Evolving CGP programs to play Atari games

Andrea Tupini (MAT. 194578) - andrea.tupini@studenti.unitn.it
Johann Seltmann (MAT. 203386) - johann.seltmann@studenti.unitn.it

I. Introduction

PLAYING games has long been a popular way to test the performance of automatic AI. This field has lately been dominated by Generative Adversarial Networks [1], but other techniques have proven quite effective as well. In this paper, we'll apply a specific evolutionary technique called Cartesian Genetic Programming (CGP) [2] which will let us evolve programs to play Atari games. The performance of our method will then be compared with the state of the art in automatic game playing.

CGP has long been used for image related tasks, specifically in the field of image recognition and image filtering [3] with quite good results. Our work is also heavily influenced by [4] in which they also apply CGP techniques for playing Atari games, but they don't consider functions from the OpenCV [5] library nor do they use crossover in the EA.

For doing the experiments we use the OpenAI Gym [6], as an interface to the ALE [7] environment. The ALE environment basically provides us with an Atari emulator that we can access pragmatically. Each frame we get an observation which is just an RGB image (a 3D matrix), which we then pass over to our CGP program, which will tell us the best action to perform. This action is then performed for 3 to 6 frames (randomly) to simulate the response time of a human. The ALE environment also offers us the *score* that a specific CGP program has obtained.

The evolutionary algorithm (and supporting methods) was set up by using the *inspyred* [8] framework. We defined our custom genetic operators (mutation, crossover, and individual generation).

In this report, we'll start by giving a small explanation of CGP and it's structure, then we'll see the functions that are part of our function set. We'll proceed to explain how we evolve each CGP program, and the EA parameters we use. After this, we'll see the results we obtained as part of the experiments performed, and finally, we'll talk about the difficulties we experienced and conclusions.

II. Methodology

A. Cartesian Genetic Programming

CGP is a form of genetic programming in which programs are represented by directed graphs. The nodes of this graph can be of 3 types:

Input nodes these are the nodes which receive the input to a CGP program

Inner nodes these are the nodes which process input and produce an output

Output nodes these are the nodes which receive the output from inner nodes

Each node, except for the input nodes, have inputs coming from 2 other nodes in the graph. Inner nodes also have a function associated with them and a parameter which can be optionally used by the function. The output on any non-output node can be used an unlimited number of times, meaning that previous results can be reused.

For our specific problem, we always have 3 input nodes (one for each channel of the input RGB image), followed by a maximum of 40 inner nodes. The number of output nodes reflects the number of possible actions that can be performed in a given Atari game, so this number changes from game to game. The maximum possible number of actions that can be done in Atari is 16, so this is the upper limit for output nodes.

We used a slightly different form of CGP, called RCGP (Recurrent Cartesian Genetic Programming) [9] which allows cycles in the graph. This was done so that the internal state of the program can be passed from one evaluation to the next. The only difference between CGP and RCGP is that we allow an inner node to take input from a node nearer than itself to the output, or take input from itself.

CGP programs can be easily represented as an array of real numbers of size:

$$[numberInputNodes+ \\ maxNumberInnerNodes+ \\ numberOutputNodes] * 4$$

Where each group of 4 numbers represents a node in this way:

$$[indexInput1, \\ indexInput2, \\ indexFunction, \\ parameter]$$

The *input indexes* are used to identify the position of the input node in the CGP array, the *function index* works the same way: we use it to index the function in the array of all available functions (the function set). In the case that a specific function were to require a parameter it can use the *parameter* component of the node.

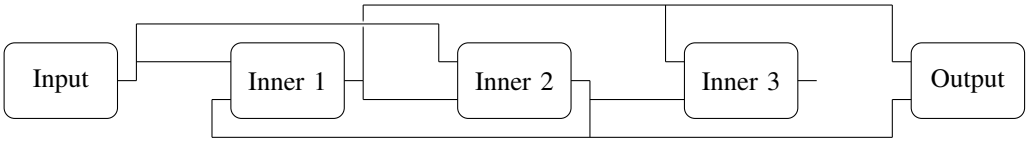


Fig. 1. A small CGP program. Each of the inner nodes has two inputs and one output. There is a recurrent connection from Inner 2 to Inner 1. Inner 3 is an inactive node since its output is not used. This graphic only shows one row of nodes since our program only consists of one row, but in general, CGP programs can contain multiple rows so that the nodes are ordered on a grid.

All of the node components have a minimum value of 0 and a maximum of 1. The index parameters are scaled by the respective array they are indexing and then cast to an integer. The *parameter* is scaled to be in the range $[-1, 1]$. The scaling of the input indexes is done by multiplying them by the size of $numInputs + maxNumInnerNodes$ because those are the *legal* nodes from which they can get an input (it is not possible to get inputs from the output nodes). This real-valued representation was proposed in [10] and it was shown to aid in evolution.

When evaluating a CGP program, only a small subset of the maximum number of inner nodes will actually be evaluated. These will be the ones that are connected to the output nodes, and are called *active nodes*. The other nodes which are not connected to the output are not evaluated and are called *inactive* or *junk* nodes. Even though these nodes are not used in the program they have been shown to aid in the evolutionary process [11]. Those CGP programs which don't have any node taking input from the *input nodes* are classified as invalid and are not evaluated.

When starting the evaluation of a GCP program we first identify which are the active nodes. Then, at each iteration, we set the value of the input nodes to their respective input, and evaluate each of the subsequent nodes in a sequential fashion. If a node takes input from a node which hasn't been evaluated yet then that input will be 0, if instead it was evaluated in the previous iteration then this last return value will be the input.

After each evaluation, the final value of the output nodes is obtained as the sum of the averages of their inputs. The next action to be performed is taken to be that represented by the output node with the highest value.

B. Function set

As with any GP problem, the selection of the function set must be done with great care since they define what is possible for the programs to do. Since making a study of all possible functions and their performance is very time consuming, we're using a function set which has shown good results for other research [3], [4], [12].

The function set consist of two main subsets:

image processing functions These functions are the same image processing functions as in [3] and mostly use the OpenCV library [5].

other functions These functions are the same as in [4] and [12]. They include mathematical, statistical, comparison, and list processing functions.

The value of all functions, except for the statistical functions, is clipped to be in the domain $[-1, 1]$. The RGB input matrix is also rescaled to be in this range, so that we don't lose any information by clipping.

After evaluating a function, any NaN values in the returned data are replaced with 0.

As with [12], we designed all our functions so that they all accept multiple input data types (*float* and *list*), and so that they all accept two inputs. If a function is a 1-arity function, then it is only applied to the first input and the second is ignored. If the function only makes sense when applied to a specific data type (for example a list processing function), then if another data type is passed it will just act a *wire*: it will pass the input directly to the output.

The complete function set can be seen in Table II and Table III.

C. Evolution

For the EA, we use a simple Genetic Algorithm. We use *rank selection* to determine which individuals will become parents of the next generation. We use *generational replacement* with 3 elite individuals, as the strategy with which we update our population across generations. And our termination condition is just a maximum number of generations. Following is a table which presents the main GA parameters we used:

Parameter	Value
Population Size	9
Max Generations	120
Number of individuals selected for crossover	6
Number of elite individuals	3
Mutation Rate	0.7
Mutation Rate Output Nodes	0.1
Mutation Rate Inner Nodes	0.6

Below we'll see more in depth how we defined the different genetic operations.

1) Generation of new individuals: The genome for all individuals is just an array of real-valued numbers with values in the range $[0, 1]$. The size of each genome is $[maxNumberInnerNodes + numOutputNodes] * 4$. Input nodes are not considered as part of the genome since they're not evolved. The generation of each new individual is just sampling a series of numbers from a random distribution defined in $[0, 1]$.

2) Mutation: For mutation we use a *general mutation rate* which is the probability that a given individual gets mutated. Besides this, we also have specific mutation rates for the inner nodes and for the output nodes. In [4] they optimize these probabilities and it was found that different mutation probabilities for inner and output nodes perform best. We use the same probabilities they used.

The mutation process is actually the same for inner and output nodes, only the probabilities change. So we'll consider the mutation process from a generic point of view: it consists of randomly picking $round(numNodes * mutProbability)$ nodes from the set of nodes, and *re-sampling* all values for those nodes from a $[0, 1]$ uniform distribution.

3) Crossover: The crossover function we're using is the same as the one proposed in [10]. For two parent individuals, $p1$ and $p2$, the genes of the offspring are calculated as:

$$o_i = r * p1_i + (1 - r) * p2_i$$

where $r \sim N(0, 1)$. In the crossover process we generate two offspring, each with a different r.

III. Results

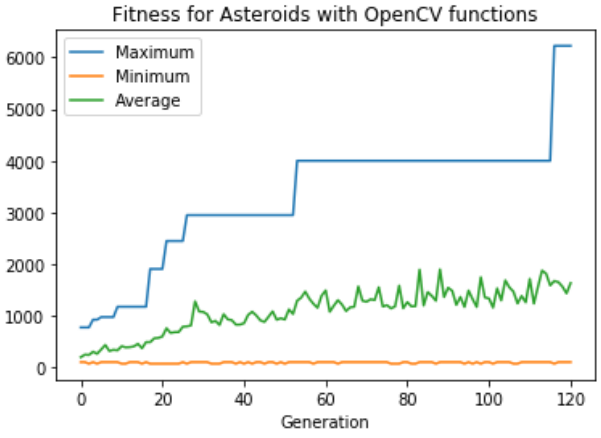
We chose to evaluate the CGP program on four different Atari games. These are:

1) Asteroids
2) Boxing
3) Ms. Pacman
4) Space Invader

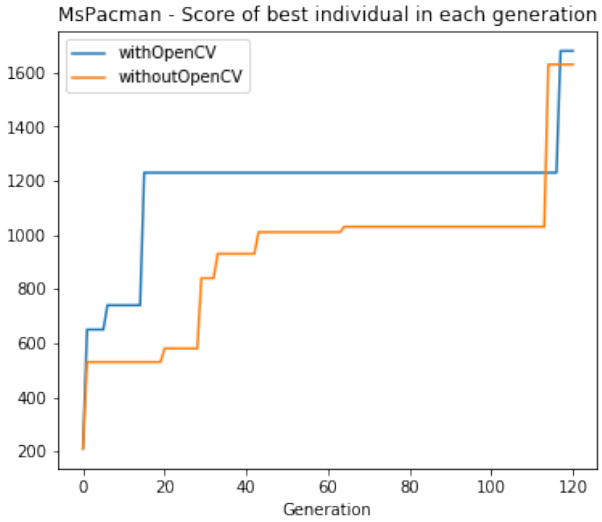
We did three different runs of each game, each starting with a different seed for the random number generator, and then used the agglomerate of these results for our analysis. For the sake of reproducibility, the seeds we used were: 84, 111, and 231.

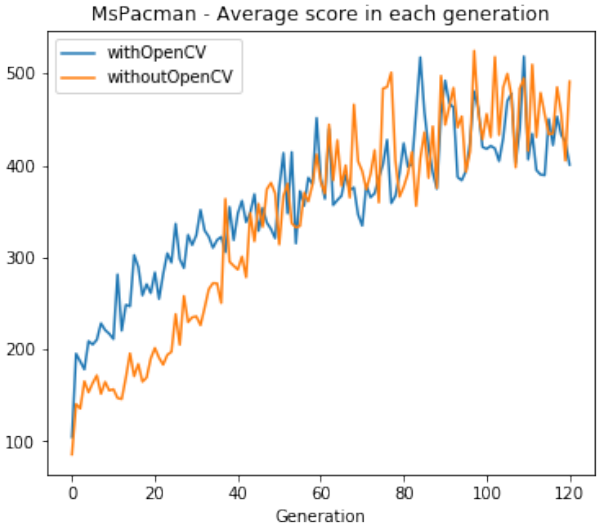
After running the EA for 120 generations, we were given the best performing individual, as well as a log of all individuals in all generations. Figure 2 shows the graph representation of an example CGP generated for the *Boxing* game. We can see how the nodes are ordered: input nodes at the top right, output nodes at the bottom of the image, and the direction of the arrows indicate the flow of the data (from input to output nodes).

Below we can see a graph for how the fitness over different generations of the game *Asteroids* behaves. Note that the *minimum* doesn't really seem to increase much, but the *mean* and *maximum* do increase steadily. Remember that we keep a pool of elite individuals, so that's why the *maximum* appears like steps.



Besides the basic performance of the algorithm, we also wanted to evaluate if OpenCV functions increased, decreased, or didn't affect at all the performance of the algorithm. So we evaluated the games both with the OpenCV functions as part of the function set and without these functions.





The graphs for the other games we used to test the algorithm can be seen in Figure 3.

As we can see, it seems that actually having the OpenCV functions as part of the function set doesn't help the overall performance of the algorithm. For another game (Space Invaders) they slightly decrease the performance. While they lead to a slight increase for Boxing.

There might be many reasons why this is happening, but one of them could be that OpenCV functions only get applied to 2D matrices (something that resembles an image), any other kind of input is just passed through untouched. So we might have a CGP program that has many OpenCV functions that just act as wires. It is also possible that the increased search space leads to a slower evolution and that the performance would improve over time. Another reason might be that the Atari output images are of a relatively small size (210 * 160 pixels in three color channels), on which the image processing operations might not be as useful as on bigger images.

The table below shows the best fitness obtained by both evolving with OpenCV functions and without.

Game	With OpenCV	Without OpenCV
Asteroids	6220	**7060**
Boxing	**39**	19
Ms. Pacman	**1680**	1630
Space Invader	695	**755**

We compare our results with the state of the art results obtained by the following methods: A3C [13], HyperNEAT [14], and also with those obtained by [4] in which they also use CGP, but they didn't use crossover nor OpenCV functions as part of the function set. We also include professional human player [15] scores in our comparison. Table I shows said comparison of results. Note that the results obtained by CGP in [4] are, in most cases, better than the ones we obtained, but our method still performs very well when compared with the other results.

All the code we used can be found in its own GitHub repository[1].

IV. Difficulties in the project

One of the main difficulties we encountered was that of developing and correctly debugging the functions is the function set. Many issues could be caught by simple tests, but others only came up during evaluation. In many occasions we needed to stop evaluation, fix the issue, and then start again, only to find another issue. This process was a bit time consuming. By making better tests many of these errors could have been caught early on.

Another difficulty we experienced, which is still related to the function set, was what to do with those functions that accept only one input (they take into consideration only the first input). The question arose if we should have symmetric versions of these functions where only the second input is evaluated. It is true that by using only the one-input versions the EA should still be able to organize the inputs in such a way that the *correct one* is the one that gets passed first and evaluated. On the other hand, it is also a good idea to have the symmetric versions of these functions so that the EA has more possibilities. However, adding the symmetric version could also be considered as making the search space more complex. In the end, we opted for keeping the symmetric versions.

The crossover operation we use might have been a bit too disruptive, meaning that offspring are not really that similar to their parents. We could have used a more standard crossover method, like single point crossover, to obtain more similar offspring. But seeing that [10] obtained good results using this crossover method we decided to stick with it.

V. Lessons learned

Working on this project gave us chance to apply an EA to a real-world problem, and discover how to set it up and configure the parameters so that it has the best performance possible. Also, investigating about the different strategies available (i.e., for crossover, replacement of generations, and individual selection for reproduction) helped us understand more about how these strategies actually work and led us to appreciate the amount of flexibility that EA have.

In the beginning, we struggled a bit with the *project management* perspective, mainly the division of tasks and organizing the dependencies between them. But we managed to come up with an effective solution regarding time and performance, by tracking what needed to be done in the project's issue tracker (in GitHub).

VI. Contributions by group members

As with any project of this nature, both of us ended up working on every aspect of the project. But each one centered their efforts on the following points:

Andrea:

- implementation of evaluation, crossover, mutation
- integration of inspyred, OpenAI gym, and our own code
- drawing of CGP programs
- parts of the function set

Johann:

- implementation of CGP cells and program
- parts of the function set

Appendix A
Tables showing the function set

Table II and Table III show the function set. $inp1$ and $inp2$ represent the input values to the cell, par the evolved parameter of the cell. For functions which only apply to $inp1$ there are symmetrical functions using $inp2$, which we omitted in the table.

Appendix B
Result graphs for other games

This section contains the graphs for the games "Boxing", "Asteroids", and "Space Invaders". For Boxing and Space Invaders the evolution for the random seed 111 was ended after 65 and 51 generations, respectively due to time constraints.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[2] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Genetic Programming*, R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 121–132.

[3] S. Harding, J. Leitner, and J. Schmidhuber, *Cartesian Genetic Programming for Image Processing (CGP-IP)*, 03 2013, pp. 31–44.

[4] D. G Wilson, S. Cussat-Blanc, H. Luga, and J. Miller, "Evolving simple programs for playing atari games," 07 2018, pp. 229–236.

[5] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, jun 2013.

[8] A. Garrett, "inspyred (version 1.0.1) [software]. inspired intelligence," https://github.com/aarongarrett/inspyred, 2012.

[1] https://github.com/sejo95/EvolutionaryGaming

Game	Human	A3C FF	A3C LSTM	HyperNeat	CGP	Our-CGP
Asteroids	13157	4474.5	5093.1	1694	**9412**	7060
Boxing	4.3	**59.8**	37.3	16.4	38.4	39
Ms. Pacman	15693	653.7	850.7	**3408**	2568	1680
Space Invader	1652	15730.5	**23846**	1251	1001	755

TABLE I

COMPARISON OF THE RESULTS OBTAINED BY PLAYING ATARI GAMES USING DIFFERENT METHODS. THE BOLDED RESULT SHOWS THE BEST RESULT OBTAINED BY THE METHOD WHICH PERFORMS BEST (EXCLUDING THE HUMAN ONES). THE *CGP* COLUMN SHOWS THE RESULT OBTAINED BY [4], WHILE THE *Our-CGP* SHOWS THE RESULTS WE OBTAINED.

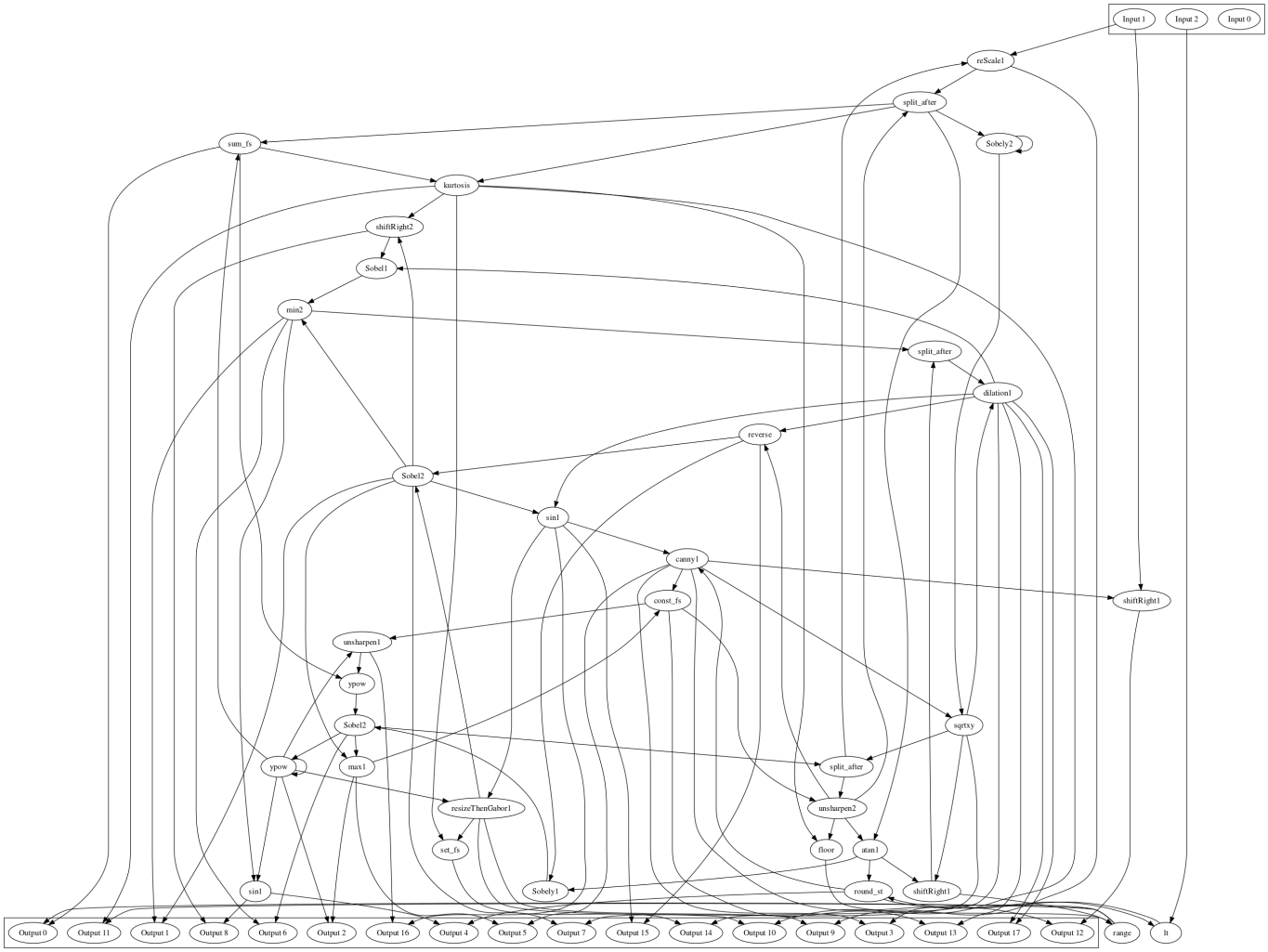


Fig. 2. Example of a CGP program created for the *Boxing* game.

[9] A. J. Turner and J. F. Miller, "Recurrent cartesian genetic programming," in *Parallel Problem Solving from Nature – PPSN XIII*, T. Bartz-Beielstein, J. Branke, B. Filipič, and J. Smith, Eds. Cham: Springer International Publishing, 2014, pp. 476–486.

[10] J. Clegg, J. A. Walker, and J. F. Miller, "A new crossover technique for cartesian genetic programming," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1580–1587. [Online]. Available: http://doi.acm.org/10.1145/1276958.1277276

[11] J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in cartesian genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 2, pp. 167–174, April 2006.

[12] S. Harding, V. Graziano, J. Leitner, and J. Schmidhuber, "Mt-cgp: Mixed type cartesian genetic programming," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 751–758. [Online]. Available: http://doi.acm.org/10.1145/2330163.2330268

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[14] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, 2013. [Online]. Available: http://nn.cs.utexas.edu/?hausknecht:tciaig14

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

Name	description
Mathematical Functions	
add, average, mult, inv, abs, sqrt, sin	applied element-wise if inputs are matrices
aminus	$\|inp1 - inp2\|/2$
cmult	$inp1 * par$
cpow	$\|inp1\|^{par}$
ypow	$\|inp1\|^{\|inp2\|}$
exp	$e^{inp1-1}/(e-1)$
sqrtxy	$\sqrt{inp1^2 + inp2^2}/\sqrt{2}$
acos	$arccos(1)/\pi$
asin	$2 * arcsin(inp1)/\pi$
atan	$4 * arctan(inp1)/\pi$
Statistical Functions	
stddev, skew, kurtosis, mean, round, ceil, floor	
range	$max(inp1) - min(inp1) - 1$
max1	$max(inp1)$
min1	$min(inp1)$
List Functions	
first, last, reverse, transpose	
split_before	all values in inp1 until $(par + 1)/2$
split_after	all values in inp1 until $(par + 1)/2$
range_in	all values in inp1 between $(inp2 + 1)/2$ and $(par + 1)/2$
index_y	value of inp1 at $(inp2 + 1)/2$
index_p	value of inp1 at $(par2 + 1)/2$
vectorize	turn inp1 into a 1D array
differences	gradient of inp1
avg_differences	mean of the differences function
rotate	circular shift of inp1, par determines how far
push_back	new vector with all values inp1 then all values of inp2
push_front	new vector with all values inp2 then all values of inp1
set_fs	return a vector that contains inp1 len(inp2) times or inp2 len(inp1) times)
sum_fs	$sum(inp1)$
vecfromdouble	if inp1 is scalar: return vector containing x
Other Functions	
y_wire	inp2
no_op	inp1
const_fs	par
constvectord	matrix of same shape as inp1 containing par as entries
zeros	matrix of same shape as inp1 containing 0 as all entries
ones	matrix of same shape as inp1 containing 1 as all entries
max2	$max(inp1, inp2)$
min2	$min(inp1, inp2)$

TABLE II
NON-IMAGE-PROCESSING FUNCTIONS FROM [4]

Name	description
sobel	calculate first derivative of inp1 with an extended Sobel operator
sobelx	calculate first derivative of inp1 with an extended Sobel operator over horizontal dimension
sobely	calculate first derivative of inp1 with an extended Sobel operator over vertical dimension
threshold	replace each value in inp1 with 1 if it is greater than inp2, with 0 otherwise
smoothMedian, smoothBilateral, gabor, laplace, canny	Apply the respective filter to inp1.
smoothBilateral	Blur inp1 using a bilateral filter.
smoothBlur	Blur inp1 using a normalized box filter.
unsharpen	Apply the unsharpen operation to inp1.
shiftLeft, shiftRight, shiftUp, shiftDown	Shift inp1 circularly into the respective direction. par determines, how far.
shift	shiftLeft followed by shiftUp
erosion, dilation	
reScale	Downscale inp1 by par, then upscale again.
resizeThenGabor	Downscale inp1, then apply a Gabor filter.
localNormalize	Apply local normalization to inp1.

TABLE III
IMAGE PROCESSING FUNCTIONS FROM [3]

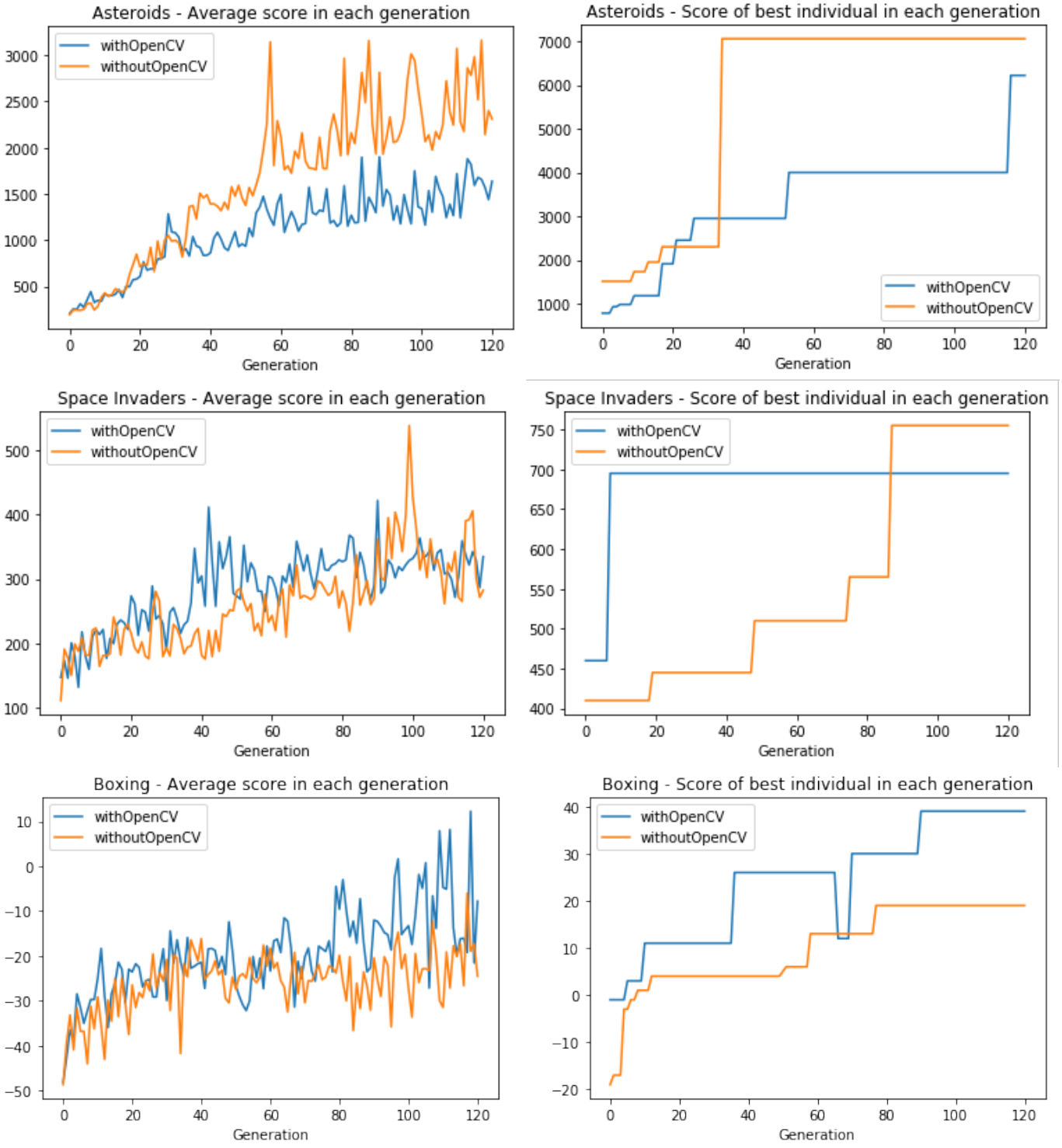


Fig. 3. Comparison of results between evolving with OpenCV functions and without.