

Gegelati: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs

Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, Maxime Pelcat

▶ **To cite this version:**

Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, Maxime Pelcat. Gegelati: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs. Workshop on Design and Architectures for Signal and Image Processing (DASIP), Jan 2021, Budapest, Hungary. 10.1145/3441110.3441575 . hal-03057652v2

HAL Id: hal-03057652
https://hal.science/hal-03057652v2

Submitted on 20 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Gegelati: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs

K. DESNOS, N. SOURBIER, and P.-Y. RAUMER, Univ Rennes, INSA Rennes, CNRS, IETR - UMR6164, France
O. GESNY, Silicom, France
M. PELCAT, Univ Rennes, INSA Rennes, CNRS, IETR - UMR6164, France

Tangled Program Graph (TPG) is a reinforcement learning technique based on genetic programming concepts. On state-of-the-art learning environments, TPGs have been shown to offer comparable competence with Deep Neural Networks (DNNs), for a fraction of their computational and storage cost. This lightness of TPGs, both for training and inference, makes them an interesting model to implement Artificial Intelligences (AIs) on embedded systems with limited computational and storage resources.

In this paper, we introduce the Gegelati library for TPGs. Besides introducing the general concepts and features of the library, two main contributions are detailed in the paper: 1/ The parallelization of the deterministic training process of TPGs, for supporting heterogeneous Multiprocessor Systems-on-Chips (MPSoCs). 2/ The support for customizable instruction sets and data types within the genetically evolved programs of the TPG model. The scalability of the parallel training process is demonstrated through experiments on architectures ranging from a high-end 24-core processor to a low-power heterogeneous MPSoC. The impact of customizable instructions on the outcome of a training process is demonstrated on a state-of-the-art reinforcement learning environment.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Computing methodologies → Machine learning**.

ACM Reference Format:
K. Desnos, N. Sourbier, P.-Y. Raumer, O. Gesny, and M. Pelcat. 2021. Gegelati: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs. In *Workshop on Design and Architectures for Signal and Image Processing (14 th edition) (DASIP '21), January 18–20, 2021, Budapest (initially), Hungary.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3441110.3441575

1 INTRODUCTION

In less than a decade, Artificial Intelligences (AIs) powered by Deep Neural Networks (DNNs) have outperformed and replaced man-made algorithms in many applicative domains, from computer vision [5] to Natural Language Processing (NLP) [4]. This sudden breakthrough of DNNs is largely due to the availability of affordable and easily programmable hardware offering important computing power, such as Graphics Processing Units (GPUs) [12]. Powered by the ever-increasing computing power of commercial chips, the current race for omnipotent AIs leads to the creation of more and more complex DNNs where millions [5] to hundreds of billions [4] parameters are needed.

Conversely to the increasing computational complexity of DNNs, the need for lightweight AIs is also growing. Indeed, the ubiquity of Internet of Things (IoT) devices, and the tremendous amount of data they generate [6], call for new paradigms where data processing is performed on-site, close to the data producer. The processing of data by AIs on embedded IoT devices is not compatible with the use of compute, memory and power-hungry DNNs. A common way to create lightweight AIs is to exploit the resilience of DNNs to approximation, and to simplify them as much as possible while maintaining their accuracy to acceptable levels. Pruning techniques, customizable data precision, and

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

approximate computing techniques are examples of techniques for reducing DNNs complexity. An alternative way to create lightweight AIs is to develop new machine learning techniques that rely on light-by-construction models, such as the TPG model studied in this paper.

TPG, which stands for Tangled Program Graph, is a machine learning model proposed by Kelly and Heywood in [10]. Building on state-of-the-art genetic programming techniques, Tangled Program Graphs (TPGs) are grown from scratch for each learning environment in which they are trained. Hence, the topology and the complexity of the TPG adapt themselves to the complexity of the learned task, without requiring an expert to select an appropriate network structure. In recent works [8, 10, 11], TPGs have proven to be a very promising model for building AIs, being competitive with state-of-the-art DNNs for a fraction of their computation and storage cost, both for training and inference.

This paper introduces Gegelati[1], an open-source library for the learning and inference of AIs modeled with TPGs. The objective of Gegelati, coded in C++, is to foster the development of efficient, lightweight and portable AIs, supporting both general-purpose and embedded hardware. The modular structure of the library fosters its extensibility and customizability to ease its evolutions and its adaptations for new learning tasks.

The principles of TPG-based AIs and related works are presented in Section 2. Section 3 presents the Gegelati library. Two distinctive features of Gegelati are detailed in this section. First, the parallel, scalable, and yet deterministic training of TPGs on heterogeneous multicore architectures is introduced in Section 3.1. Second, the support for customizable instruction sets for easing the training of TPGs in diverse learning environments is detailed in Section 3.2. Experiments on various learning environments are presented in Section 4, demonstrating the scalable performance and the customizability of the library. Finally, Section 5 concludes this paper.

2 TANGLED PROGRAM GRAPHS

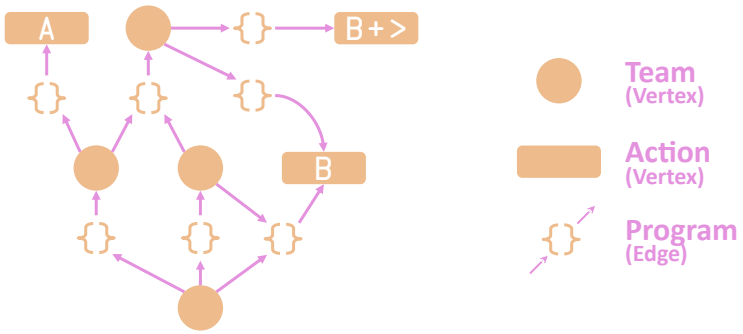
The TPG model studied in this paper, which builds on technique from the genetic programming domain, was introduced by Kelly and Heywood [10] as a reinforcement learning technique. Principles of reinforcement learning and genetic programming are presented in Section 2.1, and the TPG model is detailed in Section 2.2.

2.1 Background: Reinforcement Learning and Genetic Programming

Reinforcement learning is a branch of machine learning techniques where artificial intelligence learns, through trial and error, how to interact with an environment. In reinforcement learning, artificial intelligence, called the learning agent, observes the current state of its learning environment, and interacts with it trough a finite set of actions. As a result of these actions, or because of external phenomena such as time or physics, the state of the learning environment evolves. By observing the constantly evolving state of the environment, the learning agent has the possibility to react and to build a meaningful sequence of actions. For the agent to learn which sequences of actions are useful, an additional reward mechanism is implemented. By rewarding useful behavior of the learning agent, and penalizing harmful or useless behavior, this reward mechanism helps the learning agent select the most appropriate behavior for each new experience. Although TPGs have originally been developed for reinforcement learning purposes, the possibility to adapt them for other kinds of learning environments has already been demonstrated [11].

Genetic programming is a subset of machine learning techniques that mimics the natural selection evolution process to breed programs for a selected purpose. The iterative learning process of genetic programming can be summarized in four steps: 1/ Create an initial population of $n \in \mathbb{N}^*$ random programs. Then, iteratively: 2/ Evaluate the

[1]Generic Evolvable Graphs for Efficient Learning of Artificial Tangled Intelligence (Gegelati)



(a) TPG example (b) TPG semantics

Fig. 1. Semantics of the Tangled Program Graphs (TPGs)

fitness of these programs against the learning environment. 3/ Discard the $m < n, m \in \mathbb{N}^*$ programs of the population with the worse fitnesses. 4/ Recreate m new programs from remaining programs by using genetic operations, like mutations or crossovers. As detailed in [8, 10], TPGs add a compositional mechanism to this genetic learning process, which favors the emergence of stable clusters of useful programs by building a hierarchical decision structure.

2.2 TPG: Model and Learning Algorithm

The semantics of the Tangled Program Graph (TPG) model, depicted in Figure 1, consists of three elements composing a direct graph: *programs*, *teams* and *actions*. The *teams* and the *actions* are the vertices of the graph, *teams* being internal vertices, and *actions* being the leaves of the graph. The *programs*, associated to the edges of the graph that each connects a source *team* to a destination *team* or *action* vertex. Self-loops, that is an edge connecting a *team* to itself, are not allowed in TPGs.

From afar, a *program* can be seen as a black box that takes the current state of the learning environment as an input, processes it, and produces a real number, called a *bid*, as a result. In more detail, a *program* is a sequence of simple arithmetic *instructions*, like additions or exponents. As depicted in Figure 2, each *instruction* takes as an operand either data coming from the observed learning environment, or the value stored in a register by a previous *instruction*. The last value stored in a specific register, generally called `R0`, is the result produced by the *program*.

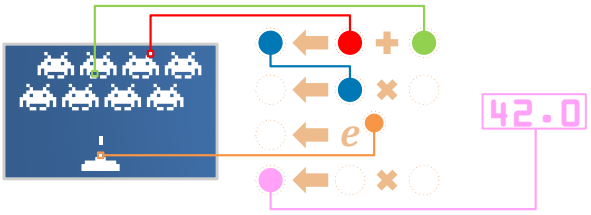


Fig. 2. *Program* from a TPG. On the left, the learning environment state fed to the *program*. In the middle, the sequence of instructions of the *program*. On the right, the result produced by the program.

The execution of a TPG starts from its unique root *team*, when a new state of the environment becomes available. All *programs* associated to outgoing edges of the root *team* are executed with the current state of the environment as their input. Once all *programs* have completed their execution, the edge associated to the largest *bid* is identified, and the execution of the TPG continues following this edge. If another *team* is pointed by this edge, its outgoing *programs* are executed, still with the same input state, and the execution continues along the edge with the largest *bid*[2]. Eventually,

[2]If a team is visited several times, previously taken edges are ignored to avoid infinite loops.

the edge with the largest *bid* leads to an *action* vertex. In this case, the *action* is executed by the learning agent, a new resulting state of the environment is received, and the TPG execution restarts from its root *team*.

The genetic evolution process of a TPG relies on a graph with several root *teams*. The initial TPG created for the first generation only contains root *teams* whose outgoing edges each lead directly to an *action* vertex. At a given generation of the learning process, each root *team* of the TPG represents a different *policy* whose fitness is evaluated. Evaluating a root *team* consists of executing the TPG stemming from it a fixed number of times, or until a terminal state of the learning environment is reached, like a game-over in a video game. The rewards obtained after evaluating each root *team* of the TPG are used by the genetic evolution process. Worst-fitting root *teams*, which obtained the lowest rewards, are deleted from the TPG.

To create new root *teams* for the next generation of the evolution process, randomly selected remaining *teams* from the TPG are duplicated with all their outgoing edges. Then, these new edges undergo a random mutation process, possibly altering their destination vertex, and modifying their *programs* by adding, removing, swapping, and changing their instructions and operands. Surviving root *teams* from previous generations may become the destination of an edge added during the mutation process, thus becoming internal vertices of the TPG. This mutation mechanism favors the emergence of long-living valuable subgraphs of connected *teams*. Indeed, useful *teams* contributing to higher rewards have a greater chance of becoming internal vertices of the TPG which can not be discarded unless they become root *teams* again. Hence, complexity is added to the TPG adaptively, only if this complexity leads to better rewards for the learning agent. A detailed description of this evolution process can be found in [8].

The capabilities of TPGs have been extensively demonstrated [8, 10] on the 55 video games from the Arcade Learning Environment (ALE) [3]. In this learning environment, the adaptive complexity leads to TPG with diverse sizes, depending on the complexity of the strategies developed to play each game. For example, there are two orders of magnitude between the smallest and largest networks built within these learning environments. On the performance side, TPGs have been shown to reach a level of competency comparable with state-of-the-art deep-learning techniques on ALE games, for a fraction of their computational and storage cost. Compared to state-of-the-art techniques, TPGs reach comparable competency with one to three orders of magnitude less computations, and two to ten orders of magnitude less memory needed to store their inference model. Recently, an extension of the TPG model supporting continuous action space was proposed in order to target new learning environments, like time-series predictions [11].

Implementations of learning frameworks for TPG, coded in C++, Java and Python, can be found in open-source repositories. The main motivations behind the creation of the Gegelati library is the desire to have an efficient, embeddable, portable, parallel and deterministic library. Because of the efficiency and embeddability objectives, C++ was a natural choice for the development of Gegelati. Previous open-source C++ implementations, including the reference C++ code from Kelly [10], were neither parallel nor deterministic. The creation of a new library from scratch was further motivated by the low code quality of existing C++ implementation, notably due to a lack of code documentation and a monolithic code.

The purpose of this paper is not to advocate the learning efficiency of TPGs against other machine learning techniques, which was already done in [8, 10]. Instead, this paper intends to present the original contributions for creating a customizable, scalable and deterministic implementation of TPGs.

3 GEGELATI: PARALLEL, EFFICIENT AND EMBEDDABLE LIBRARY FOR TPGS

Gegelati is an open-source framework, developed as a C++ library, for training and executing TPGs. From its inception, the Gegelati library has been conceived to foster its adaptability to diverse learning environments, and its portability to

various architectures, without sacrificing its performance. To this purpose, two original contributions have be integrated to the library: the parallelization of the deterministic learning process, presented in Section 3.1; and the support for customizable *instructions*, detailed in Section 3.2. An overview of additional features of the library is presented in Section 3.3.

3.1 Deterministic Parallelism and Portability

What are the motivations? Portability of the GEGELATI library enables using it both on general-purpose and embedded architectures. Indeed, when training a learning agent intended to run on an embedded system, a common design process is to prototype the agent first on a general-purpose processor before embedding it on the embedded target. The portability also makes it possible to train a learning agent offline on a high-performance computing architecture, before deploying it on a less performing architecture for inference.

Parallelism of the learning process is an essential feature to accelerate the training of new learning agents, which fosters the adoption of new machine learning techniques. Indeed, the breakthrough of deep-learning models is largely due to the acceleration of their training process with GPUs [12]. Support for parallel computations is useful for general-purpose and high-performance computing architectures, but also for embedded systems which nowadays widely integrate heterogeneous Multiprocessor Systems-on-Chips (MPSoCs).

Determinism of a learning process is the property that ensures that given a set of initial conditions, the learning process will always end with the same result. Determinism can only be obtained under the assumption that the state of the learning environment is itself changing deterministically, solely depending on the sequence of actions applied to it. Determinism is a key feature, especially for a pseudo-stochastic learning process such as the training of TPGs. Indeed, the result of training may partially depend on luck, which is exactly why being able to deterministically reproduce a result is crucial.

The determinism is antagonistic with the parallelism and portability objectives, and with the stochastic nature of the learning process, which makes all these objectives challenging to implement jointly. Indeed, parallelism is by nature a source of non-determinism as the simultaneity of computations accessing and modifying shared resources, often in an unknown order, tends to produce variable results.

How does the deterministic and scalable parallelism work? During the learning process of TPGs, the most compute-intensive parts are the fitness evaluation of the *policies*, and the mutations of the *programs* added during the evolution process. The fitness evaluation of individual *policies* can be deterministically executed in parallel, on the conditions that: 1/ the learning environment can be cloned to evaluate several policies concurrently, and 2/ any stochastic evolution of the learning environment state can be controlled deterministically. Under these conditions, the parallel evaluation of *policies* is possible, as the topology of the TPG, which is a shared resource for all *policies*, is fixed during this evaluation process. Similarly, the mutation of *programs* can be applied deterministically in parallel. Two kinds of mutations are applied to the TPG: mutations affecting the graph topology by inserting new root *teams* and edges; and mutations affecting *instructions* of the *programs* associated with the new edges. While mutating the graph topology cannot be done in parallel, the graph being a shared resource, individual *programs* are independent from each other and can be mutated in parallel.

To control a stochastic process, a Pseudo-Random Number Generator (PRNG) must be used each time a random number is needed. Given an initial seed, a PRNG produces a deterministic sequence of numbers. To ensure full determinacy of the training of a TPG, a unique PRNG should be called in a fixed order during the whole training. Letting the parallel parts of the training process call the PRNG directly is not possible, as the absolute order in which

Procedure 1: EvaluateAllPolicies

```
Input: TPG: G = ⟨Teams, Edges⟩
Data:  PRNG: prng_master
       Job queue: JobQ
       Result Queue: ResultQ
/* Prepare jobs */
idx = 0
for each root ∈ G.Teams do
    seed = prng_master.getNumber()
    job = { idx++, seed, root }
    jobQ.push(job)
endfor
/* Start parallel threads */
for i = 1 to Num_PE − 1 do
    Spawn thread: Worker(G, JobQ, ResultQ)
end
Call Worker(G, JobQ, resultQ)
Join all threads
/* Post-Process Results and Trace */
Sort ResultQ in result.jobId order
for each result ∈ resultQ do
    Post-process result.trace // Archiving [10]
    ...
endfor
```

Procedure 2: Worker

```
Input: TPG: G = ⟨Teams, Edges⟩
       Job queue: JobQ
       Result queue: ResultQ
Data:  PRNG: prng_worker
       Learning environment twin: LE
/* Poll for job */
while JobQ.hasJob() do
    /* Setup for policy evaluation */
    job = jobQ.getNextJob()
    root = job.root
    prng_worker.reset(job.seed)
    LE.reset(prng_worker.getNumber())
    /* Evaluate policy fitness */
    trace = evaluate(G, root, LE, prng_worker)
    result.jobId = job.id
    result.trace = trace
    resultQ.push(result)
end
```

parallel computations occur is itself stochastic. It is also not possible to give a pre-computed list of pseudo-random numbers to each parallel task, as the number of random numbers needed for each task is itself stochastic. For example, when mutating a *program*, mutations are applied iteratively until the program behavior becomes "original" compared to pre-existing *programs* in the TPG. Hence, giving a fixed number of pre-computed random numbers for the *program* mutations is not feasible.

The parallelization strategy adopted in Gegelati is based on the master/worker principle, with a distributed PRNG. The principle of the distributed PRNG is the use of two distinct PRNG instances: the $prng_{master}$ and the $prng_{worker}$. The $prng_{master}$ is exclusively used in the sequential parts of the learning process, which confers a deterministic nature to its usage, given an initial seed. Besides being used for stochastic tasks performed sequentially, like TPG topology mutations for example, the $prng_{master}$ is also used to generate a seed for each parallel worker task. In each worker task, a private $prng_{worker}$ is instantiated, and initialized with the seed provided by the $prng_{master}$. Since all calls to the PRNG from the worker tasks exclusively use their private $prng_{worker}$, the random number sequences generated in each parallel task are deterministic.

The pseudo-code of the master and worker tasks for the policy fitness evaluation are presented in Procedures 1 and 2, respectively. Communications between the tasks and load balancing of the computations are supported by a job queuing mechanism based on two queues: *JobQ* and *ResultQ*. Each policy evaluation job, prepared by the master procedure, encapsulates a unique job identifier `id`, a seed provided by the $prng_{master}$, and a root *team* from the TPG. All jobs are pushed in the *JobQ* queue before spawning as many worker threads as the number of secondary Processing Elements (PEs) in the target architecture. For each job it acquires from the *jobQ* queue, the worker procedure resets its

$prng_{worker}$ using the seed contained in the job. Before evaluating the fitness of the root *team* contained in a job, the worker procedure resets its private copy of the learning environment, using a number given by the $prng_{worker}$. As a result of the policy fitness evaluation, described in details in [10], a result object encapsulating execution traces for the job is pushed in the *resultQ*. When all jobs have been processed, and all workers terminated, the master procedure is responsible for post-processing the traces stored in the *resultQ*. To ensure determinism of this post-processing, results stored in the *resultQ* are first sorted in ascending `job.id` order.

The master and worker procedures used for parallelizing the mutations of *programs* are similar to the one used for policy fitness evaluation, with the difference that jobs encapsulate *programs* instead of root *teams*. In Section 4, experiments on three different multicore architectures show the scalability and load balancing capability of the proposed deterministic and parallel TPG implementation.

3.2 Customizable Instruction Set

What are the motivations? In the seminal work on TPGs [10], the *instructions* used in the *programs* are chosen exclusively among the following eight instructions: 4 binary operators $\{+, -, \times, \div\}$, 3 mathematical functions $\{cos, ln, exp\}$, and 1 conditional statement $res \leftarrow (a < b)? - a : a$. To further simplify the execution and mutation of *programs*, it was assumed that instructions only handle `double` operands.

As shown in related genetic programming works [2, 14], using a broader set of instructions with diverse data types can help improve the performance of learning agents, at the cost of longer training time. The extension of the instruction set used in the *programs* of the TPGs has already been proposed in [9], where a set of instructions for 2D images operands is added, and in [7], with instructions accepting thirteen operands tailored for predicting properties of the learning environment. In GEGELATI, both the number and types of operands, and the nature of instructions used in *programs* can be fully customized. Besides making the training more efficient for specific learning environments, this customization feature may also be used to increase the efficiency of the TPG execution on specific hardware. Indeed, using an instruction set mirroring the instruction set of the architecture used for its execution may help increase the speed and the power efficiency of the TPG execution.

How are customizable instructions supported? The support for customizable instructions within GEGELATI is based on the three classes presented in Figure 3. When creating a new training environment, a developer may create her own set of instructions, by creating new classes inheriting from the `Instruction` class. With the `operandTypes` attribute, each instruction declares the number and type of operands it accepts when calling its `execute()` method. Currently, to keep the management of registers simple during program execution, only `double` results can be produced by the `execute()` method. Each *line* of a *program* references an `instruction` from the set of available instructions, a destination `register` to store the result of its execution, and the addresses of operands to process, selected among all available data sources. The data sources accessible to the *lines* comprise both the registers used for storing instruction results, and the state of the learning environment. Data sources classes must inherit from the `DataSource` class which acts as a wrapper between the data and the *program* execution engine.

Procedure 3 presents the simplified pseudo-code for executing a *program* modeled with the classes from Figure 3. The core of the mechanism supporting customizable instructions lies between lines 5 and 14 of Procedure 3. For each operand of each *line*, the algorithm checks whether the data sources can provide the requested operand type at the requested address. If the data type can be provided by the data sources at the requested address, the data is fetched from the data sources, and later used for executing the `instruction` of the current `line` of the *program*. Otherwise, the program execution is terminated, which does not occur in practice, as the operand data types are taken into consideration when

performing *program* mutations in Gegelati. It is important to note that the `getData()` method may return data whose type differs from the native data type stored within the data source. For example, a data source storing screen pixels as `char` values can automatically return an equivalent `double` value, or even a neighborhood of 3-by-3 pixels when an operand of type `char[3][3]` is requested.

To ease the creation of new *instructions* for each training environment, a utility class `LambdaInstruction` is proposed in Gegelati. The template class `LambdaInstruction` supports the creation of instructions for any number of operands, and for operands with primitive and non-primitive types as well as 1D and 2D C-Style arrays. A code snippet illustrating the creation of an instruction with the `LambdaInstruction` class is given in Listing 1. In this example, an instruction taking an `int` operand, and a 1D array of `char` is declared, using a simple C++ lambda function.

```
auto myInstruction = LambdaInstruction<int, char[2]>(
  [](int a, char[2] b)->double {return a*(b[0] + b[1]);});
```

Listing 1. `LambdaInstruction` usage example

3.3 Library Features

In addition to the two contributions detailed in previous sections, the Gegelati library showcases several important features for easing its integration in new projects.

Configuration and logging support. Interfacing a machine learning library with third-party tools, notably scripting and data analysis tools, is an indispensable feature. To this purpose, Gegelati supports the import and export of information in several standard file formats. A `json` importer is available for configuring the meta parameters of the TPG training process. This `json` importer eases the exploration of many meta parameter configurations from a simple script, without having to re-build the library code. A TPG exporter in the `dot` format can be used to visualize the topology of the graph along the training process. The exported `dot` files also embed all the information needed to import a pre-trained TPG.

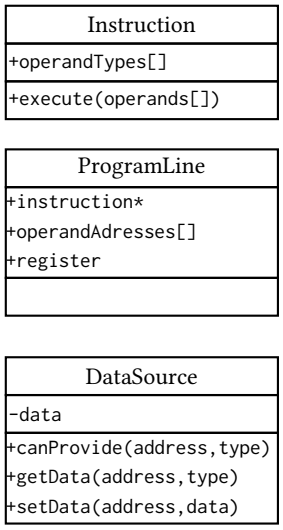


Fig. 3. Class diagrams of the data structures for customizable instructions.

Procedure 3: ExecuteProgram

Input: Program: *p*
Data sources: *data*

```
1  for each line ∈ p do
2      instruction = line.instruction
3      operands[] = { ∅ }
4      nbOperands = instruction.operandTypes.size()
5      for i = 0 to nbOperands-1 do
6          type = instruction.dataTypes[i]
7          address = line.operandAdress[i]
8          if data.canProvide(type, address) then
9              operand = data.getData(type, address)
10             operands.insert(operand)
11         else
12             Exit with an error
13         end
14     end
15     result = instruction.execute(operands)
16     data.set(line.register,result)
17 endfor
```

A logging system exporting training statistics in the `csv` format is also implemented in Gegelati. Through a simple class specialization mechanism, this logging system may be extended to export information specific to a user's learning environment.

Code quality. Maintaining a high quality of code is imperative to foster the maintainability, the evolutivity, and the trustworthiness in a library. Strict coding rules are enforced in Gegelati, notably by automatically checking that all data structures and functions are properly commented, and causing a build failure otherwise. Publicly hosted continuous integration is used to ensure that code coverage of the unit tests cover 100% of the library code [1]. This continuous integration also checks the portability of the library on Linux with gcc and clang, on Windows with MSVC19, on and macOS with clang. Public continuous code quality analysis is also used to enforce the reliability, the security, and the maintainability of the code [1].

Extension-friendly code modularity. Adopting a modular and hierarchical code organization of the code favors its evolution and its extension, either by replacing existing modules, or by adjuncting new ones. In Gegelati, separate modules with as few inter-dependencies as possible have been set up for each concept in the code. For example, classes modeling the topology of a TPG do not depend on the code responsible for mutating it, or the code for executing it. This separation of concern can be exploited to create specialization of a given module for a specific purpose. For example, while the default learning algorithm assumes a classic reinforcement learning setup with one agent interacting with its environment, an alternative algorithm has been implemented to train two or more agents to play competitive games. Another alternative learning algorithm is also available for learning environments representing a classification problem.

Open source applications. The public availability of projects based on a library is a way to demonstrate in a reproducible way the capabilities of this library. Existing projects can also serve as examples or even as a base, for a developer starting a new project with a library. The public repository of Gegelati [1] hosts several projects with diverse learning environments, including: a wrapper for the Arcade Learning Environment (ALE) [3], simple 2-player tic-tac-toe and Nim games, a physical robotic arm control and an inverted pendulum environments, and a MNIST [13] classification environment.

4 EXPERIMENTS

The Gegelati implementation of the TPG learning process is evaluated in different scenarios to assess its capabilities and performance. Version `0.4.0` of the Gegelati was used in all experiments. In Section 4.1, the rapidity and the scalability of the training process is evaluated on several multicore architectures. In Section 4.2, the potential of customizable instructions is demonstrated on an inverted pendulum environment.

4.1 Sequential Performance and Scalability

Sequential. The implementation of the contributions presented in Section 3 incurs an additional complexity in the code of Gegelati compared to the reference TPG implementation by Stephen Kelly [8, 10]. To assess the efficiency of Gegelati compared to this reference implementation, the two codes have been used to train TPGs in the ALE environment. The meta-parameters values defined in [8] have been used in all experiments, except the number of games played per evaluation, which was set to 1 to accelerate the experiments. TPGs were trained for 50 generations on 5 games with diverse complexities: `alien`, `asteroids`, `centipede`, `fishing_derby`, and `frostbite`. Due to the non-deterministic stochastic nature of the Atari 2600 emulator powering the ALE [3], 5 runs were executed for each game. This experiment was executed on one core of an Intel Xeon CPU E5-2690 processor, running with Linux.

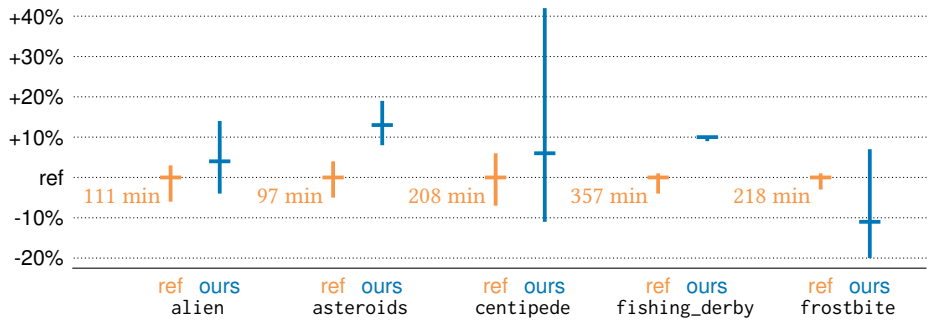


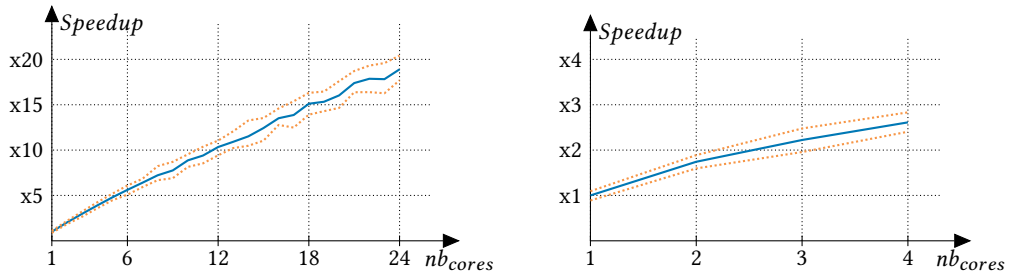
Fig. 4. Relative sequential training time of Gegelati (ours) and Kelly's code (ref) [10]. On each game, TPGs were trained 5 times, for 50 generations. For each game and code couple, the vertical line spans from the minimum to the maximum training times, and the horizontal line is the average training time. All times are relative to the average ref training time printed in the plot.

Training time of the TPGs measured in this experiment are plotted in Figure 4. For each game and implementation, the plot illustrates the minimum, maximum, and average training time, relatively to the average training time of the reference implementation. The absolute training time, in minutes, of the reference implementation is printed in the plot.

As can be seen in the experimental results in Figure 4, Gegelati is on average only 4.4% slower than the reference code. In the best case, for the `frostbite` game, Gegelati is on average 11% faster than the reference code, while it is on average 13% slower in the worst case, for the `asteroids` game. In these experiments, the variability of the training times, revealed by the min-max ranges, seems to be more important with Gegelati, which indicates a greater sensibility to variations of the TPG topology induced by the training process. The stochastic nature of the learning environment impacts the fitness of the trained agent, the chances of reaching a game-over, and the topology of the TPG, which are as many variables impacting the training time. Due to these numerous variables, an in-depth analysis of these results is out of the scope of this paper. Nevertheless, a fair conclusion to this experiment is that, despite the overhead of its additional mechanisms for supporting deterministic parallelism and customizable instructions, the sequential performance of Gegelati are only slightly lower than the reference code.

Parallel scalability. To assess the efficiency, the portability, and the scalability of the parallelization strategy implemented in Gegelati, experiments were conducted on 3 distinct architectures: 1/ a high-performance Intel Xeon CPU E5-2690 processor with 24 cores, 2/ an Intel i7-8650U CPU with 4 cores, and 3/ a Samsung Exynos5422, an embedded heterogeneous chip with four Arm A15 cores, and four A7 cores. To assess the scalability of the parallelism on each target, the training time was measured for a number of thread varying between 1 and the number of physical core of the architecture. No affinity constraint was set on the threads, thus letting the Linux operating system handle the scheduling. On the Exynos chip, the `taskset` command was used to control the number of cores of each type, A15 and A7, available to the threads. For each number of threads, and for each game on the Intel CPUs, the experiment was repeated 5 times with different seeds.

On the Xeon and i7 CPUs, the ALE learning environment was used, with the same 5 games and meta-parameters as in the sequential experiments. On the Exynos CPU, a more lightweight learning environment, the inverted pendulum learning environment was used as a more realistic use case for an embedded processor. For each number of threads, the experiment was conducted 5 times with different seeds. On the pendulum learning environment, full determinism of the training process is observed, in all configurations.



(a) Intel Xeon CPU (24 cores) (b) Intel i7 CPU (4 cores)

Fig. 5. Training Scalability on Homogeneous Multicore CPUs

Figure 5 presents the results of the scalability evaluation of the training process of GEGELATI on the homogeneous multicore CPUs. For each target architecture, the speedup compared to the sequential training time is plotted for a varying number of threads running the worker process. The thick line represents the average speedup observed for all configurations, and the dotted lines represent the minimum and maximum speedup observed. As can be seen in these results, the parallelization of the learning process is highly scalable, with an average speedup of 18.9 on the 24 cores of the Xeon CPU, and a speedup of 2.61 on the 4 cores of the i7 CPU.

Figure 6 presents the experimental results for the Exynos chip. On this heterogeneous chip, the average speedup is 4.12 when using all 8 cores of the architecture, compared to a single A15 core. There is a clear benefit in using the four A7 cores, in addition to the four A15 cores, as the average speedup when using only the four A15 cores is only 2.32.

In experiments based on the ALE games, the training time is dominated by the computation time needed to evaluate the fitness of each policy by playing the game, which takes up 99.8% of all training time. On the pendulum learning environment, 40% of the training time is spent performing the graph mutations. In this environment, mutations are sped-up by a factor 3.80 on average, and policy evaluation by a factor 4.41, on the 8 heterogeneous cores of the Exynos chip.

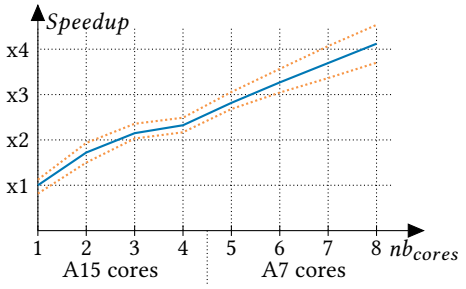


Fig. 6. Scalability on Exynos5422 (8 big.LITTLE cores)

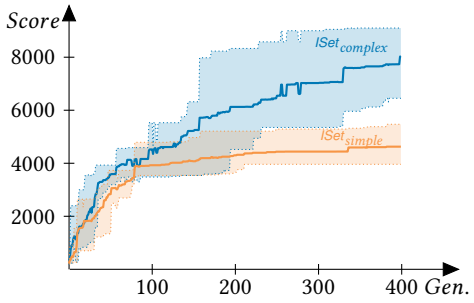


Fig. 7. Champion policy score on `frostbite` over 400 generations with 2 instruction sets. For each set, the thick line represents the champion policy score averaged on 5 trainings. The colored background covers the area from the minimum to the maximum champions score among the 5 trainings.

4.2 Customizable Instruction Sets

To illustrate the utility of customizable instructions, GEGELATI was trained for 400 generations on the `frostbite` game from the ALE, using two different sets of instructions, namely $ISet_{simple}$ and $ISet_{complex}$. The $ISet_{complex}$ set contains the eight instructions from Kelly's work: $\{+, -, \times, \div, cos, ln, exp, <\}$, and the $ISet_{simple}$ contains only the following five low-complexity instructions $\{+, -, \times, \div, <\}$. For each instruction set, the training was performed with 5 different seeds.

For each instruction set, Figure 7 plots statistics of the score of the champion policies observed during training. The thick lines represent the average score of the champion policies during the training process. The surrounding colored zone covers the area between the minimum and maximum scores observed during the five trainings for each instruction set. The trainings based on the $ISet_{complex}$ instruction set outperform the trainings based on the $ISet_{simple}$ set. On average, the score of the champion policies with the $ISet_{simple}$ set plateau at 4626 points, whereas the champion policies built with the $ISet_{complex}$ set reach 8042 points. The gap between the training efficiencies with the two instruction sets is also revealed by the distance between the best score obtained after 400 generations for the $ISet_{simple}$ set: 5470 points, which is far below the worst score obtained with the $ISet_{complex}$ set: 6450 points. This experiment shows that by providing support for customizable instruction sets, Gegelati unlocks the possibility to tailor the training instruction set to maximize the training efficiency.

5 CONCLUSION

The Gegelati C++ library for the training of AIs based on TPGs was introduced in this paper. Two original features of the library, namely its scalable and deterministic parallelism, and its customizable instructions, confer Gegelati great portability on various types of architectures. The raw performance, the scalability, and the customizable efficiency of the training process of TPGs was assessed experimentally using state-of-the-art reinforcement learning environments. A direction for future work is the creation of a methodology for selecting appropriate instruction sets to maximize the efficiency of the training process, while jointly optimizing the time and energy efficiency of selected instructions.

REFERENCES

[1] 2020. Gegelati repository. https://github.com/gegelati/. [Online; accessed 17-September-2020].

[2] Daniel Atkins, Kourosh Neshatian, and Mengjie Zhang. 2011. A domain independent genetic programming approach to automatic feature extraction for image classification. In *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, 238–245.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47 (jun 2013), 253–279.

[4] Brown T. et al. 2020. Language Models are Few-Shot Learners. (2020). arXiv:2005.14165 [cs.CL]

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678* (2016).

[6] CISCO. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016 – 2021. White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf

[7] Olivier Gesny, Pierre-Marie Satre, and Julien Roussel. 2018. CBWAR: Classification de Binaires Windows via Apprentissage par Renforcement. In *Computer & Electronics Security Applications Rendez-vous (C&ESAR)*. https://www.cesar-conference.org/wp-content/uploads/2018/11/articles/C&ESAR_2018_J2-16_O-GENTRY_CBWAR.pdf

[8] Stephen Kelly. 2018. *Scaling genetic programming to challenging reinforcement tasks through emergent modularity.* Ph.D. Dissertation. Dalhousie University, Halifax, Nova Scotia, Canada.

[9] Stephen Kelly and Wolfgang Banzhaf. 2020. *Temporal Memory Sharing in Visual Reinforcement Learning.* Springer International Publishing, Cham, 101–119.

[10] Stephen Kelly and Malcolm I Heywood. 2017. Emergent tangled graph representations for Atari game playing agents. In *European Conference on Genetic Programming*. Springer, 64–79.

[11] Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. 2020. A Modular Memory Framework for Time Series Prediction. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 949–957. https://doi.org/10.1145/3377930.3390216

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[14] Esteban Real, Chen Liang, David R. So, and Quoc V. Le. 2020. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *Proceedings of the 37th International Conference on Intelligent User Interfaces.* arXiv:2003.03384 [cs.LG]