Neuro-Evolution of Spiking Neural Networks on SpiNNaker Neuromorphic Hardware

Alexander Vandesompele[*], Florian Walter, Florian Röhrbein
Chair of Robotics and Embedded Systems, Department of Informatics, Technical University of Munich, Boltzmannstraße 3, 85748 Garching bei München, Germany

***Abstract*—Neuro-evolutionary algorithms optimize the synaptic connectivity of sets of candidate neural networks based on a task-dependent fitness function. Compared to the commonly used methods from machine learning, many of them not only support the adaptation of connection weights but also of the network topology. However, the evaluation of the current fitness requires running every candidate network in every generation. This becomes a major impediment especially when using biologically inspired spiking neural networks which require considerable amounts of simulation time even on powerful computers. In this paper, we address this issue by offloading the network simulation to SpiNNaker, a state-of-the art neuromorphic hardware architecture which is capable of simulating large spiking neural networks in biological real-time. We were able to apply SpiNNaker's simulation power to the popular NEAT algorithm by running all candidate networks in parallel and successfully evolved spiking neural networks for solving the XOR problem and for playing the Pac-Man arcade game**

I. Introduction

Biological neural networks are computing systems. They perform a variety of complex tasks, including sensory processing and muscular control. As such, biological networks have inspired artificial intelligence (AI) researchers since the beginning of the field. Some of the oldest concepts in AI are the McCulloch-Pitts neuron model [1] and Rosenblatt's perceptron [2]. These neurons follow a digital threshold logic. Advancing these neuron models by enabling a continuous output, rather than a binary, resulted in networks that are proven to be universal function approximators (given a small number of requirements [4]). Indeed, their computational power has resulted in outstanding performance on a variety of pattern recognition tasks, including supervised classification, natural language processing and robot control [5], [6].

Regardless of their performance, these artificial neural networks are still comprised of neuron models that are vast abstractions of biological neurons. Biological neurons integrate incoming spikes which affect the membrane potential and emit a spike if some threshold is crossed. The spike is a stereotypical event, therefore all information is assumed to be contained in the timing of the spike. The neuron models used in spiking neural networks (SNN) aim to better mimic their biological counterparts. The output of these neurons is no longer a simple function of its inputs, rather the internal state of the neuron is modelled and incoming spikes have a temporal effect [7]. The outputs are single, stereotypical events. A range of neuron models is available such as the leaky integrate-and-fire neuron [8] or the Hodgkin-Huxley model [9]. Maass [3] proved that SNNs are computationally more powerful than other types of artificial neural networks of the same size.

To create a functional SNN, a number of network architecture design choices have to be addressed, like the network size and connectivity. Evolutionary algorithms offer automated evolution of neural networks, some of them determining complete network topology (altering the number of neurons and their connectivity). Another advantage of evolutionary algorithms is that the performance criterion can be defined in a more flexible manner compared to other learning algorithms for neural networks. The main computational cost for neuro-evolution of SNNs is usually the numerous network simulations that are required for evaluating the networks.

SpiNNaker (Spiking Neural Network Architecture) is a neuromorphic hardware designed to efficiently simulate large numbers of neurons. SpiNNaker can simulate networks of virtually any topology, including multiple networks in parallel. It is therefore advantageous to apply its simulation power in the evaluation of the networks in the evolutionary algorithm.

In the following sections, the NEAT algorithm and the SpiNNaker hardware are briefly introduced. The methods section presents the experimental setup that combines SpiNNaker and NEAT in order to create functional SNNs. The setup is tested by solving the XOR problem and next used to create functional SNNs that play Pac-Man[1], a classic arcade game originally developed by Toru Iwatani for the Namco Company in 1980.

A. Neuro-Evolution of Augmenting Topology (NEAT)

Functional networks for a variety of problems have been found by the NEAT algorithm [14], [15] and by other neuro-evolutionary algorithms, including the design of robotic morphology and/or control [18]–[21] and the design of agents for different games [22]–[24].

[*] Now employed at the Electronics and Information Systems Department at Ghent University, Belgium

[1]https://en.wikipedia.org/wiki/Pac-Man

NEAT is an evolutionary algorithm that creates neural networks [16]. It evolves both topology and connection weights. NEAT employs a direct encoding system, all neurons and connections in a given network are directly represented by a neuron or link gene. It is particularly suited for evolving neural networks because of (i) gradual complexification, starting with minimal networks and incrementally adding neurons and/or connections, (ii) a structured crossover mechanism that allows inheritance of network functionality, (iii) innovation protection by maintaining multiple species.

NEAT starts with a population of minimal networks. Each networks consists of only input and output neurons, which are fully connected with random connection weights. The number of input and output nodes are given by the experimenter and remain static. The fitness of each network determines its chance to create offspring. Offspring can differ from its parents through any of three mutation types: change of connection weight, addition/removal of a connection and addition/removal of a hidden neuron. Networks of the population can thus gradually grow in size with each generation.

Structural mutations often decrease fitness initially even if they are required for a solution in the long run. Some time may be needed for a new structure to be integrated. Due to the initial decrease in fitness the network is less likely to survive and the innovation might become extinct before optimization revealed its added value. Therefore the NEAT algorithm has a mechanism to protect innovation: speciation. Speciation implies the maintenance of multiple species among the entire population of networks. Offspring is only created based on two parents from the same species. Different species can thus evolve in parallel and networks compete primarily within their species. NEAT uses explicit fitness sharing [17], forcing networks of the same species to share their fitness. The result is that the size of each species is proportional to the performance of its networks. Fitness sharing prevents a single species from dominating the entire population and allows some time for new mutations to optimize.

B. SpiNNaker

SpiNNaker is a neuromorphic hardware designed to simulate large scale spiking neural networks in biological real-time [10], [11]. SpiNNaker is built with standard computer hardware components, but the architecture is optimized for spike transmission [12]. In SNNs, spikes are stereotypical all-or-none events. All the information carried by a spike is contained in the source of the spike and the timing of the spike. Therefore, a SNN can be efficiently modelled with packet-based communication using the AER protocol [13]. This communication scheme detaches the topology of the neural network from the actual physical wires.

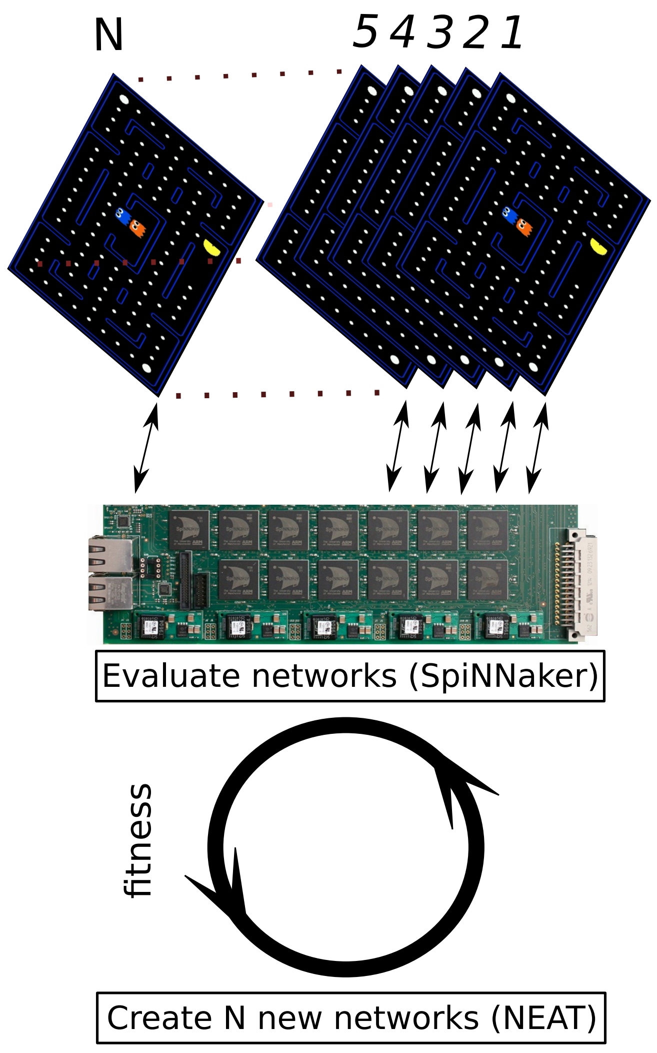

Fig. 1. Overview of the experimental loop. Based on the fitness values of the parent generation, the NEAT algorithm creates new networks (the offspring generation, below). All offspring networks are simultaneously simulated on SpiNNaker to assess their performance (here N Pac-Man games, top). The score achieved by each network is used as fitness values by the NEAT algorithm.

On SpiNNaker, AER packets contain a source neuron identifier and are sent to the on-chip packet router that relays the packet to any possible location. The routing mechanism of the SpiNNaker architecture [11] is optimized to communicate large numbers of small AER packets. The unit of a SpiNNaker machine is a chip containing 18 ARM968 processors and local memory. All chips together form a toroidal grid, a spike emitted in one chip can be routed to any other location on the grid. The topology of the simulated network is therefore easily reconfigurable by adjusting the routing tables. SpiNNaker can emit spikes much faster than biological networks. Therefore many biological wires (axons) can be replaced by few electronic wires. This difference in operational speed between SpiNNaker and biological networks enables topology virtualization and simulations on a biologically realistic time-scale. Spike timing is implicit. The combination of a packet arriving at a specific time and at a specific location is equivalent to the biological synaptic event where one neuron receives input from another neuron.

The number of neurons that can be simulated in practice depends on a number of factors including the neuron model, synapse model, number of synapses per neurons, average firing rate of neurons and network connectivity. Each chip can simulate a few thousand simple neuron models such as leaky integrate-and-fire neurons, with approximately 1000 input synapses per neuron. The SpiNNaker architecture is highly scalable, from a single chip up to 65 536 chips [12]. SpiNNaker is built with energy efficient components. Power consumption is further optimized by its event-driven operation. A single chip consumes only 1 W. This makes SpiNNaker affordable to operate and suitable for use on mobile, wireless robotic platforms.

II. Methods

The NEAT algorithm and a 48-chip SpiNNaker machine (SpiNN-5) are both integrated in an automatized loop that constitutes the evolutionary experiment (fig. 1 on the preceding page). Each experiment starts with NEAT creating a random initial population of networks. In order to create the next generation, the NEAT algorithm needs the fitness of each network. To obtain these fitnesses, SpiNNaker is configured to simulate all networks of the population in parallel. Each network can communicate inputs and outputs via an ethernet connection with a host PC. An open source C++ implementation of NEAT [2] is used to find functional networks for the XOR problem and for playing a Pac-Man [3] game. Table I displays used values of the NEAT algorithm's main parameters.

All neurons are simple leaky integrate-and-fire neurons (fig. 2) with alpha-shaped synaptic currents. When the threshold potential (-50mV) is reached, a spike is emitted and the membrane potential is set to -70mV for 2ms (the refractory period). Connection weights are real-valued and kept in the range [-8.0, 8.0], and can change between inhibitory and excitatory by connection weight mutations. Only feed forward connections are used.

To run multiple networks on SpiNNaker in parallel, all networks together are simulated as one big network, described with pyNN [25]. Individual networks will not affect each other due to the lack of connectivity. For evolving Pac-Man controllers (see section III.B), a Pac-Man game is instantiated on the host PC for each individual network simulated on SpiNNaker. Inputs for the networks are extracted from Pac-Man game frames. The inputs activate the corresponding input neurons and are processed through the network simulations. The activity of the output neurons is sent back to the host PC. Based on the output activity, a decision for each Pac-Man game instance is extracted and the next frame is simulated on the host PC. This loop continues until game over occurs or a specified number of frames is reached. The score of the Pac-Man game at this point is used as the networks fitness value by NEAT algorithm that creates the next generation of networks.

[2]MultiNEAT, https://github.com/peter-ch/MultiNEAT

[3]Edx-ai-project1, https://github.com/kalys/edx-ai-project1

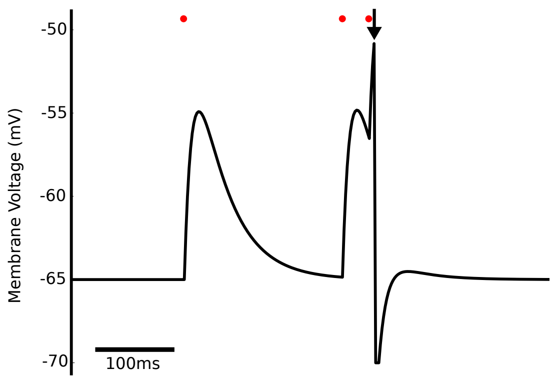

Fig. 2. Membrane voltage of a leaky integrate-and-fire neuron simulated on SpiNNaker. Red dots indicate the arrival of a spike from an excitatory synapse. The arrow indicates time of crossing firing-threshold.

Parameter	Value
population size	250
survival proportion	0.2
overall mutation rate	0.25
crossover rate	0.75
elitism proportion	0.1
mutation probability change connection weight	0.5
mutation probability add node	0.1
mutation probability remove node	0.1
mutation probability add connection	0.1
mutation probability remove connection	0.1
mutation probability remove simple neuron	0.1

TABLE I
Main parameters of the NEAT algorithm.

III. Results

A. The XOR problem

To test the experimental set-up, the benchmark XOR problem is used. The XOR problem cannot be solved by a linear classifier and artificial neural networks require hidden neurons to solve non-linear problems.

Each network has two input neurons and two output neurons. The number of hidden neurons, connections and connection weights were evolved by the NEAT algorithm. Table II displays the training pattern used. Classification was determined by the output neuron with highest firing rate. The NEAT algorithm in this set-up could find a solution to the XOR problem in 199 generations (fig. 3 on the following page).

Input1	Input2	Output
5 Hz	5 Hz	1
20 Hz	5 Hz	2
5 Hz	20 Hz	2
20 Hz	20 Hz	1

TABLE II
INPUT/OUTPUT FIRING PATTERNS FOR THE XOR PROBLEM. OUTPUT REFERS TO THE OUTPUT NEURON WITH HIGHEST FIRING RATE.

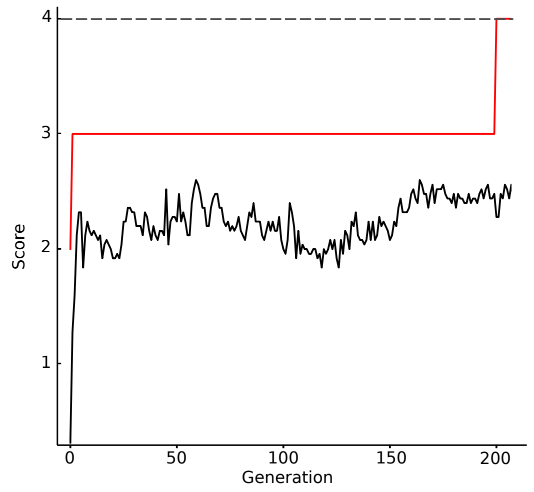

Fig. 3. Result for the XOR problem. Maximum (red, middle trace) and average (black, bottom trace) score (number of correctly classified input patterns) of each generation. Score of 4 (dashed line, top) means all input patterns were correctly classified.

B. The Pac-Man controller

The Pac-Man task used is a traditional game of 11x20 square tiles (fig. 4). It includes all components of the classic Pac-Man game. Ghosts have a random component in their decision making, and can eat Pac-Man ('hungry ghost'). Except if Pac-Man has eaten a power-pill (the large dots), which makes the ghosts eatable ('scared ghost'). The initial position of Pac-Man is random in each game, ghosts always start at the same central position.

The sensory inputs that the Pac-Man controller receives (table III) consist of only local information, i.e. things that occur within a radius of 1 or 2 tiles from Pac-Man. This resembles the case of a natural agent or a mobile robot that has only sensory input of its immediate surrounding available. Among the inputs are presence of environmental objects (walls, dots, power-pills) and ghosts (hungry or scared), and the previous position of Pac-Man (if Pac-Man moved since the previous frame, the neuron corresponding to the previous position of Pac-Man will be active, otherwise all will be silent). Each object can be detected in four directions (up/right/down/left, N=4 input neurons), this makes a total of 24 input neurons. All input neurons operate in a binary mode, meaning that the input neuron is maximally active if the object is present and silent otherwise.

N (Input neurons)	Object	Radius
4	dot	1
4	wall	1
4	powerpill	1
4	hungry ghost	2
4	scared ghost	2
4	prev. position	1
= 24		

TABLE III
SENSORY INPUTS FOR THE PAC-MAN CONTROLLER NETWORKS.

Action	Points
eat dot	+10
eat ghost	+200
be eaten	-100
each frame	-1

TABLE IV
SCORE SYSTEM OF A PAC-MAN GAME.

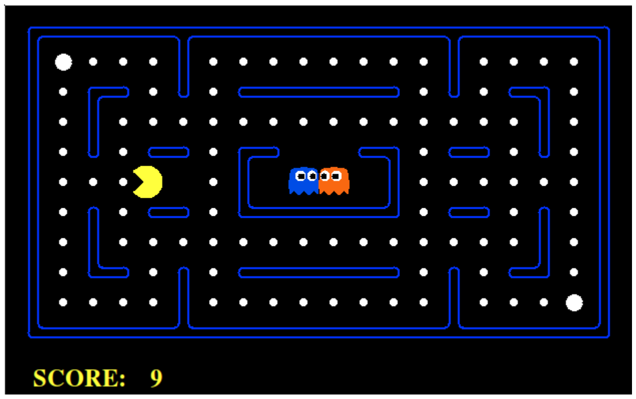

Fig. 4. Game maze. All classic Pac-Man ingredients are included.

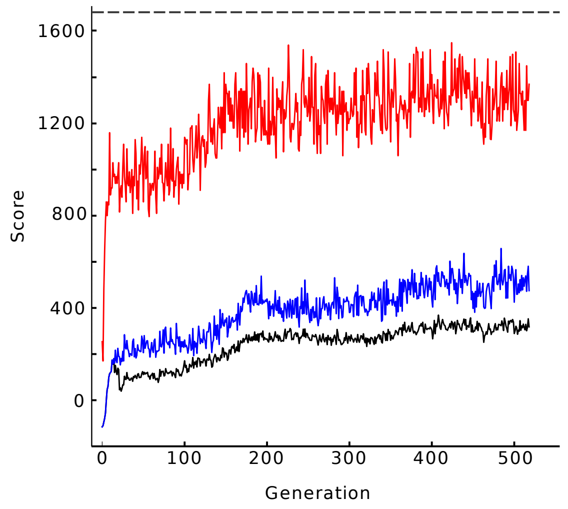

Fig. 5. Result for the Pac-Man game. Maximum (red), average (black) and average of best species (blue) are shown for each generation (we refer to the online version for the color coding). Gray dashed line indicates the theoretical maximum score.

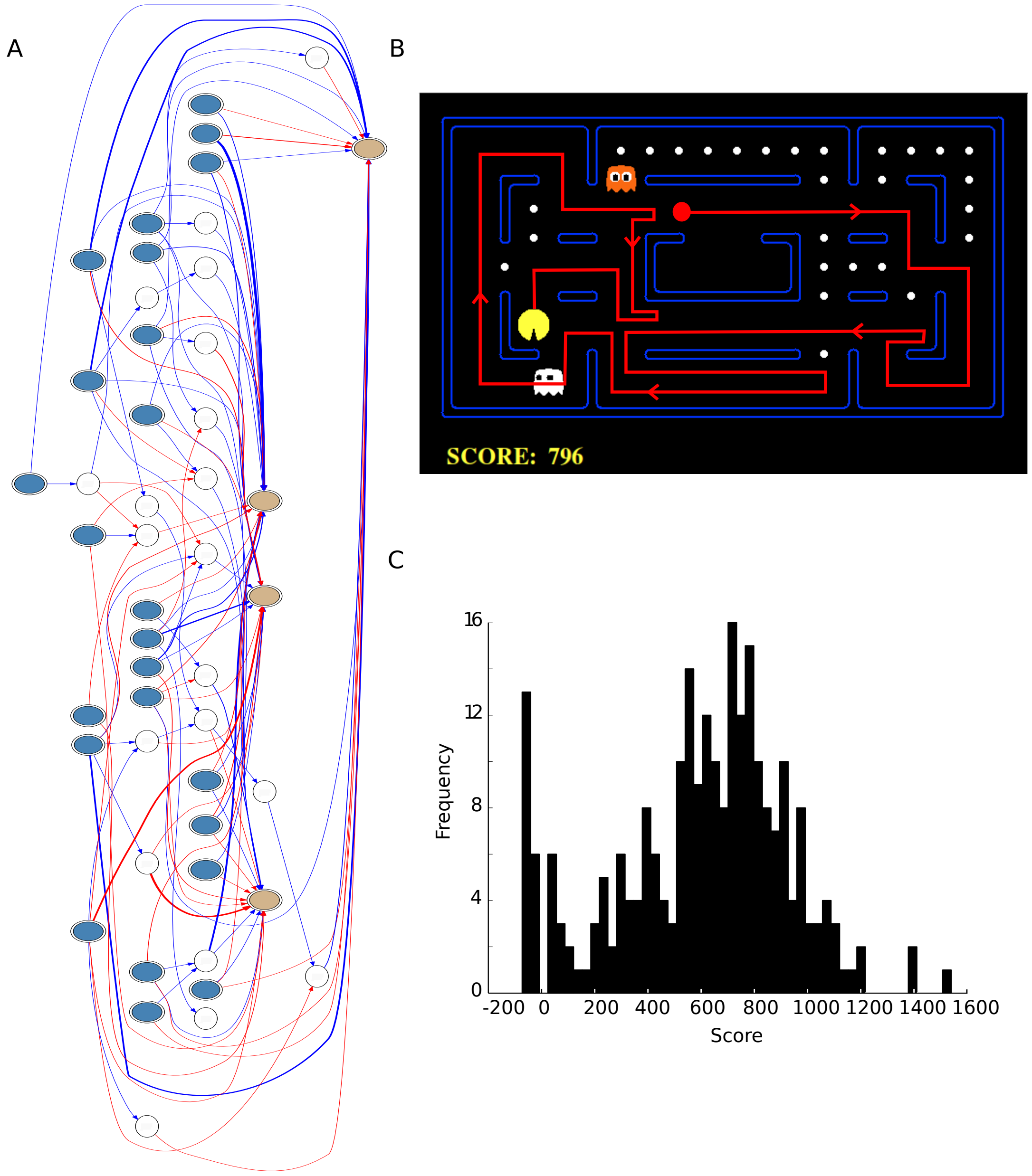

Fig. 6. A. Example network from the best species in the last generation (blue nodes = inputs, white = hidden, brown = outputs). Thickness of the excitatory (blue) and inhibitory (red) connections corresponds to its weight (we refer to the online version for the color coding). B. Example trajectory of a Pac-Man controlled by the network in A. Red dot indicates start position. C. Histogram of the performance of the network in A (250 games, bin width = 40).

There are 4 output neurons per network, each representing an action by Pac-Man (go up/right/down/left). The action of Pac-Man is decided by the output neuron with highest firing rate, given that it surpasses a threshold of 7 Hz. This threshold is arbitrary and allows the controller to undertake no action given a low output firing rate.

By interacting with it's environment, Pac-Man can gain or lose points (table IV). Pac-Man receives points for eating dots and ghosts. It loses 1 point per frame, which serves as a penalty for doing nothing. When being eaten, Pac-Man gets a penalty plus the game is terminated. Each network gets to play two games. The average end score of both games is used as the fitness value assigned to the network controlling Pac-Man.

The NEAT algorithm ran for 518 generations and was capable of creating neural controllers of increasing quality (fig. 5 on the preceding page). The theoretical maximum score was never reached, which can be expected given that the networks receive only local information and thus cannot, for instance, hunt down far-away ghosts. A network of the last generation had on average 18 hidden neurons and 111 connections. Fig 6A displays an example network from the best species in the last generation. It scored on average 602 ± 323 (N=250) (fig. 6 C).

This experiment took approximately 15 hours. The relatively long experiment time is due in part to the time required to load the networks onto the SpiNNaker machine. This loading time increases with larger networks and higher connectivity. A lot of time is also used by the Python program, there is great room for reducing experiment time by rewriting the program and make full use of parallel computing (e.g. for the simulation of the Pac-Man games). Up to 11.5k neurons were simulated in parallel, meaning that only a fraction of the simulation power of the 48-chip SpiNNaker was used. This was again a consequence of the limitations of the specific Python program used (it could handle only a certain rate of input) but is by no means a fundamental limit. Simulating more neurons on SpiNNaker could allow for instance to assess the same network design multiple times in parallel, improving network assessment quality and, depending on the problem at hand, the search speed.

IV. Conclusion

Spiking neural networks can serve as neural controllers. NEAT is an evolutionary algorithm that can evolve such neural controllers for a given application. This algorithm requires a great amount of simulations. The simulation power of neuromorphic hardware such as SpiNNaker can be harvested to perform these evaluations in parallel and in real-time. Running the same SNN simulations on standard computers would be much less efficient.

The experiments presented here deliver a proof-of-concept, showing that the combination of neuro-evolution and neuromorphic hardware can produce functional networks for the XOR benchmark problem and the more complex problem of the Pac-Man arcade game.

The experimental set-up could be easily adopted for other I/O applications of similar complexity. Furthermore, many improvements are available that could boost the performance of this approach and could allow more complex problems to be solved. For instance, an online learning rule like Spike Time Dependent Plasticity can be implemented in SpiNNaker SNN simulations [26], or a more powerful version of the NEAT algorithm that exploits geometrical information of the problem domain (HyperNEAT [27]) could be used.

Acknowledgment

This work was supported by the EU 7th Framework Programme (FP7/2007-2013) under grant agreement no 604102 The Human Brain Project.

References

[1] W. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, pp. 115-133, 1943.

[2] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain", *Psychological Review*, vol. 65, pp. 386-408, 1958.

[3] W. Maass, "Networks of Spiking Neurons: The Third Generation of Neural Network Models", *Neural Networks*, vol. 10, pp. 1659-1671, 1997.

[4] K. Hornik, "Approximation Capabilities of Multilayer Feedforward Networks", *Neural Networks*, vol. 4, pp. 251-257, 1991.

[5] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview", *Neural Networks*, vol 61., pp. 85-117, 2015.

[6] Y. LeCun, Y. Bengio and G. Hinton, "Deep Learning", *Nature*, vol. 521, pp. 436-444, 2015.

[7] F. Walter, F. Röhrbein and A. Knoll, "Computation by Time", *Neural Processing Letters*, pp. 1-22, 2015.

[8] W. Gerstner and W. Kistler, "Spiking Neuron Models: Single Neurons, Populations, Plasticity", *Cambridge University Press*, Cambridge, U.K. and New York, 2002.

[9] A. Hodgkin and A. Huxley, "A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve", *The Journal of Physiology*, vol. 117, pp. 500-544, 1952.

[10] S. Furber, F. Galluppi, S. Temple, and L. Plana, "The SpiNNaker project", *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652-665, 2014.

[11] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the SpiNNaker system architecture", *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454-2467, 2013.

[12] E. Painkras, L. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. Lester, A. Brown, and S. Furber, "SpiNNaker: A 1 W 18-core system-on-Chip for massively- parallel neural network simulation", *IEEE J. Solid-State Circuits*, vol. 48, no. 8, pp. 1943-1953, 2013.

[13] M. Mahowald, "An Analog VLSI System for Stereoscopic Vision", Boston, USA, 1994.

[14] S. Jang, J. Yoon, S. Cho, "Optimal Strategy Selection of Non-Player Character on Real Time Strategy Game Using a Speciated Evolutionary Algorithm", *IEEE Symposium on Computational Intelligence and Games*, Politecnico di Milano, Italy, pp. 75-79, 2009.

[15] K. Stanley, N. Kohl, R. Sherony, R. Miikkulainen, "Neuroevolution of an Automobile Crash Warning System", *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, Washington, DC, USA, pp. 1977-1984, 2005.

[16] K. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies", *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, 2002.

[17] D. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization", *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 148-154, San Francisco, USA, 1987.

[18] H. Lipson and J. Pollack, "Automatic design and manufacture of robotic lifeforms", *Nature*, vol. 406, pp. 974-978, 2000.

[19] V. Valsalam, J. Hiller, R. MacCurdy, H. Lipson, and R. Miikkulainen, "Constructing controllers for physical multilegged robots using the enso neuroevolution approach", *Evolutionary Intelligence*, vol. 5, pp. 1-12, 2012.

[20] J. Clune, K. Stanley, R. Pennock and C. Ofria, "On the Performance of Indirect Encoding Across the Continuum of Regularity", *IEEE Transactions on Evolutionary Computations*, vol. 15, pp. 346-367, 2011.

[21] J. Clune, B. Beckmann, C. Ofria and R. Pennock, "Evolving coordinated quadruped gaits with the HyperNEAT generative encoding", *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, pp. 2764-2771, Piscataway, NJ, USA, 2009.

[22] J. Gauci and K. Stanley, "A case study on the critical role of geometric regularity in machine learning", *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 2008.

[23] P. Verbancsics and K. Stanley, "Evolving Static Representations for Task Transfer" *J. Machine Learning Research*, vol. 11, pp. 1737-1769, 2010.

[24] 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.

[25] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet and P. Yger, "PyNN: a common interface for neuronal network simulators", *Frontiers in Neuroinformatics*, vol.2, pp. 1-10, 2009.

[26] F. Walter, F. Röhrbein and A. Knoll, "Neuromorphic Implementations of Neurobiological Learning Algorithms for Spiking Neural Networks", *Neural Networks*, vol. 72, pp. 152-167, 2015.

[27] J. Gauci and K. Stanley, "Autonomous evolution of topographic regularities in artificial neural networks", *Neural Computation*, vol. 22, pp. 1860-1898, 2010.