Evolving simple programs for playing Atari games

Dennis G Wilson
dennis.wilson@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France

Sylvain Cussat-Blanc
sylvain.cussat-blanc@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France

Hervé Luga
herve.luga@irit.fr
University of Toulouse
IRIT - CNRS - UMR5505
Toulouse, France

Julian F Miller
julian.miller@york.ac.uk
University of York
York, UK

ABSTRACT

Cartesian Genetic Programming (CGP) has previously shown capabilities in image processing tasks by evolving programs with a function set specialized for computer vision. A similar approach can be applied to Atari playing. Programs are evolved using mixed type CGP with a function set suited for matrix operations, including image processing, but allowing for controller behavior to emerge. While the programs are relatively small, many controllers are competitive with state of the art methods for the Atari benchmark set and require less training time. By evaluating the programs of the best evolved individuals, simple but effective strategies can be found.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; *Model development and analysis*;

KEYWORDS

Games, Genetic programming, Image analysis, Artificial intelligence

1 INTRODUCTION

The Arcade Learning Environment (ALE) [1] has recently been used to compare many controller algorithms, from deep Q learning to neuroevolution. This environment of Atari games offers a number of different tasks with a common interface, understandable reward metrics, and an exciting domain for study, while using relatively limited computation resources. It is no wonder that this benchmark suite has seen wide adoption.

One of the difficulties across the Atari domain is using pure pixel input. While the screen resolution is modest compared to modern game platforms, processing this visual information is a challenging task for artificial agents. Object representations and pixel reduction schemes have been used to condense this information into a more palatable form for evolutionary controllers. Deep neural network controllers have excelled here, benefiting from convolutional layers and a history of application in computer vision.

Cartesian Genetic Programming (CGP) also has a rich history in computer vision, albeit less so than deep learning. CGP-IP has capably created image filters for denoising, object detection, and centroid determination. There has been less study using CGP in reinforcement learning tasks, and this work represents the first use of CGP as a game playing agent.

The ALE offers a quantitative comparison between CGP and other methods. Atari game scores are directly compared to published results of multiple different methods, providing a perspective on CGP's capability in comparison to other methods in this domain.

CGP has unique advantages that make its application to the ALE interesting. By using a fixed-length genome, small programs can be evolved and later read for understanding. While the inner workings of a deep actor or evolved neural network might be hard to discern, the programs CGP evolves can give insight into strategies for playing the Atari games. Finally, by using a diverse function set intended for matrix operations, CGP is able to perform comparably to humans on a number of games using pixel input with no prior game knowledge.

This article is organized as follows. In the next section, § 2, a background overview of CGP is given, followed by a history of its use in image processing. More background is provided concerning the ALE in §§ 2.3. The details of the CGP implementation used in this work are given in § 3, which also covers the application of CGP to the ALE domain. In § 4, CGP results from 61 Atari games are compared to results from the literature and selected evolved programs are examined. Finally, in § 5, concerns from this experiment and plans for future work are discussed.

2 BACKGROUND

While game playing in the ALE involves both image processing and reinforcement learning techniques, research on these topics

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205578

using CGP has not been equal. There is a wealth of literature concerning image processing in CGP, but little concerning reinforcement learning. Here, we therefore focus on the general history of CGP and its application to image processing.

2.1 Cartesian Genetic Programming

Cartesian Genetic Programming [16] is a form of Genetic Programming in which programs are represented as directed, often acyclic graphs indexed by Cartesian coordinates. Functional nodes, defined by a set of evolved genes, connect to program inputs and to other functional nodes via their coordinates. The outputs of the program are taken from any internal node or program input based on evolved output coordinates.

In its original formulation, CGP nodes are arranged in a rectangular grid of R rows and C columns. Nodes are allowed to connect to any node from previous columns based on a connectivity parameter L which sets the number of columns back a node can connect to; for example, if $L = 1$, nodes could connect to the previous column only. Many modern CGP implementations, including that used in this work, use $R = 1$, meaning that all nodes are arranged in a single row [17].

Recurrent CGP showed performance improvements on time-based tasks, as shown in Turner and Miller [26]. Here, a recurrency parameter r was introduced to express the likelihood of creating a recurrent connection; when $r = 0$, standard CGP connections were maintained, but r could be increased by the user to create recurrent programs. This work uses a slight modification of the meaning of r, but the idea remains the same.

In practice, only a small portion of the nodes described by a CGP chromosome will be connected to its output program graph. These nodes which are used are called "active" nodes here, whereas nodes that are not connected to the output program graph are referred to as "inactive" or "junk" nodes. While these nodes are not used in the final program, they have been shown to aid evolutionary search [15, 28, 30].

The functions used by each node are chosen from a set of functions based on the program's genes. The choice of functions to include in this set is an important design decision in CGP. In this work, the function set is informed by MT-CGP [5] and CGP-IP [6]. In MT-CGP, the function of a node is overloaded based on the type of input it receives: vector functions are applied to vector input and scalar functions are applied to scalar input. The choice of function set is very important in CGP. In CGP-IP, the function set contained a subset of the OpenCV image processing library and a variety of vector operations.

2.2 Image Processing

CGP has been used extensively in image processing and filtering tasks. In Montes and Wyatt [21], centroids of objects in images were determined by CGP. A similar task was more recently undertaken in Paris et al. [22], which detected and filtered simple shapes and musical notes in images. Other image filters were evolved in Smith et al. [25] and Sekanina et al. [24] which involved tasks such as image denoising. Finally, Harding [4] demonstrated the ability to use GPUs with CGP for improved performance in image processing tasks.

Many of these methods use direct pixel input to the evolved program. While originally demonstrated using machine learning benchmarks, MT-CGP [5] offered an improvement to CGP allowing for greater image processing techniques to follow. By using matrix inputs and functions, entire images could be processed using state of the art image processing libraries. A large subset of the OpenCV library was used in Harding et al. [6] for image processing, medical imaging, and object detection in robots.

2.3 Arcade Learning Environment

The ALE offers a related problem to image processing, but also demands reinforcement learning capability, which has not been well studied with CGP. Multiple neuroevolution approaches, including HyperNEAT, and CMA-ES were applied to pixel and object representations of the Atari games in Hausknecht et al. [8]. The performance of the evolved object-based controllers demonstrated the difficulties of using raw pixel input; of the 61 games evaluated, controllers using pixel input performed the best for only 5 games. Deterministic random noise was also given as input and controllers using this input performed the best for 7 games. This demonstrates the capability of methods that learn to perform a sequence of actions unrelated to input from the screen.

HyperNEAT was also used in Hausknecht et al. [7] to show generalization over the Freeway and Asterix games, using a visual processing architecture to automatically find an object representation as inputs for the neural network controller. The ability to generalize over multiple Atari games was further demonstrated in Kelly and Heywood [10], which followed Kelly and Heywood [9]. In this method, tangled problem graphs (TPG) use a feature grid created from the original pixel input. When evolved on single games, the performance on 20 games was impressive, rivaling human performance in 6 games and outperforming neuroevolution. This method generalized over sets of 3 games with little performance decrease.

The ALE is a popular benchmark suite for deep reinforcement learning. Originally demonstrated with deep Q-learning in Mnih et al. [19], the capabilities of deep neural networks to learn action policies based on pixel input was fully demonstrated in Mnih et al. [20]. Finally, an actor-critic model improved upon deep network performance in Mnih et al. [18].

3 METHODS

While there are many examples of CGP use for image processing, these implementations had to be modified for playing Atari games. Most importantly, the input pixels must be processed by evolved programs to determine scalar outputs, requiring the programs to reduce the input space. The methods following were chosen to ensure comparability with other ALE results and to encourage the evolution of competitive but simple programs.

3.1 CGP genotype

In this work, a floating point representation of CGP is used. It has some similarity with a previous floating point representation [3]. In the genome, each node n in C columns is represented by four floats, which are all bound between $[0.0, 1.0]$: x input, y input, function, parameter p.

Function	Description	Arity	Broadcasting
Mathematical			
ADD	$(x+y)/2$	2	Yes
AMINUS	$\|x-y\|/2$	2	Yes
MULT	xy	2	Yes
CMULT	xp_n	1	Yes
INV	$1/x$	1	Yes
ABS	$\|x\|$	1	Yes
SQRT	$\sqrt{\|x\|}$	1	Yes
CPOW	$\|x\|^{p_n+1}$	1	Yes
YPOW	$\|x\|^{\|y\|}$	2	Yes
EXPX	$(e^x-1)/(e^1-1)$	1	Yes
SINX	$\sin x$	1	Yes
SQRTXY	$\sqrt{x^2+y^2}/\sqrt{2}$	2	Yes
ACOS	$(\arccos x)/\pi$	1	Yes
ASIN	$2(\arcsin x)/\pi$	1	Yes
ATAN	$4(\arctan x)/\pi$	1	Yes
Statistical			
STDDEV	$std(\vec{x})$	1	No
SKEW	$skewness(\vec{x})$	1	No
KURTOSIS	$kurtosis(\vec{x})$	1	No
MEAN	$mean(\vec{x})$	1	No
RANGE	$max(\vec{x})-min(\vec{x})-1$	1	No
ROUND	$round(\vec{x})$	1	No
CEIL	$ceil(\vec{x})$	1	No
FLOOR	$floor(\vec{x})$	1	No
MAX1	$max(\vec{x})$	1	No
MIN1	$min(\vec{x})$	1	No
Comparison			
LT	$x<y$	2	Yes
GT	$x>y$	2	Yes
MAX2	$\max(x,y)$	2	Yes
MIN2	$\min(x,y)$	2	Yes

Table 1: A part of the function set used. Many of the mathematical and comparison functions are standard for inclusion in CGP function sets for scalar inputs. Where broadcast is indicated, the function was applied equally to scalar and matrix input, and where it is not, scalar inputs were passed directly to output and only matrix inputs were processed by the function.

The x and y values are used to determine the inputs to n. The function gene is cast to an integer and used to index into the list of available functions, determining f_n. Finally, the parameter is scaled between $[-1.0, 1.0]$ using $p_n = 2p-1$. Parameters are passed to functions, as they are used by some functions. Parameters are also used in this work as weights on the final function, which has been done in other CGP work [11].

Nodes are ordered based on their ordering in the genome. The genome begins with n_{output} nodes which determine the index of the output nodes in the graph, and then all genes for the C program nodes. The first n_{input} nodes correspond to the program inputs and are not evolved; the first node after these will correspond to the first four floating point values after n_{output} in the genome, and the next node will correspond to the next four values, and so on. The number of columns C counts only the program nodes after n_{input}, so, in total, the graph is composed of $N = n_{input} + C$ nodes and is based on $G = n_{output} + 4C$ genes.

When determining the inputs for a node n, the x_n and y_n genes are scaled according to r and then rounded down to determine the index of the connected nodes, xi_n and yi_n. The value r indicates the range over which x_n and y_n operate; when $r = 0$, connections are only possible between the first input node and n, and when $r = 1$, connections are possible over the entire genome.

$$xi_n = \lfloor x_n((1-\frac{n}{N})r + \frac{n}{N})\rfloor$$
$$yi_n = \lfloor y_n((1-\frac{n}{N})r + \frac{n}{N})\rfloor$$

Output genes are also rounded down to determine the indices of the nodes which will give the final program output. Once all genes have been converted into nodes, the active graph is determined. Starting from the output nodes, xi_n and yi_n are used to recursively trace the node inputs back to the final program input. Nodes are marked as active when passed, and nodes which have already been marked active are not followed, allowing for a recursive search over graphs with recurrent connections.

With the proper nodes marked as active, the program can be processed. Due to the recurrent connections, the program must be computed in steps. Each node in the graph has an output value, which is initially set to the scalar 0. At each step, first the output values of the program input nodes are set to the program inputs. Then, the function of each program node is computed, using the outputs from connected nodes of the previous step as inputs.

for $n = 0$ to n_{input} **do**
  out_n = `program_input[n]`
end for
for $n = n_{input}$ to N **do**
  $out_n = p_n f_n(out_{xi_n}, out_{yi_n}, p_n)$
end for

The floating point representation in this work was chosen to simplify the genome and evolution. It allows all genes to be represented as the same type, a float, while still allowing for high precision in the evolution of the parameter gene.

3.2 Evolution

A standard $1+\lambda$ EA is used to evolve the programs. At initialization, a random genome is created using G uniformly distributed values in $[0.0, 1.0]$. This individual is evaluated and is considered the first elite individual. At each generation, λ offspring are generated using genetic mutation. These offspring are each evaluated and, if their fitness is greater than or equal to that of the elite individual, they replace the elite. This process is repeated until n_{eval} individuals have been evaluated; in other words, for $\frac{n_{eval}}{\lambda}$ generations. The stop condition is expressed here as a number of evaluations to make runs comparable during optimization of λ.

The genetic mutation operator randomly selects m_{nodes} of the program node genes and sets them to new random values, again drawn from a uniform random distribution in $[0.0, 1.0]$. The output nodes are mutated according to a different probability; m_{output} of the output genes are randomly set to new values during mutation. When these values have been optimized, they are often found to be distinct. It therefore seems beneficial to include this second parameter for output mutation rate.

Function	Description	Arity	Broadcasting
	List processing		
SPLIT_BEFORE	return all values before $\frac{p_n+1}{2}$ in $\vec{x}$	1	No
SPLIT_AFTER	return all values after $\frac{p_n+1}{2}$ in $\vec{x}$	1	No
RANGE_IN	return the values of $\vec{x}$ in $[\frac{y+1}{2}, \frac{p_n+1}{2}]$	2	No
INDEX_Y	return the value of $\vec{x}$ at $\frac{y+1}{2}$	2	No
INDEX_P	return the value of $\vec{x}$ at $\frac{p_n+1}{2}$	1	No
VECTORIZE	return all values of $\vec{x}$ as a 1D vector	1	No
FIRST	return the first value of $\vec{x}$	1	No
LAST	return the last value of $\vec{x}$	1	No
DIFFERNCES	return the computational derivative of the 1D vector of $\vec{x}$	1	No
AVG_DIFFERENCES	return the mean of the DIFF function	1	No
ROTATE	perform a circular shift on $\vec{x}$ by p_n elements	1	No
REVERSE	reverse $\vec{x}$	1	No
PUSH_BACK	create a new vector with all values of x or $\vec{x}$, then y or $\vec{y}$	2	No
PUSH_BACK	create a new vector with all values of y or $\vec{y}$, then x or $\vec{x}$	2	No
SET	return the scalar value x $\texttt{len}(\vec{y})$ times, or y $\texttt{len}(\vec{x})$	2	No
SUM	return the sum of $\vec{x}$	1	No
TRANSPOSE	return the transpose of $\vec{x}$	1	No
VECFROMDOUBLE	return the 1-element $\vec{x}$ if x is a scalar	1	No
	MISCELLANEOUS		
YWIRE	y	1	No
NOP	x	1	No
CONST	p_n	0	No
CONSTVECTORD	return a matrix of $size(\vec{x})$ with values of p_n	1	No
ZEROS	return a matrix of $size(\vec{x})$ with values of 0	1	No
ONES	return a matrix of $size(\vec{x})$ with values of 1	1	No

Table 2: List processing and other functions in the function set. The choice of many of these functions was inspired by MT-CGP [5].

C	40	m_{nodes}	0.1
r	0.1	m_{output}	0.6
λ	9	n_{eval}	10000

Table 3: CGP parameter values. All parameters except n_{eval} were optimized using irace.

The parameters C, r, λ, m_{nodes}, and m_{output} were optimized using irace [13]. The values used in this experiment are presented in Table 3 and are somewhat standard for CGP. λ is unusually large; normal values are 4 or 5, and the maximum allowed during parameter optimization was 10. The other main parameter setting in CGP is the choice of function set, which is detailed next.

3.3 Mixed Types

In this work, the program inputs are pixel values of the Atari screen and program outputs must be scalar values, representing the preference for a specific action. Intermediate program nodes can therefore receive a mix of matrix and scalar inputs. To handle this, each node's function was overloaded with four possible input types: $(x, y), (x, \vec{y}), (\vec{x}, y), (\vec{x}, \vec{y}$ For some functions, broadcasting was used to apply the same function to the scalar and matrix input types. In other functions, arity made it possible to ignore the type of the y argument. Some functions, however, such as $std(\vec{x})$, require matrix input. In these cases, scalar x input was passed directly to output; in other words, these functions operated as a wire when not receiving matrix input. In other functions, scalar input of either x or y is necessary. In these cases, the average value of matrix input is used. Finally, some functions use inputs to index into matrices; when floating point values are used to index into matrices, they are first multiplied by the number of elements in the matrix and then rounded down.

To account for matrix inputs of different sizes, the minimum of each dimension between the two matrices is taken. This inherently places more import on the earlier values along each dimension than later ones, as the later ones will often be discarded. However, between minimizing the sizes of the two matrices and maximizing them, minimizing was found to be superior. Maximization requires a padding value to fill in smaller dimensions, for which 0, 1, and p_n were used, but the resultant graphs were found to be highly dependent on this padding value.

All functions in the chosen set are designed to operate over the domain $[-1.0, 1.0]$. However, some functions, such as $std(\vec{x})$, return values outside of this domain or are undefined for some values in this domain. Outputs are therefore constrained to $[-1.0, 1.0]$ and NaN and inf values are replaced with 0. This constraining operator is applied element-wise for matrix output. While this appears to limit the utility of certain functions, evolution must choose to use functions in an appropriate case. There have been instances

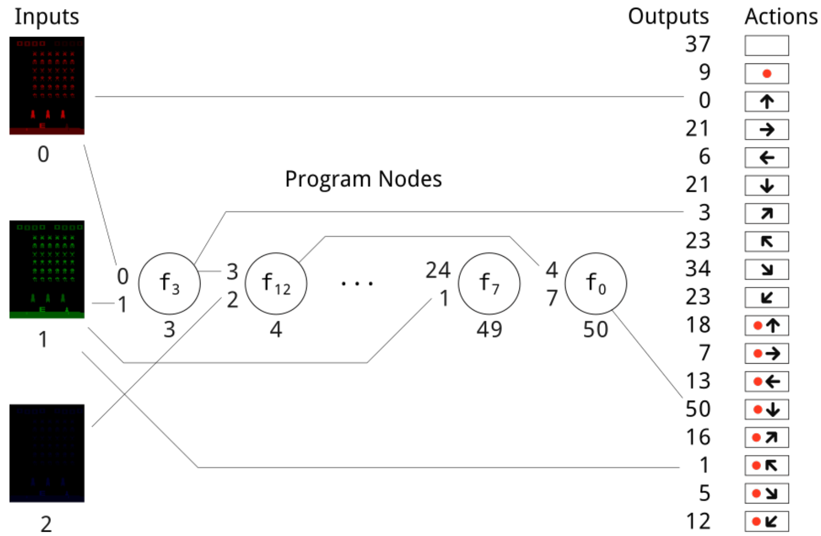

Figure 1: Using CGP to play Atari. Red, green, blue pixel matrices are input to the evolved program, and evolved outputs determine the final controller action. Here, all legal controller actions are represented, but most games only use a subset of possible actions. Actions with a red mark indicate a button press.

of exaptation where evolution has used such functions with out of domain bounds, raceto achieve constant 0.0 or p_n output.

The function set used in this work was designed to be as simple as possible while still allowing for necessary pixel input processing. No image processing library was used, but certain matrix functions allow for pixel input to inform program output actions. The function set used in this work defined in tables Table 1 and Table 2. It is a large function set and it is the intention of future work to find the minimal necessary function set for Atari playing.

To determine the action taken, each node specified by an output gene is examined. For nodes with output in matrix format, the average value is taken, and for nodes with scalar output, the scalar value is taken. These output values are then compared and the maximum value triggers the corresponding action.

3.4 ALE

In the ALE, there are 18 legal actions, corresponding to directional movements of the controller (8), button pressing (1), no action (1), and controller movement while button pressing (8). Not all games use every possible action; some use as few as 4 actions. In this work, outputs of the evolved program correspond only to the possible actions for each game. The output with the highest value is chosen as the controller action.

An important parameter in Atari playing is frame skip [2]. In this work, the same frame skip parameter as in Hausknecht et al. [8], Kelly and Heywood [9] and Mnih et al. [20] is used. Frames are randomly skipped with probability $p_{fskip} = 0.25$ and the previous controller action is replayed. This default value was chosen as the highest value for which human play-testers were unable to detect a delay or control lag [14]. This allows the results from artificial controllers to be directly compared to human performance.

The screen representation used in this work is pixel values separated into red, green, and blue layers. A representation of the full CGP and Atari scheme is included in Figure 1.

CGP parameter optimization was performed on a subset of the full game set consisting of Boxing, Centipede, Demon Attack, Enduro, Freeway, Kung Fu Master, Space Invader, Riverraid, and Pong. These games were chosen to represent a variety of game strategies and input types. Games were played until completion or until reaching 18000 frames, not including skipped frames.

4 RESULTS

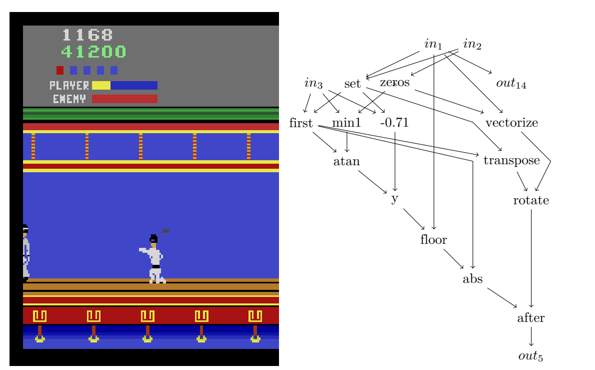

Figure 2: The Kung-Fu Master crouching approach and the functional graph of the player. Outputs which are never activated, and the computational graph leading to them, are omitted for clarity.

By inspecting the resultant functional graphs of an evolved CGP player and observing the node output values during its use, the strategy encoded by the program can be understood. For some of the best performing games for CGP, these strategies can remain incredibly simple. One example is Kung-Fu Master, shown in Figure 2. The strategy, which can receive a score of 57800, is to alternate between the crouching punch action (output 14), and a lateral movement (output 5). The input conditions leading to these actions can be determined through a study of the output program, but output 14 is selected in most cases based simply on the average pixel value of input 1.

While this strategy is difficult to replicate by hand, due to the use of lateral movement, interested readers are encouraged to try simply repeating the crouching punch action on the Stella Atari emulator. The lateral movement allows the Kung-Fu Master to sometimes dodge melee attacks, but the crouching punch is sufficient to wipe out the enemies and dodge half of the bullets. In fact, in comparison to the other attack options (low kick and high kick) it appears optimal due to the reduced exposure from crouching. For the author, employing this strategy by hand achieved a better score than playing the game normally, and the author now uses crouching punches exclusively when attacking in this game.

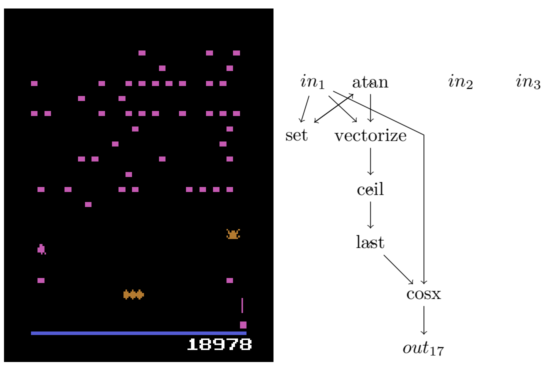

Figure 3: The Centipede player, which only activates output 17, down-left-and-fire. All other outputs are linked to null or constant zero inputs and are not shown.

Other games follow a similar theme. Just as crouching is the safest position in Kung-Fu Master, the bottom left corner is safe from most enemies in Centipede. The graph of an individual from early in evolution, shown in Figure 3, demonstrates this. While this strategy alone receives a high score, it does not use any pixel input. Instead, output 17 is the only active output, and is therefore repeated continuously. This action, down-left-and-fire, navigates the player to the bottom left corner and repeatedly fires on enemies. Further evolved individuals do use input to dodge incoming enemies, but most revert to this basic strategy once the enemy is avoided.

The common link between these simple strategies is that they are, on average, effective. Evolution rewards agents by selecting them based on their overall performance in the game, not based on any individual action. The policy which the agent represents will therefore tend towards actions which, on average, give very good rewards. As can be seen in the case of the Kung-Fu Master, which has different attack types, the best of these is chosen. Crouching punch will minimize damage to the player, maximizing the game's score and therefore the evolutionary fitness. The policy encoded by the program doesn't incorporate other actions because the average reward return for these actions is lower. The safe locations found in these games can also be seen as an average maximum over the entire game space; the players don't move into different positions because those positions represent a higher average risk and therefore a worse evolutionary fitness.

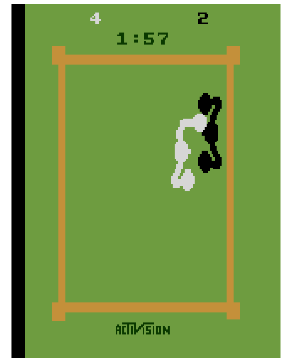

Figure 4: Boxing, a game that uses pixel input to continuously move and take different actions. Here, the CGP player has pinned the Atari player against the ropes by slowly advancing on it with a series of jabs.

Not all CGP agents follow this pattern, however. A counter example is boxing, which pits the agent against an Atari AI in a boxing match. The CGP agent is successful at trapping the Atari player against the ropes, leading to a quick victory, as shown in Figure 4. Doing this requires a responsive program that reacts to the Atari AI sprite, moving and placing punches correctly to back it into a corner. While the corresponding program can be read as a CGP program, it is more complex and performs more input manipulation than the previous examples. Videos of these strategies are included as supplementary material.

Finally, in Table 4, CGP is compared to other state of the art results. CGP performs better than all other compared artificial agents on 8 games, and is tied for best with HyperNEAT for one game. On a number of games where CGP does not perform the best, it still achieves competitive scores to other methods. However, there are certain games where CGP does not perform well. There appears to be a degree of similarity between the evolved agents (TPG [9], HyperNEAT [8]). There is also a degree of similarity between the and the deep learning agents (Double [27], Dueling [29], Prioritized [23], and A3C [18]). The authors attribute this similarity to the creation of a policy model for deep learning agents, which is trained over a number of frames, as opposed to a player which is evaluated over an entire episode, as is the case for the evolutionary methods. This difference is discussed further in the next section.

5 DISCUSSION

Taking all of the scores achieved by CGP into account, the capability of CGP to evolve competitive Atari agents is clear. In this work, we have demonstrated how pixel input can be processed by an evolved program to achieve, on certain games, human level results. Using a function set based on list processing, mathematics, and statistics, the pixel input can be properly processed to inform a policy which makes intelligent game decisions.

The simplicity of some of the resultant programs, however, can be disconcerting, even in the face of their impressive results. Agents like a Kung-Fu Master that repeatedly crouches and punches, or a Centipede blaster that hides in the corner and fires on every frame, do not seem as if they have learned about the game. Even worse, some of these strategies do not use their pixel input to inform their final strategies, a point that was also noted in Hausknecht et al. [8].

This is a clear demonstration of a key difficulty in evolutionary reinforcement learning. By using the reward over the entire sequence as evolutionary fitness, complex policies can be overtaken by simple polices that receive a higher average reward in evolution. While CGP showed its capability to creating complex policies, on certain games, there are more beneficial simple strategies which dominate evolution. These simple strategies create local optima which can deceive evolution.

In future work, the authors intend to use novelty metrics to encourage a variety of policies. Novelty metrics have shown the ability to aid evolution in escaping local optima. [12] Furthermore, deep reinforcement learning has shown that certain frames can be more important in forming the policy than others [23]. Similarly, evolutionary fitness could be constrained to reward from certain frames or actions and not others. Finally, reducing the frame count in evolution could also decrease the computational load of evolving on the Atari set, as the same frame, action pairs are often computed multiple times by similar individuals.

A more thorough comparison between methods on the Atari games is also necessary as future work. Deep learning methods use frame counts, instead of episode counts, to mark the training experience of a model. While the use of frame skipping is consistent between all compared works, the random seeding of environments and resulting statistical comparisons are difficult. The most available comparison baseline is with published results, but these are often averages or sometimes single episode scores. Finally, a thorough computational performance comparison is necessary. The authors believe that CGP can achieve the reported results much faster than other methods using comparable hardware, as the main computational cost is performing the Atari games, but a more thorough analysis is necessary.

In conclusion, this work represents a first use of CGP in the Atari domain, and the first case of a GP method using pure pixel input. CGP was best among or competitive with other artificial agents while offering agents that are far less complex and can be read as a program. It was also competitive with human results on a number of games and gives insight into better human playing strategies. While there are many avenues for improvement, this work demonstrates that CGP is competitive in the Atari domain.

ACKNOWLEDGMENTS

This work is supported by ANR-11-LABX-0040-CIMI, within programme ANR-11-IDEX-0002-02. This work was performed using HPC resources from CALMIP (Grant P16043).

REFERENCES

[1] 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.

[2] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. 2000. Frame skip is a powerful parameter for learning to play atari. *Space* 1600 (2000), 1800.

[3] Janet Clegg, James Alfred Walker, and Julian Frances Miller. 2007. A new crossover technique for Cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*. ACM Press, New York, New York, USA, 1580.

[4] Simon Harding. 2008. Evolution of image filters on graphics processor units using cartesian genetic programming. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 1921–1928.

[5] Simon Harding, Vincent Graziano, Jürgen Leitner, and Jürgen Schmidhuber. 2012. MT-CGP: Mixed Type Cartesian Genetic Programming. *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12* (2012), 751.

[6] Simon Harding, Jürgen Leitner, and Juergen Schmidhuber. 2013. Cartesian genetic programming for image processing. In *Genetic programming theory and practice X*. Springer, 31–44.

[7] Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, and Peter Stone. 2012. HyperNEAT-GGP: A HyperNEAT-based Atari general game player. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 217–224.

[8] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. 2014. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2014), 355–366.

[9] 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.

[10] Stephen Kelly and Malcolm I Heywood. 2017. Multi-task learning in Atari video games with Emergent Tangled Program Graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 195–202.

[11] Karlo Knezevic, Stjepan Picek, and Julian F. Miller. 2017. Amplitude-oriented Mixed-type CGP Classification. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. 1415–1418.

[12] Joel Lehman and Kenneth O Stanley. 2008. Exploiting open-endedness to solve problems through the search for novelty.. In *ALIFE*. 329–336.

[13] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43–58.

[14] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. 2017. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *CoRR* abs/1709.06009 (2017).

[15] J. F. Miller and S. L. Smith. 2006. Redundancy and computational efficiency in Cartesian Genetic Programming. *IEEE Trans. on Evolutionary Computation* 10, 2 (2006), 167–174.

[16] J. F. Miller and P. Thomson. 2000. Cartesian Genetic Programming. In *Proc. European Conf. on Genetic Programming (LNCS)*, Vol. 10802. 121–132.

[17] Julian F Miller and Peter Thomson. 2011. *Cartesian Genetic Programming*. Natural Computing Series, Vol. 1802. Springer Berlin Heidelberg, Berlin, Heidelberg. 121–132 pages. arXiv:arXiv:1011.1669v3

[18] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[21] Héctor A Montes and Jeremy L Wyatt. 2003. Cartesian Genetic Programming for Image Processing Tasks.. In *Neural Networks and Computational Intelligence*. 185–190.

[22] Paulo Cesar Donizeti Paris, Emerson Carlos Pedrino, and MC Nicoletti. 2015. Automatic learning of image filters using Cartesian genetic programming. *Integrated Computer-Aided Engineering* 22, 2 (2015), 135–151.

[23] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[24] Lukas Sekanina, Simon L Harding, Wolfgang Banzhaf, and Taras Kowaliw. 2011. Image processing and CGP. In *Cartesian Genetic Programming*. Springer, 181–215.

[25] Stephen L Smith, Stefan Leggett, and Andrew M Tyrrell. 2005. An implicit context representation for evolving image processing filters. In *Workshops on Applications of Evolutionary Computation*. Springer, 407–416.

[26] Andrew James Turner and Julian Francis Miller. 2014. Recurrent Cartesian Genetic Programming. In *Proc. Parallel Problem Solving from Nature*. 476–486.

[27] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning.. In *AAAI*, Vol. 16. 2094–2100.

[28] V. K. Vassilev and J. F. Miller. 2000. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In *Proc. Int. Conf. on Evolvable Systems (LNCS)*, Vol. 1801. Springer Verlag, 252–263.

[29] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[30] T. Yu and J. F. Miller. 2001. Neutrality and the Evolvability of Boolean function landscape. In *Proc. European Conference on Genetic Programming (LNCS)*, Vol. 2038. 204–217.

	Human	Double	Dueling	Prioritized	A3C FF	A3C LSTM	TPG	HyperNEAT	CGP
Alien	6875	1033.4	1486.5	900.5	518.4	945.3	**3382.7**	1586	1978 (± 268)
Amidar	1676	169.1	172.7	218.4	263.9	173	**398.4**	184.4	199 (± 1)
Assault	1496	6060.8	3994.8	7748.5	5474.9	**14497.9**	2400	912.6	890.4 (± 255)
Asterix	8503	16837	15840	**31907.5**	22140.5	17244.5		2340	1880 (± 57)
Asteroids	13157	1193.2	2035.4	1654	4474.5	5093.1	3050.7	1694	**9412** (± 1818)
Atlantis	29028	319688	445360	593642	**911091**	875822		61260	99240 (± 5864)
Bank Heist	734.4	886	**1129.3**	816.8	970.1	932.8	1051	214	148 (± 18)
Battle Zone	3800	24740	31320	29100	12950	20760	**47233.4**	36200	34200 (± 5848)
Beam Rider	5775	17417.2	14591.3	**26172.7**	22707.9	24622.2		1412.8	1341.6 (± 21)
Berzerk		1011.1	910.6	1165.6	817.9	862.2		**1394**	1138 (± 185)
Bowling	154.8	69.6	65.7	65.8	35.1	41.8	**223.7**	135.8	85.8 (± 15)
Boxing	4.3	73.5	**77.3**	68.6	59.8	37.3		16.4	38.4 (± 4)
Breakout	31.8	368.9	411.6	371.6	681.9	**766.8**		2.8	13.2 (± 2)
Centipede	11963	3853.5	4881	3421.9	3755.8	1997	**34731.7**	25275.2	24708 (± 2577)
Chopper Comman	9882	3495	3784	6604	7021	**10150**	7010	3960	3580 (± 179)
Crazy Climber	35411	113782	124566	131086	112646	**138518**		0	12900 (± 6620)
Defender		27510	33996	21093.5	56533	233021.5		14620	**993010** (± 2739)
Demon Attack	3401	69803.4	56322.8	73185.8	113308.4	**115201.9**		3590	2387 (± 558)
Double Dunk	-15.5	-0.3	-0.8	**2.7**	-0.1	0.1	2	2	2 (± 0)
Enduro	309.6	1216.6	**2077.4**	1884.4	-82.5	-82.5		93.6	56.8 (± 7)
Fishing Derby	5.5	3.2	-4.1	9.2	18.8	**22.6**		-49.8	-51 (± 10)
Freeway	29.6	28.8	0.2	27.9	0.1	0.1		**29**	28.2 (± 0)
Frostbite	4335	1448.1	2332.4	2930.2	190.5	197.6	**8144.4**	2260	782 (± 795)
Gopher	2321	15253	20051.4	**57783.8**	10022.8	17106.8		364	1696 (± 308)
Gravitar	2672	200.5	297	218	303.5	320	786.7	370	**2350** (± 50)
H.E.R.O.	25763	14892.5	15207.9	20506.4	**32464.1**	28889.5		5090	2974 (± 9)
Ice Hockey	0.9	-2.5	-1.3	-1	-2.8	-1.7		**10.6**	4 (± 0)
James Bond	406.7	573	835.5	3511.5	541	613		5660	**6130** (± 3183)
Kangaroo	3035	**11204**	10334	10241	94	125		800	1400 (± 0)
Krull	2395	6796.1	8051.6	7406.5	5560	5911.4		**12601.4**	9086.8 (± 1328)
Kung-Fu Master	22736	30207	24288	31244	28819	40835		7720	**57400** (± 1364)
Montezumas Revenge	4367	42	22	13	**67**	41	0	0	0 (± 0)
Ms. Pacman	15693	1241.3	2250.6	1824.6	653.7	850.7	**5156**	3408	2568 (± 724)
Name This Game	4076	8960.3	11185.1	11836.1	10476.1	**12093.7**		6742	3696 (± 445)
Phoenix		12366.5	20410.5	27430.1	52894.1	**74786.7**		1762	7520 (± 1060)
Pit Fall		-186.7	-46.9	-14.8	-78.5	-135.7		**0**	**0** (± 0)
Pong	9.3	19.1	18.8	18.9	5.6	10.7		-17.4	**20** (± 0)
Private Eye	69571	-575.5	292.6	179	206.9	421.1	**15028.3**	10747.4	12702.2 (± 4337)
Q*Bert	13455	11020.8	14175.8	11277	15148.8	**21307.5**		695	770 (± 94)
River Raid	13513	10838.4	16569.4	**18184.4**	12201.8	6591.9	3884.7	2616	2914 (± 90)
Road Runner	7845	43156	58549	56990	34216	**73949**		3220	8960 (± 2255)
Robotank	11.9	59.1	**62**	55.4	32.8	2.6		43.8	24.2 (± 1)
Seaquest	20182	14498	37361.6	**39096.7**	2355.4	1326.1	1368	716	724 (± 26)
Skiing		-11490.4	-11928	-10852.8	-10911.1	-14863.8		**-7983.6**	-9011 (± 0)
Solaris		810	1768.4	2238.2	1956	1936.4		160	**8324** (± 2250)
Space Invaders	1652	2628.7	5993.1	9063	15730.5	**23846**		1251	1001 (± 25)
Star Gunner	10250	58365	90804	51959	138218	**164766**		2720	2320 (± 303)
Tennis	-8.9	-7.8	**4.4**	-2	-6.3	-6.4		0	0 (± 0)
Time Pilot	5925	6608	6601	7448	12679	**27202**		7340	12040 (± 358)
Tutankham	167.6	92.2	48	33.6	**156.3**	144.2		23.6	0 (± 0)
Up n Down	9082	19086.9	24759.2	29443.7	74705.7	**105728.7**		43734	14524 (± 5198)
Venture	1188	21	200	244	23	25	**576.7**	0	0 (± 0)
Video Pinball	17298	367823.7	110976.2	374886.9	331628.1	**470310.5**		0	33752.4 (± 6909)
Wizard of Wor	4757	6201	7054	7451	17244	**18082**	5196.7	3360	3820 (± 614)
Yars Revenge		6270.6	25976.5	5965.1	7157.5	5615.5		24096.4	**28838.2** (± 2903)
Zaxxon	9173	8593	10164	9501	**24622**	23519	6233.4	3000	2980 (± 879)

Table 4: Average CGP scores from five $1 + \lambda$ evolutionary runs, compared to state of the art methods. Bold indicates the best score from an artificial player. Reported methods Double [27], Dueling [29], Prioritized [23], A3C [18], TPG [9], and HyperNEAT [8] were chosen based on use of pixel input. Professional human game tester scores are from Mnih et al. [19].