. * DOES NOT INCLUDE CONTRIBUTION FROM CONVOLUTIONAL LAYERS. TPG COMPLEXITY IS DEFINED IN TERMS OF THE AVERAGE NUMBER OF INSTRUCTIONS EXECUTED PER CLASSIFICATION

Algorithm	Avg. Test Accuracy	Solution Complexity
Auto-WEKA [27]	43.05%	†
Auto-SkLearn [27]	48.3%	‡
Auto-WEKA+SMAC [26]	62.39%	†
Gaussian RBM [1]	63.8%	$K = 10,000$
Gaussian RBM+Tuning [1]	64.8%	$K = 10,000$
3-Way Factored RBM [1]	65.3%	$K = 4,096$
Sparse RBM [2]	72.4%	$K = 1,600$
Sparse auto-encoder [2]	73.4%	$K = 1,600$
Fast Shallow CNN [28]	75.86%	4.2×10^6*
TPG (long) + balanced action		
worst	62%	763
median	70.0%	2,317
best	72.0%	1,284

In the case of unsupervised feature construction, both Ranzato et al. [1] and Coates et al. [2] employ zero phase whitening to the entire dataset as a pre-processing step. This has the potential to 'engineer out' sources of variation from the image data (e.g. difference in lighting conditions between images). Given that the TPG results do not employ such a preprocessing step, one avenue for future research would be to investigate the impact of a whitening preprocessing step on TPG. The solution complexities quoted in Table VII for the unsupervised feature construction methods reflect the number of features *per patch*. For example, in [2] the patch size, $N = w \times w \times d$, was also independently optimized with $w = 6$ and d reflecting the number of colour channels. As such the actual computational cost of the unsupervised feature construction approach is actually $106 \times K$, i.e. 2 orders of magnitude higher.

Performance for each of the TPG solutions is quoted in the bottom three rows of Table VII, essentially placing them at the same point as the unsupervised feature constructors. However, the complexity of TPG solutions is 2 to 3 orders of magnitude lower. Specifically, the last column of Table VI summarizes the typical 'static' complexity of the three champion TPG solutions. However, only a faction of the number of teams and learners comprising a TPG solution need be evaluated before a class label is predicted. This is captured by the average instruction counts for TPG in Table VII. It is this property that enables TPG to operate on embedded computing platforms (e.g. Raspberry Pi) [7].

Naturally, deep learning approaches to image classification have reached the point where classification accuracy above 95% is common. Moreover, some attention has also been applied to addressing complexity in deep learning solutions using deterministic search algorithms (e.g. NASNet [29]) or evolutionary computation (e.g. [30]). Although maintaining accuracy, such frameworks still produce solutions composed from millions of parameters, thus reliant on GPU computational support. It is not clear whether solution complexities can be achieved that approach that of TPG while maintaining classification accuracy.

VI. Conclusions

The CIFAR-10 benchmark is revisited from the perspective of evolving light-weight solutions. The goal is to promote the investigation of alternative frameworks for solving image classification problems then presently the case, i.e. deep learning. The characteristics of the CIFAR-10 benchmark are such that multiple criteria have to be addressed. Specifically, multi-class classification, high dimensionality and cardinality, where this is not currently the norm, say, with genetic programming as applied to supervised learning tasks. Moreover, the task is interesting because the source data is highly variable, with comparatively little attempt to control lighting conditions, frame of reference, or diversity of source material (e.g. class content may cover a wide range of species).

We motivate an approach based on tangled program graphs, a genetic programming framework for the emergent discovery of modules and their self-organization into graphs of teams of programs. In order to decouple the process of fitness evaluation from the underlying cardinality of the training data, a data subset is adopted in which a pool of training records is sampled every τ generations. In addition, we demonstrate that lexicase selection is particularly effective at maintaining population diversity. Finally, particular attention is also given to the maintenance of class labels in the population of learners.

Comparison against other light-weight ML solutions to the CIFAR-10 benchmark demonstrates that TPG performs significantly better than AutoML, and comparable with unsupervised feature construction while also being significantly simpler than the latter (by 2 to 3 orders of magnitude). Future work will consider the relative merits of image whitening and speedups such as multi-threading or intron skipping.

References

[1] M. Ranzato, A. Krizhevsky, and G. E. Hinton, "Factored 3-way restricted boltzmann machines for modeling natural images," in *Proceedings of the International Conference on Artificial Intelligence and Statistics*, ser. JMLR Proceedings, vol. 9. JMLR.org, 2010, pp. 621–628.

[2] A. Coates, A. Y. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the International Conference on Artificial Intelligence and Statistics*, ser. JMLR Proceedings, vol. 15. JMLR.org, 2011, pp. 215–223.

[3] A. Krizhevsky, "Learning multiple layers of features from tiny images," Toronto University, Tech. Rep. TR-2009, 2009.

[4] S. Kelly and M. I. Heywood, "Emergent tangled graph representations for Atari game playing agents," in *European Conference on Genetic Programming*, ser. LNCS, vol. 10196, 2017, pp. 64–79.

[5] ——, "Emergent solutions to high-dimensional multitask reinforcement learning," *Evolutionary Computation*, vol. 26, no. 3, 2018.

[6] R. J. Smith and M. I. Heywood, "Scaling tangled program graphs to visual reinforcement learning in ViZDoom," in *European Conference on Genetic Programming*, ser. Lecture LNCS, vol. 10781, 2018, pp. 135–150.

[7] K. Desnos, N. Sourbier, P. Raumer, O. Gesny, and M. Pelcat, "Gegelati: Lightweight artificial intelligence through generic and evolvable tangled program graphs," in *ACM Workshop on Design and Architectures for Signal and Image Processing*, T. Kryjak and A. Pinna, Eds., 2021, pp. 35–43.

[8] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[9] P. J. Darwen and X. Yao, "Speciation as automatic categorical modularization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 2, pp. 101–108, 1997.

[10] K. Imamura, T. Soule, R. B. Heckendorn, and J. A. Foster, "Behavioral diversity and a probabilistically optimal GP ensemble," *Genetic Programming and Evolvable Machines*, vol. 4, no. 3, pp. 235–253, 2003.

[11] R. Thomason and T. Soule, "Novel ways of improving cooperation and performance in ensemble classifiers," in *ACM Genetic and Evolutionary Computation Conference*, 2007, pp. 1708–1715.

[12] S. Y. Chong, P. Tiňo, and X. Yao, "Relationship between generalization and diversity in coevolutionary learning," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 3, pp. 214–232, 2009.

[13] D. P. Muni, N. R. Pal, and J. Das, "A novel approach to design classifiers using genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 2, pp. 183–196, 2004.

[14] P. Lichodzijewski and M. I. Heywood, "Pareto-coevolutionary genetic programming for problem decomposition in multi-class classification," in *ACM Genetic and Evolutionary Computation Conference*, 2007, pp. 464–471.

[15] ——, "Symbiosis, complexification and simplicity under GP," in *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, 2010, pp. 853–860.

[16] G. Folino, C. Pizzuti, and G. Spezzano, "A scalable cellular implementation of parallel genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 1, pp. 37–53, 2003.

[17] I. Arnaldo, K. Veeramachaneni, A. Song, and U. O'Reilly, "Bring your own learner: A cloud-based, data-parallel commons for machine learning," *IEEE Computational Intelligence Magazine*, vol. 10, no. 1, pp. 20–32, 2015.

[18] J. A. Doucette and M. I. Heywood, "GP classification under imbalanced data sets: Active sub-sampling and AUC approximation," in *European Conference on Genetic Programming*, ser. LNCS, vol. 4971, 2008, pp. 266–277.

[19] S. Kelly, R. J. Smith, and M. I. Heywood, "Emergent policy discovery for visual reinforcement learning through tangled program graphs: A tutorial," in *Genetic Programming Theory and Practice XVI*, ser. Genetic and Evolutionary Computation, W. Banzhaf, L. Spector, and L. Sheneman, Eds., 2018, pp. 37–57.

[20] R. J. Smith and M. I. Heywood, "A model of external memory for navigation in partially observable visual reinforcement learning tasks," in *European Conference on Genetic Programming*, ser. LNCS, vol. 11451, 2019, pp. 162–177.

[21] R. I. McKay, "Fitness sharing in genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2000, pp. 435–442.

[22] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An introduction*, 2nd ed. MIT, 2018.

[23] T. Helmuth, L. Spector, and J. Matheson, "Solving uncompromising problems with lexicase selection," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 5, pp. 630–643, 2015.

[24] T. Helmuth, E. Pantridge, and L. Spector, "On the importance of specialists for lexicase selection," *Genetic Programming and Evolvable Machines*, vol. 21, no. 3, pp. 349–373, 2020.

[25] M. Nicolau and J. McDermott, "Genetic programming symbolic regression: What is the prior on the prediction?" in *Genetic Programming Theory and Practice XVII*. Springer, 2019, pp. 201–225.

[26] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: combined selection and hyperparameter optimization of classification algorithms," in *ACM International Conference on Knowledge Discovery and Data Mining*, 2013, pp. 847–855.

[27] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: Efficient and robust automated machine learning," in *Automated Machine Learning - Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Springer, 2019, ch. 6, pp. 113–134.

[28] M. D. McDonnell and T. Vladusich, "Enhanced image classification with a fast-learning shallow convolutional neural network," in *IEEE/INNS International Joint Conference on Neural Networks*, 2015, pp. 1–7.

[29] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710.

[30] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "Evolving the topology of large scale deep neural networks," in *European Conference on Genetic Programming*, ser. LNCS, vol. 10781, 2018, pp. 19–34.