Super Mario Evolution

Julian Togelius, Sergey Karakovskiy, Jan Koutník and Jürgen Schmidhuber

Abstract— **We introduce a new reinforcement learning benchmark based on the classic platform game *Super Mario Bros*. The benchmark has a high-dimensional input space, and achieving a good score requires sophisticated and varied strategies. However, it has tunable difficulty, and at the lowest difficulty setting decent score can be achieved using rudimentary strategies and a small fraction of the input space. To investigate the properties of the benchmark, we evolve neural network-based controllers using different network architectures and input spaces. We show that it is relatively easy to learn basic strategies capable of clearing individual levels of low difficulty, but that these controllers have problems with generalization to unseen levels and with taking larger parts of the input space into account. A number of directions worth exploring for learning better-performing strategies are discussed.**

Keywords: Platform games, Super Mario Bros, neuroevolution, input representation

I. WHY?

Why would we want to use evolutionary or other reinforcement learning algorithms to learn to play a video game?

One good reason is that we want to see what our learning algorithms and function representations are capable of. Every game requires a somewhat different set of skills to play, and poses a somewhat different learning challenge. Games of different genres often have radically different gameplay, meaning that a different set of skills has to be learned to a different level of proficiency and in a different order. The first goal when trying to automatically learn to play a game is to show that it can be done.

Video games are in many ways ideal testbeds for learning algorithms. The fact that people play these games mean that the skills involved have some relevance to the larger problem of artificial intelligence, as they are skills that humans possess. Good video games also possess a smooth, long learning curve, making them suitable for continual learning by humans and algorithms alike [1].

A related reason is that we would like to compare the performance of different learning algorithms and function representations. A large number of algorithms are capable of solving at least some classes of reinforcement learning problems. However, their relative effectiveness differ widely, and the differences are not always in line with theoretical predictions [2]. To accurately characterize the capabilities of different algorithms we need a wide range of testbed problems, which are easily reproducible (so that different researchers can test their algorithms on the very same problem) and which preferably should have some relevance to real-life problems. We would like to have a collection of problems that cover the multidimensional space formed by the dimensions along which reinforcement learning problems can vary as completely as possible. It seems likely that video games can form the basis of many of these parametrisable testbed problems, especially those on the more complex end of the scales: continuous, high-dimensional state spaces with partial observability yet high-dimensional observations, and perhaps most important of all, requiring a the execution of sequences of different behaviours.

But there is yet another reason, which could be just as important: the development of better adaptation mechanisms for games. While there is limited demand in the games industry for higher-performing opponents in most game genres, there is a demand for more interesting NPCs (opponents, allies, sidekicks etc.), for better ways of adapting the game to the player, and for automatically generating game content. Recently proposed methods for meeting these demands assume that there is already an RL algorithm in place capable of learning to play the particular game that is being adapted [3], [4], [5] and/or models of player experience [6].

These reasons have motivated researchers to apply RL algorithms (most commonly phylogenetic methods such as evolutionary algorithms) to successfully learn to play a large variety of video games from many different game genres. These include arcade games such as *Pac-Man* [7] and *X-pilot* [8], first-person shooter games such as *Quake II* [9] and *Unreal Tournament* [10], varieties of racing games [11], [12] and fighting games [13].

So, given this list of titles and genres, why learn to play yet another game of another genre? For the simple reason that it is not represented in the list above. Each game type presents new challenges in terms of atomic behaviours and their sequencing and coordination, input and output representation, and control generalization.

This paper investigates the evolutionary reinforcement learning of successful strategies/controllers (we will use these words interchangeably) for *Super Mario Bros*, the platform game *par excellence*. We are not aware of any previous attempts at learning to automatically play a platform game.

When a video game-based benchmark has been devised, it's important that the source code and an easy to use interface is released on the Internet so that other researchers can test their own algorithms without going through hassle or re-implementing or re-interfacing the code, and ensuring that the comparisons remain valid. A particularly good way to do this is to organize a competition around the benchmark, where the competitors learn or otherwise develop controllers that play the game as well as possible. This has previously been done for several of the games mentioned above, in-

JT is with the IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark. SK, JK and JS are with IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland. Emails: {julian, sergey, hkou, juergen}@idsia.ch

978-1-4244-4815-9/09/$25.00 ©2009 IEEE

Fig. 1. Infinite Mario Bros.

cluding our own simulated car racing competitions [12]. The benchmark developed for this paper is therefore also used for a competition run in conjunction with international conferences on CI and games, and complete source code is downloadable from the competition web page [1].

II. What?

The game studied in this paper is a modified version of Markus Persson's *Infinite Mario Bros* (see Figure 1) which is a public domain clone of Nintendo's classic platform game *Super Mario Bros*. The original Infinite Mario Bros is playable on the web, where Java source code is also available[2].

The gameplay in Super Mario Bros consists in moving the player-controlled character, Mario, through two-dimensional levels, which are viewed sideways. Mario can walk and run to the right and left, jump, and (depending on which state he is in) shoot fireballs. Gravity acts on Mario, making it necessary to jump over holes to get past them. Mario can be in one of three states: Small (at the beginning of a game), Big (can crush some objects by jumping into them from below), and Fire (can shoot fireballs).

The main goal of each level is to get to the end of the level, which means traversing it from left to right. Auxiliary goals include collecting as many as possible of the coins that are scattered around the level, clearing the level as fast as possible, and collecting the highest score, which in part depends on number of collected coins and killed enemies.

Complicating matters is the presence of holes and moving enemies. If Mario falls down a hole, he loses a life. If he touches an enemy, he gets hurt; this means losing a life if he is currently in the Small state. If he's in the Big state, he changes to Small, and if he's in the Fire state, he's degraded to merely Big. However, if he jumps so that he lands on the enemy from above, different things happen. Most enemies (e.g. goombas, fireballs) die from this treatment; others (e.g. piranha plants) are not vulnerable to this and proceed to hurt Mario; finally, turtles withdraw into their shells if jumped on, and these shells can then be picked up by Mario and thrown at other enemies to kill them.

Certain items are scattered around the levels, either out in the open, or hidden inside blocks of brick and only appearing when Mario jumps at these blocks from below so that he smashes his head into them. Available items include coins which can be collected for score and for extra lives (every 100 coins), mushrooms which make Mario grow Big if he is currently Small, and flowers which make Mario turn into the Fire state if he is already Big.

No textual description can fully convey the gameplay of a particular game. Only some of the main rules and elements of Super Mario Bros are explained above; the original game is one of the world's best selling games, and still very playable more than two decades after its release in the mid-eighties. It's game design has been enormously influential and inspired countless other games, making it a good choice for experiment platform for player experience modelling.

While implementing most features of Super Mario Bros, the standout feature of Infinite Mario Bros is the automatic generation of levels. Every time a new game is started, levels are randomly generated by traversing a fixed width and adding features (such as blocks, gaps and opponents) according to certain heuristics. The level generation can be parameterized, including the desired difficulty of the level, which affects the number and placement of holes, enemies and obstacles. In our modified version of Infinite Mario Bros we can specify the random seed of the level generator, making sure that we can recreate a particular randomly created level whenever we want.

Several features make Super Mario Bros particularly interesting from an RL perspective. The most important of these is the potentially very rich and high-dimensional environment representation. When a human player plays the game, he views a small part of the current level from the side, with the screen centered on Mario. Still, this view often includes many tens of objects such as brick blocks, enemies and collectable items. These objects are spread out in a semi-continuous fashion: the static environment (grass, pipes, brick blocks etc.) and the coins are laid out in a grid (of which the standard screen covers approximately $15 * 15$ cells), whereas moving items (most enemies, as well as the mushroom power-ups) move almost continuously at pixel resolution.

The action space, while discrete, is also rather large. In the original Nintendo game, the player controls Mario with a D-pad (up, down, right, left) and two buttons (A, B). The A button initiates a jump (the height of the jump is determined partly by how long it is pressed) and the B button initiates running mode. Additionally, if Mario is in the Fire state, he shoots a fireball when the B button is pressed. Disregarding the unused up direction, this means that the information to be supplied by the controller at each time step is five bits, yielding $2^5 = 32$ possible actions, though some of these are nonsensical and disregarded (e.g. pressing left and right at the same time).

[1] http://julian.togelius.com/mariocompetition2009
[2] http://www.mojang.com/notch/mario/

Another interesting feature is that different sets of behaviours and different levels of coordination between those behaviours are necessary in order to play levels of different difficulty, and complete these with different degrees of success. In other words, there is a smooth learning curve between levels, both in terms of which behaviours are necessary and their necessary degree of refinement. For example, to complete a very simple Mario level (with no enemies and only small and few holes and obstacles) it might be enough to keep walking right and jumping whenever there is something (hole or obstacle) immediately in front of Mario. A controller that does this should be easy to learn. To complete the same level while collecting as many as possible of the coins present on the same level likely demands some planning skills, such as smashing a power-up block to retrieve a mushroom that makes Mario Big so that he can retrieve the coins hidden behind a brick block, and jumping up on a platform to collect the coins there and then going back to collect the coins hidden under it. More advanced levels, including most of those in the original Super Mario Bros game, require a varied behaviour repertoire just to complete. These levels might include concentrations of enemies of different kinds which can only be passed by observing their behaviour pattern and timing Mario's passage precisely; arrangements of holes and platforms that require complicated sequences of jumps to pass; dead ends that require backtracking; and so on. How to complete Super Mario Bros in minimal time while collecting the highest score is still the subject of intense competition among human players[3].

III. How?

Much of the work that went into this paper consisted in transforming the Infinite Mario Bros game into a piece of benchmarking software that can be interfaced with reinforcement learning algorithms. This included removing the real-time element of the game so that it can be "stepped" forward by the learning algorithm, removing the dependency on graphical output, and substantial refactoring (as the developer of the game did not anticipate that the game would be turned into an RL benchmark). Each time step, which corresponds to 40 milliseconds of simulated time (an update frequency of 25 fps), the controller receives a description of the environment, and outputs an action. The resulting software is a single-threaded Java application that can easily be run on any major hardware architecture and operating system, with the key methods that a controller needs to implement specified in a single Java interface file (see figures 2 and 3). On an iMac from 2007, $5-20$ full levels can be played per second (several thousand times faster than real-time) depending on the level type and controller architecture. A TCP interface for controllers is also provided, along with an example Python client. However, using TCP introduces a significant connection overhead, limiting the speed to about one game per minute (three times real-time speed).

[3]Search for "super mario speedrun" on YouTube to gauge the interest in this subject.

```
public enum AGENT_TYPE
       {AI, HUMAN, TCP_SERVER}
public void reset();
public boolean[] getAction
       (Environment observation);
public AGENT_TYPE getType();
public String getName();
public void setName(String name);
```

Fig. 2. The *Agent* Java interface, which must be implemented by all controllers. Called by the game each time step.

```
// always the same dimensionality 22x22
// always centered on the agent
public byte[][] getCompleteObservation();
public byte[][] getEnemiesObservation();
public byte[][] getLevelSceneObservation();
public float[] getMarioFloatPos();
public float[] getEnemiesFloatPos();
public boolean isMarioOnGround();
public boolean mayMarioJump();
```

Fig. 3. The *Environment* Java interface, which contains the observation, i.e the information the controller can use to decide which action to take.

We devised a number of variations on a simple neural-network based controller architecture, varying in whether we allowed internal state in the network or not, and how many of the "blocks" around Mario were used as inputs. The controllers had the following inputs; the value for each input can be either 0 (on) or 1 (off).

- A bias input, with the constant value of 1.
- One input indicating whether Mario is currently on the ground.
- One input indicating whether Mario can currently jump.
- A number of input indicating the presence of environmental obstacles around Mario.
- A number of input indicating the presence of enemies

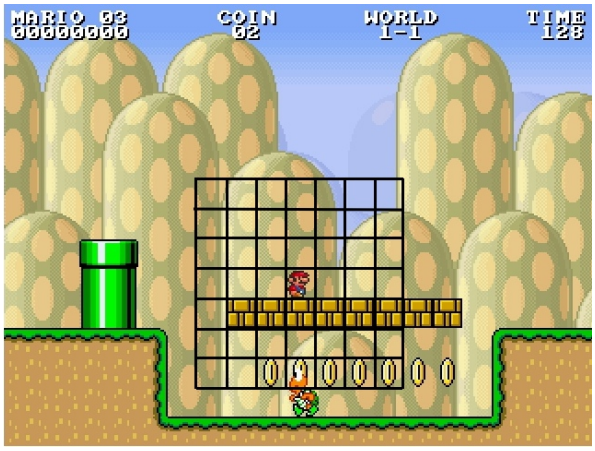

Fig. 4. Visualization of the environment and enemy sensors. Using the smallest number of sensors, the top six environment sensors would output 0 and the lower three input 1. All of the enemy sensors would output 0, as even if all 49 enemy sensors were consulted none of them would reach all the way to the body of the turtle, which is four blocks below Mario. None of the sensors register the coins.

around Mario.

The number of inputs for environmental obstacles and enemies are either 9, 25 or 49, arranged in a square centering on Mario (in his Small state, Mario is the size of one block. This means that each controller has either 21 (bias + ground + jump + 9 environment + 9 enemy), 53 or 101 inputs. See figure 4 for a visualization and further explanation of the inputs.

These inputs are then fed in to either an Multi-Layer Perceptron (MLP) or a Simple Recurrent Network (SRN, also called Elman network). Both types of network have 10 hidden nodes and *tanh* transfer functions.

We initially used simple $\mu + \lambda$ Evolution Strategies (ES) with $\mu = \lambda = 50$ and no self-adaptation. The mutation operator consisted in adding random numbers drawn from a Gaussian distribution with mean 0 and standard deviation 0.1 to all weights. Each run of the ES lasted for 100 generations.

The input space for this problem has a higher-dimensionality than what is commonly the case for RL problems, and there is likely to be significant regularities in the inputs that can be exploited to design competent controllers more compactly. The simple neuroevolutionary mechanism described above does not take any such regularity into account. We therefore decided to also explore the *HyperGP* [14] hybrid neuroevolution/genetic programming algorithm, which has previously been shown to efficiently evolve solutions that exploit regularity in high-dimensional input spaces.

HyperGP evolves neuron weights as a function of their coordinates in a Cartesian grid called a substrate using Genetic Programming. HyperGP is an indirect encoding algorithm inspired by the HyperNEAT [15], which uses the evolved neural networks (generated with NEAT) as the weight generating function. In HyperGP, the NEAT is replaced by the Genetic Programming. NEAT features complexification, which means that it starts with a simple linear function and adds more units during the evolution. HyperGP generates complex expression from the beginning, thus the convergence is in many cases faster than in HyperNEAT [14]. Each HyperGP controller was evolved 100 generations of populations of 100 individuals consisting 7 evolved function for weight matrices. The function expressions of maximum depth of 4 were used.

The fitness function is based on how far Mario could progress along a number of different levels of different difficulty. The progress is measured in the game's own units; the levels vary slightly in length, but are between 4000 and 4500 units long. Each controller was evaluated by testing it on one level at a time, and using the progress made on this level as fitness value. The same random seed (and thus the same level, as long as the difficulty stayed the same) was used for each fitness evaluation during an evolutionary run in order to not have to remove noise from fitness evaluations; this seed was changed to a new random number between evolutionary runs. Each evolutionary run started with a difficulty level of 0 but every time a controller in the population reached a fitness above 4000 , which we interpret as clearing a level or at least being very close to clearing it, the difficulty was incremented by one step. This means that a new level, usually including somewhat more gaps and enemies and more complicated terrain, was used for fitness evaluation instead.

After each evolutionary run, the generalization capacity of the best controller present in the population of the last generation was tested. This was done by testing it on 4000 new levels, 1000 each of the difficulties 0, 3, 5 and 10. The random seeds for these levels were kept fixed between evolutionary runs.

Table I presents the highest difficulty reached by the controllers of each type, and the performance of the controllers on the test set of 4000 levels.

TABLE I

RESULTS (LEVEL REACHED, SCORES IN LEVELS 0, 3, 5 AND 10), AVERAGED OVER THE BEST CONTROLLERS OF EACH TYPE FOUND DURING APPROXIMATELY 6 (BETWEEN 4 AND 8) INCREMENTAL EVOLUTIONARY RUNS. RESULTS FOR BOTH MLP- AND SRN-BASED NETWORKS ARE SHOWN. LAST THREE LINE CONTAIN STATISTICS FROM LARGE SRN CONTROLLERS EVOLVED BY HYPERGP ALGORITHM.

Controller	Level	0	3	5	10
Small MLP	3	1784	719	606	531
Medium MLP	0.83	864	456	410	377
Large MLP	0.5	559	347	345	300
Small SRN	2.83	3050	995	834	692
Medium SRN	1.83	1625	670	576	512
Large SRN	0.25	757	440	408	373
Small HyperGP SRN	1.87	2365	780	695	585
Medium HyperGP SRN	2.13	2352	786	679	545
Large HyperGP SRN	1.25	2314	633	588	534

As can be seen from Table I, we can relatively easily evolve controllers that can clear individual levels of difficulty level two, and sometimes three. Levels of difficulty three contains occasional gaps, and a healthy number of enemies of all types, including cannons. (In contrast, levels of difficulty zero contain no gaps, fewer enemies (*goombas* and *turtles* only) and overall a flatter landscape.)

However, there are problems with generalization. Controllers that have managed to progress to clear levels of difficulty 2 or 3 have problems with clearing levels of the same difficulty other than the particular level they were trained on, and often even fail to clear level 0.

Looking at the behaviour of some of the best controllers from individual evolutionary runs on other levels than they were trained on, it seems that the one skill every controller has learnt is to run rightwards and jump when the current stretch of ground they are on ends. This could be either in front of a gap (which Mario would die from falling into) or when the platform Mario stands on ends, even though there is firm ground to land on below. In some cases, Mario jumps unnecessarily jumps off a platform just to land inside a gap later on, something that could have been avoided if a larger portion of the environment could have been taken into account.

None of the controllers are very good at handling ene-

mies. Most of the time Mario just runs into them, though occasionally he seems to be jumping over enemies directly in front. Still, failing to complete a level because of dying from running into an enemy seems to be comparably rare, meaning that selection pressure for handling enemies better is likely too low. As Mario starts in the Fire state, he needs to run into three enemies in order to die. Instead, failure to complete a level is typically due to falling into a hole, or getting stuck in front of a wall, for some reason failing to jump over it.

None of the evolved controllers pay any attention to coins and item blocks, and any collected coins are purely by chance — they happened to be where to controller wanted to go anyway. This is not surprising as they have no way of "seeing" coins or items.

Comparing the fitness of reactive and recurrent controllers, the SRN-based controllers perform about as good as the MLP-based controllers both in terms of the average maximum training level reached and in terms of score on the test levels. However, controlelrs with larger input spaces that "see" more of the game environment perform worse even though they have access to more information; Large controllers perform worse than Medium controllers which in turn perform worse than Small controllers. The simplest controllers, based on a feedforward network with 21 inputs performed very much better than the most complex controllers, based on recurrent controllers with 101 inputs. It seems that the high dimensionality of the search space is impeding the evolution of highly-fit large controllers, at least as long as the controller is represented directly with a linear-length encoding.

Main advantage of the HyperGP is a capability of evolution of large controllers with 101 inputs. The following set of 7 functions is an example of a genome that can clear level 2. Note that the first function just generates 0 valued weights for inputs containing shape of the ground. This controller moves forward, jumps and kills enemies by firing but is not robust enough to avoid randomly placed holes in the terrain:

$$f_1 = 0, f_2 = x_2^2 x_3^2, f_3 = \sin\cos x_1, f_4 = |x_1| + \cos x_1,$$

$$f_5 = e^{-\left(\sqrt{|x_1|}-1\right)^2}, f_6 = \sqrt{|x_1|}\cos x_1, \sin x_1 x_2, f_7 = x_1^4$$

The complete set of function contains 42 nodes, whereas the generated large network contains 1172 weights. Such compression of the search space allows to generate large network with a good performance in a reasonable number of evaluations. Performance of HyperGP evolved networks is similar regardless to a number of inputs used. The HyperGP evolved recurrent neural network do not outperform small networks evolved by direct encoding of weights in genomes. The HyperGP in fact allows evolution of networks with a high number of inputs, which is almost impossible or gives poor results using direct encoding.

Figure 5 depicts a typical evolution of large controller evolved by the HyperGP. The plot contains 100 generations

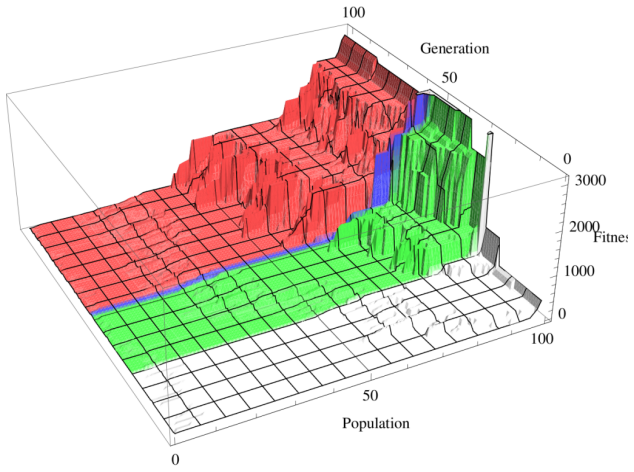

Fig. 5. Example HyperGP evolution of the large SRN controller. The plot contains sorted population of individuals. Each level in the incremental evolution is colored with a different color (white for level 0). For example, controller that clears level 1 just needs three generations to be able to clear level 3.

of individual controllers sorted by their fitness value. We can see how the controller advances the game levels (colored stripes), when it reaches maximum fitness of 4000. All controllers are reevaluated when the desired fitness is reached (therefore the maximum fitness is not included in the plot) and used in the next level. We can see that the controllers may perform well in the next level. For example, the controller for level 1 performs well in level 2 and just three generations are enough to advance to level 3.

IV. So?

We have described a new RL benchmark based on a version of the popular platform game Super Mario Bros, and characterized how it offers unique challenges for RL research. We have also shown that it is possible to evolve controllers that play single levels of the game quite well, using a relatively naive neural network architecture and input representation. However, these controllers have problems with generalization to other levels, and with taking anything temporally or spatially distant into account.

So where do we go from here? How do we go about to learn controllers that play Super Mario Bros better?

The problems with generalization might be solved through using new seeds for every evaluation, though that will lead to problems with noise that might require averaging over a large number of evaluations to achieve reliable enough fitness values for evolution. Another solution could be to incrementally increase the number of levels used for each evaluation, as was done in [16]; however, this also requires additional computational time.

It is arguably more important, and more interesting, to overcome the problems with spatial and temporal reach. From our results above, it is clear that using simple recurrent networks rather than MLPs did not affect the performance significantly; nor did we expect this simple recurrent architecture to be able to solve the problems of temporal reach.

It is possible that more sophisticated recurrent architecture such as Long-Short Term Memory (LSTM) can be used to learn controllers that take more temporally distant events into account [17]. An example of the long-term dependencies that could be exploited is that if a wall was encountered 50 or 100 time steps ($2-4$seconds) ago, hindering progress towards the goal, the controller could remember this and go into "backtracking mode", temporarily moving away from the goal and trying to jump onto a higher platform before resuming movement towards the goal.

The problems of spatial reach were not solved by simply adding more inputs, representing a larger part of the environment, to the standard neural network. Indeed, it seems that simply adding more inputs decreases the evolvability of the controllers, probably due the added epistasis of high-dimensional search spaces.

Given that there are certain regularities to the environment description (e.g. a piranha plant in front of Mario means approximately the same thing regardless of whether it is 6 or 8 blocks away) we believe that these problems can be overcome by using neural network architectures that are specifically designed to handle high-dimensional input spaces with regularities. In particular, we plan to perform experiments using both Multi-Dimensional Recurrent Neural Networks [18] to see if a larger part of the input space can successfully be taken into account.

HyperGP evolves controllers with similar performance regardless to a size of the input window. The results are worse than those given by small networks evolved using direct encoding but it can evolve large input networks. Although it performs relatively well with large number of inputs, further testing of hypercube encoded networks will be focused on scalability of either in input space or in the size of the network itself. It requires testing of networks evolved with a particular number of inputs on a setup with different number of inputs or different number of neurons using the same functions that generate the weight matrices.

If this is successful, the next step would be to include more observation matrices, allowing the controller to see coins and item blocks, and possibly differentiate between different types of enemies, which in turn would allow more sophisticated strategies. This would mean observations with many hundreds of dimensions. Examples of successful reinforcement learning in nontrivial problems with such large input spaces are scarce or nonexistent; this is probably due to lack of both learning algorithms capable to handle such problems, and benchmark problems to test them. We believe that the video game-based benchmark presented in this paper goes some way towards meeting the demands for such benchmarks.

Another way in which the environment representation can be made richer in order to permit more sophisticated game play is to introduce continuous inputs signifying how far from the center of the current block Mario is. This would allow more precise spatial positioning, which is necessary for some complicated jump sequences.

We believe that the techniques used in this paper have merely scratched the surface of what is possible when it comes to learning strategies for this Super Mario Bros-based benchmark. Fortunately, the free availability of the source code and the associated competition makes it possible for anyone, including you, to try your best technique at the problem and compare your results with others.

ACKNOWLEDGEMENTS

This research was supported in part by the SNF under grant number 200021-113364/1. Thanks to Tom Schaul for useful comments and chocolate.

REFERENCES

[1] R. Koster, *A theory of fun for game design*. Paraglyph press, 2005.
[2] J. Togelius, T. Schaul, D. Wierstra, C. Igel, F. Gomez, and J. Schmidhuber, "Ontogenetic and phylogenetic reinforcement learning," *Zeitschrift Künstliche Intelligenz*, 2009.
[3] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation in racing games," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.
[4] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.
[5] A. Agapitos, J. Togelius, S. M. Lucas, J. Schmidhuber, and A. Konstantinides, "Generating diverse opponents with multiobjective evolution," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.
[6] G. N. Yannakakis and J. Hallam, "Real-time Adaptation of Augmented-Reality Games for Optimizing Player Satisfaction," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. Perth, Australia: IEEE, December 2008, pp. 103–110.
[7] S. Lucas, "Evolving a neural network location evaluator to play ms. pac-man," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 203–210.
[8] M. Parker and G. B. Parker, "The evolution of multi-layer neural networks for the control of xpilot agents," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.
[9] M. Parker and B. D. Bryant, "Visual control in quake ii with a cyclic controller," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008, p. 8.
[10] R. Kadlec, "Evolution of intelligent agent behaviour in computer games," Master's thesis, Charles University in Prague, Sep 2008.
[11] J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," in *Proceedings of the Congress on Evolutionary Computation*, 2005.
[12] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Saez, "The WCCI 2008 simulated car racing competition," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.
[13] T. Graepel, R. Herbrich, and J. Gold, "Learning to fight," in *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.
[14] Z. Buk, J. Koutník, and M. Šnorek, "NEAT in HyperNEAT substituted with genetic programming," in *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms (ICANNGA 2009)*, 2009.
[15] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based indirect encoding for evolving large-scale neural networks," *Artificial Life*, vol. 15, no. 2, 2009.
[16] J. Togelius and S. M. Lucas, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.
[17] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context free and context sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, pp. 1333–1340, 2001.
[18] T. Schaul and J. Schmidhuber, "Scalable neural networks for board games," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 2008.