Proceedings of the International Multiconference on Computer Science and Information Technology pp. 89–98
ISBN 978-83-60810-22-4
ISSN 1896-7094

EVM: Lifelong reinforcement and self-learning

Mariusz Nowostawski
Information Science Department
Otago University
PO Box 56
Dunedin, New Zealand
Email: mariusz@nowostawski.org

***Abstract*—Open-ended systems and unknown dynamical environments present challenges to the traditional machine learning systems, and in many cases traditional methods are not applicable. Lifelong reinforcement learning is a special case of dynamic (process-oriented) reinforcement learning. Multi-task learning is a methodology that exploits similarities and patterns across multiple tasks. Both can be successfully used for open-ended systems and automated learning in unknown environments. Due to its unique characteristics, lifelong reinforcement presents both challenges and potential capabilities that go beyond traditional reinforcement learning methods. In this article, we present the basic notions of lifelong reinforcement learning, introduce the main methodologies, applications and challenges. We also introduce a new model of lifelong reinforcement based on the Evolvable Virtual Machine architecture (EVM).**

I. Introduction

IN THIS article we re-asses the state of reinforcement learning methodologies in modern AI in the context of learning in open-ended and unknown environments. We first analyse the biological aspects of reinforcement learning, discuss the basic concepts of conditioning and so called Law of Effect. Then we focus on the mechanisms of reinforcement itself – how it operates in biological systems. We approach the state of the reinforcement learning in the computational domains from the previously developed biological perspective. Later, we introduce the EVM architecture. Then we define a machine learning formalism, and based on it we define a truly self-learning system class. We discuss what properties truly self-learning systems exhibits, and how appropriate search bias can be obtained through the learning process itself. First we introduce and then discuss the relationship between two self-learning systems: EIRA and the EVM. We conclude the article with short summary and plans for future work.

II. Biological learning

Most of the modern theory of learning originated in the work of Watson [28] and then Pavlov [16]. The main ideas are the notion of *conditioning*, and *conditioned stimulus* with a *conditioned response*. In those models, a reward system is a fundamental element. The basic idea is that positive feedback is an essential part of conditioning, i.e. given a conditioned stimulus positive feedback is employed to achieve the conditioned response. In other words, when a positive reward accompanies a particular response to a stimulus over many repetitions, then that particular response will become conditional to the stimulus. In experimental studies, where conditioning trials were not accompanied with positive feedback (rewards), the expected conditioning was not achieved. The early work on biological learning and conditioning perceived (or projected) the living organisms as structured mechanistic systems. Contemporary biology and psychology departs from this simplistic model and instead, treats associative processes as being fundamental to all learning. In contemporary studies on conditioning, the exclusively mechanistic view is being redefined to account for richer and more complex dependencies. These dependencies between the stimulus, conditioning and positive and negative feedback loops can form a highly complex, interlinked, and hierarchical network of dependencies. It is important to note, that the role of a (positive and negative) feedback continues to play the central role in contemporary theories of learning as it did at the beginning of 20^{th} century.

The reward system in many biological systems is based on binary (all or nothing) signalling. A subject receives the full reward if the expected conditioned response is achieved, or otherwise, there is no reward at all. There is no concept of partial reward for partial conditioned responses. For example, an animal reaches for food/ catches a prey or it does not. WIthout food, there is no positive feedback. It is important to distinguish this type of learning, from other machine learning techniques where the feedback mechanism is provided in a form of a gradient, i.e. there are partial rewards for partial responses (partial solutions).

Law of Effect. The *Law of Effect* [23] tries to capture the idea that those aspects of behaviour which satisfy certain needs of an organism will tend to be repeated, whereas those which do not satisfy any needs will tend to be omitted from future behaviour. Intuitively, it is clear that something like the *Law of Effect* must operate on different levels of biological organisation to facilitate the process of learning and adaptation. The *reinforcement*, both, positive through rewards and negatively through punishments is necessary for the process of learning to occur. In the context of biological systems, the question is which features of the environment function as positive and which as negative reinforcers. The difficulty of setting up classes of reinforcers is mainly that a reinforcement is dependent on the context in which it operates. The reinforcement in one context will not be the same in another. In fact, positive reinforcement in one context may have exactly

opposite effect in another context. For example, when hungry, the smell of food provides positive reinforcement and increases subject's desire to eat. However, when a subject is full and satisfied, the smell of food may have quite the opposite effect. Detailed discussion on biological aspects of reinforcement in the context of cybernetics is provided by George [4].

Natural reinforcers work subject to their physical context. They also operate over time delays. Often, a reinforcement (in the form of a reward or punishment) does not occur immediately after a cause. Both, negative feedback and positive reinforcement can take substantial amounts of time to occur. The positive (or negative) feedback, may be immediately proceeded by events that have no causal relationship to the reinforcement. For example, a dog might have chewed the owner's shoes after which he went in front of the house and lay down for some time. Once the owner notices the damage the dog makes, and punishes the dog, it may not be apparent for the dog (properly associated) what the reinforcement is actually for (chewing on the shoes or lying in front of the house). Biological organisms are surprisingly well-attuned in selecting the appropriate causal relationships and filtering out the events that do not have causal relationship to the rewards (or punishments). Animals and humans usually infer (most of the time without any cognitive or conscious reasoning) the correct context from cues and regularities of the environment and the reinforcement itself.

Short term vs. long term reinforcers. Sometimes, short-term positive (or negative) reinforcement is cancelled out by a long term negative (or positive) reinforcement. Biological organisms seem to have little trouble learning the best long-term strategy that maximises the overall positive feedback. Howeer, there are certain exceptions. Sometimes, organisms seem to be wired in such a way that short-term positive reinforcement is so strong that it overrides any long-term negative reinforcement that the organism may experience. In extreme cases, such a short-term override may eventually lead to the death of a biological organism. An example of such situation is studied for example through the intracranial self-stimulation (ICS) experiments [2]. In a typical ICS experiment subjects (e.g. rats or monkeys) are taught to activate self-stimulation (for example dopamine or cocaine stimulation) directly to their brains through manipulation of certain physical objects (such as a switch or a button). After initial training, subjects are incapable of breaking the cycle and continue to perpetually activate the self-pleasuring stimuli, paying no attention to food or any other survival activities. The short-term reinforcement is so strong, that the subjects can ultimately pleasure themselves to death [2].

In a majority of circumstances, however, the short and long term reinforcers work together and complement each other. Organisms' adaptation can be viewed as a balancing act between different, sometimes conflicting, reinforcers.

Based on the above discussion, a complex picture of biological learning systems emerges. We will highlight below some of the main points.

- Reinforcement is a key element to biological learning.
- Reinforcement works often as a binary system; a given reinforcement is present, or not. In biological systems there is often no gradual or partial negative or positive reinforcement.
- Reinforcement is often not immediate; it frequently follows the triggering organism actions with a delay.
- Reinforcement is context-dependant; the same stimuli can provide opposite or contrasting reinforcement in different contexts.
- The same stimuli can have opposite short-term and long-term effects, hence short-term negative reinforcement can lead to a long-term positive feedback, and vice verse.
- Generalisation skills and lifelong learning appears to be essential in coping with the complexities of the dynamical environments and to facilitate the process of biological learning.

Learning how to learn. There are many open questions in the area of biological learning. The emerging theories and overall picture of biological learning appears as a complex, highly dynamic network of interlinked dependencies. We perceive the learning process occurring simultaneously at different levels of biological organisation, e.g. both on the base level (to learn the immediate tasks at hand) and at the meta-level – to learn the abstractions, generalisations and biases that can be transferred, directly or indirectly, to enhance one's abilities to learn new tasks. In general, learning how to learn in biological systems appears to be important yet quite an unknown phenomenon.

Binary reinforcement, lifelong learning, and meta-learning – these are the three fundamental properties that our EVM model tries to investigate and exploit through:

- a hierarchically organised computing architecture;
- reflection and reification mechanisms allowing the modification of one's own internal processing (learning);
- keeping the history of earlier *discoveries* (memory);
- reusing learned skills in different contexts (exaptation);
- the ability to deal with delayed rewards/punishments;
- the ability to generalise previous experiences;
- meta-learning constructs built-in into the architecture.

III. Computational learning

A large part of computer science in the context of AI is concerned with the optimisation, prediction and the use of heuristics to solve problems by means of computational learning systems. In this work, we are concerned exclusively with a small subset of machine learning inspired by highly dynamic biological learning systems that exhibit lifelong learning process. These reinforcement learning systems form a hierarchical short and long-term reward/punishment structures. We will briefly review the state of the art through discussion and analysis of various existing techniques.

Early reinforcement learning. Reinforcement learning concerns agents that sense and act in their environments, trying to learn to choose optimal actions to achieve a maximisation of the reward intake. Each time an agent performs an action, a trainer may provide a reward or penalty to indicate the

desirability of the resulting state. The task of the agent is to learn from the, possibly delayed, rewards a sequences of actions that produces the greatest cumulative reward intake rate.

The study of reinforcement machine learning is as old as the theory of algorithmic computation itself. In his seminal article, Turing [26] studied a special type of unorganised machines (u-machines) that to a certain extent can be seen as precursors of contemporary evolutionary and computational learning systems. He advised three types of unorganised machines: A-type, B-type and P-type. These machines consist of randomly connected two-state machines whose operation is synchronised by means of a central digital clock. By the application of *"appropriate interference, mimicking education"* a B-type machine can be trained to *"do any required job, given sufficient time and provided the number of units is sufficient"* [26]. His P-type unorganised machines, have *"only two interfering inputs, one for 'pleasure' or 'reward' [...] and the other for 'pain' or 'punishment' "*. Turing studied P-types in the hope of discovering training procedures *analogous to the kind of process by which a child would really be taught*. Later, Turing said: *I have done some [...] experiments with one such child-machine, and succeeded in teaching it a few things.*

The foundations for many machine learning techniques have been then laid by John Holland, [7], [8], who provided initial mechanisms for many machine learning techniques including early reinforcement learning algorithms and evolutionary computation.

Formally, we model the state of the environment (and the agent) in time t as $s_t \in S$, where S is the entire possible state space. An agent performs actions $a_t \in A$ that generate the next state s_{t+1} through the state transition function $\delta(s_t, a_t)$. All the future states $s_{t+2}, s_{t+3}, \ldots$ can be influenced by a given action a_t. The reward function of the state of the environment, in the general case, $r(s_t, a_t) \in R$ can vary in time, too. The task of the agent is to choose such a sequence of actions a_t that will maximise its reward intake over time. This can be expressed by different objective functions. The two most common are:

1) maximisation of discounted cumulative reward, $\max \sum_{i=0}^{\infty} \gamma^i r_{t+i}$, where γ is a parameter from the range $[0, 1]$ representing the memory of the previously collected rewards – the larger the γ, the longer the system *remembers* previously achieved rewards. For small γ the older rewards influence the cumulative score less and less.
2) maximisation of average reward, $\max \lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+1}$.

More details related to reinforcement learning can be found in the literature, for example [12], pages 367–387.

IV. Lifelong reinforcement learning

The reinforcement learning methodology is applied within a context of autonomous intelligent agents and multi-agent systems. Agents, (e.g. robots), try to operate in complex, unknown dynamical environments. Such environments require that an agent learns several related tasks within the same environment, using the same sensors and actuators. There is no possibility of *re-setting* the agent state to the "starting point" – the agent must continuously sense, learn and act. The agent may in fact never revisit any of its previous states in its entire lifetime. This is different from some learning methods, such as, for example, genetic algorithms, which generate a hypothesis to be tested on a single isolated and idealised testcase. After that, a new hypothesis is tested on exactly the same isolated test case, again, and again. In lifelong learning this is not possible – any given (new) hypothesis cannot be tested on any of the old isolated and idealised test cases, because there is no possibility of reverting the environment precisely to any of its previous states. One may say that in lifelong reinforcement learning, the time flows in one direction. There is no possibility of *resetting* the clock. Mobile robots cannot test a hypothesis and move from a given location only to restart again with a different hypothesis from the same location again. Once a decision has been taken, the situation has changed, and new conditions need to be taken into account, a new environment state must be sensed, and new decision needs to be taken again.

Multi-task systems. We will use the terms *learning tasks* or *solving tasks/problems* interchangeably. Tasks that need to be solved can be provided to the system sequentially [20], [22], or the tasks can be provided all at once, in a group. When tasks are provided in sequence, it is referred as *incremental learning*. Incremental learning takes advantage of the fact that earlier tasks (typically easier) can provide beneficial biases towards solving/learning later, usually harder, tasks. It is possible for incremental learning to use binary reinforcement mechanisms.

Multi-task learning is an area of machine learning which studies methods that can take advantage of previously learned knowledge by generalising and reusing it in the course of solving a set of possibly related tasks. This is closely related to the notion of *incremental learning*. Incremental learning is a special case of multi-task learning, where incrementally more difficult tasks are provided to the system sequentially, in such a way that the system can tackle easier tasks first and gradually solve more and more difficult tasks as time goes on. Multi-task learning is a more general term, where the difficulty of individual tasks may or may not be of concern. There is research work conducted within the area of multi-task learning applied to various domains and experimental settings and employing different learning mechanisms, eg. [29]. If the tasks share some similar internal structure, the learner may exploit these regularities and find it easier to learn them together rather than in isolation [24], [25]. Caruana and Baxter demonstrated that learning multiple tasks within an environment of related tasks can potentially give much better generalisation than learning a single task alone [3], [1].

Meta-learning [5] is a relatively new area of research within the machine learning and data mining fields. Researchers within the field try to understand the process of exploiting knowledge about learning how to learn. The goal is to build algorithms that can improve the performance of the learning process itself. Recent advances provide the foundations for constructing meta-learning assistants and task-

adaptive learners. One of the main motivations is the fact that successful applications of complex software systems in the real-world require a continuous adaptation to new needs, changing requirements and changing contexts. When a given model fails to perform efficiently at a given time, one would expect the learning mechanism itself to re-learn, taking into account previously learned experiences [5]. Meta-learning capabilities are necessary in exploiting cumulative expertise gained from previous experiences. This is particularly evident in biological learning and in artificial multi-task learning systems.

One of the contributions that inspired the EVM architecture is the work related to the Optimal Ordered Problem Solver, OOPS [22]. OOPS is based on the sequential exploration of program search spaces associated with Levin's universal search algorithm [11]. OOPS uses self-delimiting binary programs[1] and explores the space of programs by trying to find one that provides a solution to a given target set of problems. Each new problem is provided to the system only after the previously provided problem has been successfully solved. Each solution found (to any of the previously solved tasks) is stored in the system's storage. Schmidhuber work extends earlier work on bias-optimal search algorithms done by Hutter [9] (for bias-learning see also [12]). Inherently, such search must deal with the trade-off between:

- **exploration**, that is the search for completely new programs; the search starts from an empty program, and explores all the possible programs from scratch;
- **exploitation**, search for variant solutions; the search tries to reuse already explored subspaces of programs, and use some (or all) of the already gained experiences.

The assumption here is that exploiting experiences collected in previous searches can solve the target problem faster. This is because the learning process can exploit any regularities that have been discovered earlier. In the context of OOPS, earlier discovered programs are stored and they provide (partial) solutions to other problems. In OOPS, meta-knowledge is stored in the form of candidate program solutions. The system uses this information and tries to exploit it for incremental self-improvement. For example, the task of performing long addition and multiplication[2] can be split into individual tasks concerned with additions, and tasks concerned with multiplications. Programs capable of addition and programs capable of multiplications are then stored in the OOPS framework and are used as first-order primitives when searching for solutions to consecutive problems. An important difference between EVM and OOPS is the fact that newly discovered programs in OOPS are simply added into the primitive instruction set. This, for a large number of sub-tasks typical in long lifelong reinforcement learning will lead to an exponential growth of the program search space, due to the increase in the instruction set. In the EVM this has been addressed through the notion of computational levels. Newly discovered instructions can be added to the primitive level, just as in OOPS. But, they can also be added to new levels created and managed by the EVM itself. Unlike OOPS, the meta-management of instructions allows the EVM to manage the complexity and trim the program search space. The details of EVM architecture will be provided later in this article.

In the context of meta-learning and exploration vs. exploitation, one possible approach is to split the search process into two phases, as in [21]. One part of the search process tries to exploit already built (partial) solutions, and performs traditional exploration/exploitation of the search space. At the same time, the other part of the search process, concurrently, explores the program generators space (explores the meta-search). For simple tasks the former will be more efficient and will find a solution faster. For complex tasks, the time wasted on traditional search (not the meta-search) is insignificantly small compared to the time that meta-search techniques take; therefore no substantial inefficiency can be expected. Other improvements on this scheme may be possible in a specific subclass of computational tasks. Such improvements must be incorporated into the search process in an automated and adaptable way in order to exploit the underlying regularities of the problem, and the search process itself.

Applications of multi-task learning. From a practical perspective there are many problem domains that can be viewed as sets of related tasks. For example, speech recognition may be decomposed along many different axes: words, speakers, accents, etc. Face recognition represents a potentially infinite domain of related tasks. Medical diagnosis and prognosis problems using the same pathology tests are yet another example, as well as time-series prediction, database mining, autonomous robots, personal software assistants, etc [1]. See also [3] for more considerations about applications. Multi-task learning techniques have shown promising results when applied to artificial neural networks [3], [24]. In these models neural nets that learn different tasks usually share a common hidden layer that successfully exploits regularities and common properties across multiple tasks. In most of the systems, however, multiple-task problems are usually translated into single-task problems. A potentially rich source of information about the problem is then omitted and the possible benefits of multi-task learning and meta-learning is lost. The main advantage of multi-task and meta-learning systems is the ability to reuse previously acquired knowledge and to exploit

[1]Such programs are binary strings with a specific length encoded into the program itself, or with special end-of-program markers, that identify uniquely an end of a given program in a binary sequence. Note that a given program may end anywhere in, potentially, an infinite string of bits. This is in contrast with ordinary binary programs, which start at the beginning of a bit string and end at the end of the bit string. The advantage of self-delimiting programs is that they can be spliced together and such splicing will not change their semantics.

[2]Long arithmetic operation means an operation that spans numbers longer than what the single arithmetic unit can handle in a single operation. Consider an arithmetic unit that can handle a 1-digit binary operation of subtraction and addition with a single extra carry bit. If one now tries to add 8-bit long words, the operation would be considered long addition, and the arithmetic operation would need to be chained and performed multiple times on the simple 1-digit arithmetic unit. It is similar to adding for example two 25-digit long decimal numbers with a piece of paper. One needs to understand the addition up to number 19 (adding two arbitrary decimal digits) to be able to perform long addition on arbitrary long decimal numbers, with a pencil and piece of paper.

any regularities across different tasks that can be generalised. Solutions, or partial solutions of previously solved problems, can be valuable in solving new, unsolved (and often more difficult) tasks. The system is designed to shift its *bias* accordingly to search for a hypothesis space that contains good solutions to many of the problems in the environment. By transferring knowledge across related learning tasks, a learner becomes more experienced and generalises better.

V. EVM OVERVIEW

The Evolvable Virtual Machine architecture (EVM) [15], [14], [13] is a computing architecture based on the notion of distributed interactive asynchronously communicating computational cells. The EVM provides a massively decentralised and distributed asynchronous framework for experimenting with, and studying the properties of lifelong self-learning systems. It can be used for distributed multi-task learning [3], [17], for automated program discovery [20], and for lifelong reinforcement learning. The main emphasis is on initial bias-free state that requires a meta-learning that through reinforcement will diverge the random search into more efficient learning strategy for a given problem domain.

The Evolvable Virtual Machine consists of two parts. The abstract layer, called EVMA (Evolvable Virtual Machine Architecture). And the concrete instantiation, called EVMI (Evolvable Virtual Machine Instantiation). The full description of the EVMA/EVMI together with a formalisms has been presented in [13].

The current EVMI provides a framework to experiment with and instantiate EVMA models. The current EVMI supports most but not all of the EVMA features. The architecture consists of independently operating computational *cells*. The cells operate in a local environment where they provide their results and obtain their feedback. A computational entity, a cell, acts (operates) in an unknown environment, trying to maximise its own reward intake. The cell continuously adjusts its actions in such a way as to collect more positive and avoid negative feedback. The rewards (positive reinforcement) and punishments (negative reinforcement) can be delayed in time and do not necessarily correlate directly with the last actions of the cell. As in Q-learning [12, p. 367–387], the actual real reinforcement mechanism is unknown and can be subject to dynamic changes.

The cell computations consume the resources that the cell has been initiated with. As long as the cell's reward mechanisms refuel the resources the cell continues to perform its activities, and the *task* is considered solved within the EVMI context. The EVMI approximates an unknown, infinitely long, computational solution to the task by a temporal snapshot that has been computed up to a specified time mark. The task is considered failed or unsolved when the activities of the cell do not bring the required reward intake for the cell, and cell's computational expenditure cannot be balanced out with the cell's reward intake. In our experiments we have used two types of situations:

- there is periodic feedback to the cell, with rewards, or punishments, or both (positive and negative reinforcement). The cell operates in an internal loop mode for an extended amount of time. In this case the cell's state trajectory is evolving in the state space in a more or less continuous fashion.
- there is only one reward feedback provided after all the activities of the cell have ceased. In this case the cell's state trajectory has been clearly divided into generations. After each activity-reward feedback cycle, the cell's state has been reset to the initial state. Note, that the environmental state is not being reset to its initial state, but continues its own evolution in a continuous fashion. This results in a complex dynamics between internal and external activities of the cells.

Cells. Computational cells are organised into a regular lattice (or grid). Neighbourhood is typically 4 or 8 adjacent cells, but, this can be programmatically controlled. Various topologies are possible, although in this article we only focus on regular grids. EVMI cells are constrained in such a way that an externally provided resource (reward) is necessary for the cells to continue to exist (or to be allowed to perform their activities). In other words the cells' computational resources (memory and CPU cycles) are abstracted into a single parameter, called a *resource*, which again, is balanced by a single parameter *reward*. This is an external system-level constraint that may not have any direct linkage with the problem domain of any of the tasks. It can be treated simply as an *organising principle* [6]. It is a domain-independent artifact that models certain constraints that are conceptually (metaphorically) equivalent to physical constraints. This physical constraints are modelled through the abstract notions of a *resource* and *reward*.

Intra-cell computing. A cell's program takes data from and produces output to the *environment*. In the abstract architecture these mechanisms can be arbitrary, and the EVMA does not mandate any particular mechanism. For efficiency reasons however, we have designed and used a number of fixed mechanisms for the cell processing. Therefore, each cell's specialisation and program execution mechanism was built-in, and could not be changed by a given cell itself. In the abstract sense, these could have been reified and provided as mechanisms on the base level, subject to direct manipulation, degradation and improvement when needed. Experiments with meta learning proved to be prohibitively slow.

At every iteration, the execution engine tries to solve some tasks provided by the environment. For each task, it runs the cell's program (possibly calling some neighbours' programs) in interaction with the task's resource. If the output is correct, the cell receives some rewards. A specialisation mechanism uses these rewards to modify its state and provide a new program for the next iterative step. The *specialisation mechanism* is an umbrella term that relates to the ability of the cell to trim its computational capabilities (that means some of the computational capabilities are removed from the cell repertoire).

Individual cells work in a distributed and asynchronous fashion. The processing of an individual cell is sequential and consists of sets of operations, also called *iterations*. The iterations can be imposed by the environment (as it was the case with some of our experiments), can be managed by the cell and cell program alone, or they can be in an arbitrary order (subject to a mixture of environmental and cellular influences).

At every iteration, the specialisation mechanism of the cell must generate one program. This program will try to solve some tasks from the environment, and this process may yield some rewards coming from the environment. From an artificial life perspective, the program attempts to maximise the reward and maintain its existence by trying to achieve homeostasis with regard to the external resources (rewards)[3].

Environment and resources. Every cell maintains a program[4]. The cell's *goal* is to find a successful program: one that, by solving a task, yields enough rewards for the cell to survive. Programs can call other programs.

When a given program obtains a reward, the reward will be shared proportionally with any programs used as *assistants* to compute the solution. All of the participants will benefit from their relationship. In other words, symbiotic (mutualistic or parasitic) relationships will appear between programs. This ability to access other programs has thus facilitated complex hierarchical organisation and self-assembly. As a consequence, cells are able to collaborate to solve complex problems. Problems that none of the cells would be able to solve on their own can thereby be solved through cell collaboration.

Figure 1 depicts a simple example of cell C_1 using other cells during the course of C_1's computation. The computation starts with C_1 executing its own instruction number 1. The second instruction of C_1 calls its right neighbour, the cell C_3. Therefore the computation follows to the first instruction of the cell C_3. C_3's program is executed up to instruction number 2, and the third instruction of C_3 calls the program of upper neighbour, cell C_5. And so on. On the bottom of the figure you may see the actual order of instructions being executed and where they belong across all cells. This type of hierarchical assembly between the cells is distinct from the hierarchy on the virtual machine level, which is supported by the EVM assembly language. This multi-level machine hierarchy (as opposed to cell's hierarchy) is depicted on Figure 2. Note that the Current Machine (CM) index and Higher Machine (HM) index may point to various parts of the HM list.

The environment represents the external constraints on the system. Its role is to keep the system under pressure to force it to solve the tasks specified from outside. We have designed the environment as a set of *resources*. There is a one-to-one mapping between the resources and the tasks to solve (every resource corresponds to a task). The purpose of these resources is to give rewards to the cells when they solve their task.

[3]In terms of *trying* to achieve homeostasis, the homeostasis is either achieved or not, and the cells that achieve it persist in the system and those that fail are removed and replaced by other cells. Therefore, it appears as if the cells are trying to achieve homeostasis, where in fact they merely follow their own deterministic computational trajectories. The use of intentionality and causality helps in discussing and explaining the dynamics of the system.

[4]A program in this case means simply a sequence of instructions in the EVM assembly language.

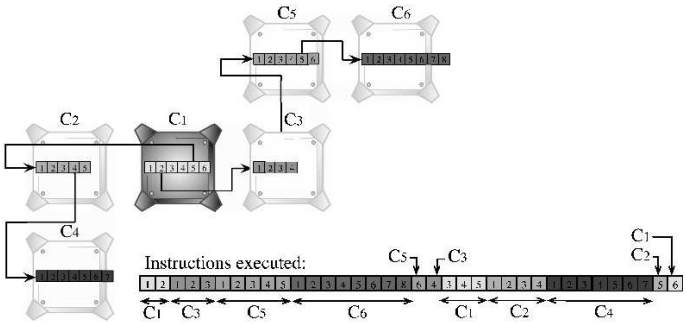

Fig. 1. The dark cell executes its program. Arrows show instructions that call neighbours' programs.

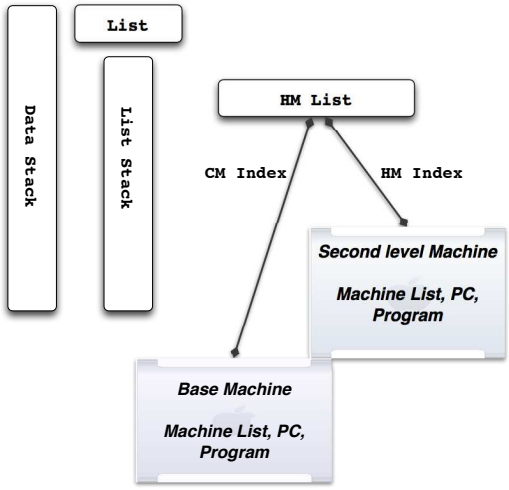

Fig. 2. EVM hierarchy. The HM (Higher Machine) List stores indices to all created and managed machines. The CM (Current Machine) pointer points to the currently executing machine, and the HM index points to the higher machine that can be manipulated by the EVM instructions.

Cell specialisation consists of trimming the cell computational capabilties. The cell self-adapts to a particular task in the environment. Several specialisation mechanisms have been studied: classic genetic algorithms, ad hoc stochastic search (maintaining a tree of probabilities of potential building blocks), or an adaptation of an environment-independent reinforcement learning method (proposed by [19]). Different methods of learning have been experimented with, including random search, stochastic search and genetic algorithms. The results of our experiments have been published previously [14].

VI. Self-learning system

One of the goals of AI learning algorithms in general is to provide efficient solutions to a wide class of problems. This means that the selection of instruction sets, the algorithm itself, and any information that is fed into the algorithm is chosen specifically to aid the goal. To make the search easier. To make the search biased toward a particular problem at hand. The

biases, the rules that can guide code synthesis, are essential in any non-trivial computational search process.

The goal was to develop an adaptable, self-regulating system, in which the *expert rules* and *search bias* are not given to the system *ab initio*. We expect any search bias to be acquired on-the-fly, during runtime, based on the binary feedback the system obtains from the outside environment. In other words, the biases and the expert rules are obtained through a series of test cases, and should not be built into the actual learning algorithm as such. It is our aim to provide an architecture that facilitates investigations into different possible mechanisms of code synthesis.

In the EVM architecture we use the concepts of symbiosis and exaptation augmented with mechanisms of bias-optimal search techniques to build the necessary structures and learning algorithm based on the past history of the system itself. The design and implementation of the basic EVM is a step towards the ultimate goal of a self-learning artificial (computational) system. By *truly self-learning* system, we mean an abstracted learning system inspired by biological theory of reinforcement and conditioning, in which no prior bias or knowledge is present. In particular, we stress that the feedback (positive and negative) is (a) provided in binary form only, (b) it is delayed, and (c) context sensitive, and the system starts its learning process without any inductive bias. What it means is that our search will (hopefully only initially) be no different to random search (or enumeration). However, as time progresses, the system should be capable of building the bias based on the learning process itself. Note that this formulation of *self-learning system* departs from the traditional machine learning systems, where the inductive bias is introduced *ab initio* through various elaborate mechanisms.

To express the notion of *self-learning* more clearly, let us consider the following abstract architecture, consisting of four elements:

- code generator (C) – this is an active unit of computation that takes feedback stream as an input and generates the program stream on its output;
- program stream (P) – this is a sequence of programs that is being tested by an environment; the environment then generates the feedback stream;
- feedback stream (F) – this is an outcome of evaluating programs in a given environment;
- environment (E) – this an active unit that takes program stream as its input, and generates feedback stream as its output.

In a truly self-learning system, the last component, the environment E, is (by definition) unknown *a priori*. E may be computable, in which case the sole purpose of the learning system C is to learn (discover) the functional mapping of the environment, to discover the function E – i.e. the functional mapping between the programs fed on environment input, and the feedback read on its output. Even if E is ultimately computable, it may be impractical to discover the actual computation of E due to computational limitations of the learning computation itself (limitation of C). In that case, the learning system might try to approximate true E by E_{approx}. Note, on more philosophical grounds, E may be ultimately uncomputable (or unknowable, or undefinable), in which case the purpose of the learning system C is to approximate the unknown/uncomputable environmental E to the desired level of accuracy, using a given model of computation C that the learning system is using.

In soft computing and machine learning systems it is often the case that both, C and E_0 (initial environment) are given in the form of computable functions. The task of the learning system is then not to discover an unknown environmental function E, but to convert a given form of the function E_0 into a different form, e.g. E_1. In such a case, a purpose of such (somewhat incorrectly called *"learning"*) system is not to learn E, but to (simply) translate a given computation E_0 into a different computation E_1[5].

Based on this formulation, most of the mainstream evolutionary computation methods are not truly self-learning. For example, genetic algorithms [27] (and related models, such as original Koza's tree-based genetic programming [10]) can be seen as an automated computational method of re-coding one solution (E_0) to a problem provided by an expert, into another solution (E_1), discovered by the search mechanism. The solution provided by an expert (E_0) is in a form of phenotype/genotype encodings and fitness function evaluation mechanisms. The search process C is, in most cases, (extremely) *encoding-sensitive*. Some expert formulations of a problem (a particular solution encoding and fitness function) may lead to fast, effective and efficient re-coding by GAs for a specific problem ($E_0 \rightarrow E_1$). Some, on the other hand, may fail to provide a re-coding at all ($E_0 \rightarrow unknown$).

Note, that, although the EVM belongs the category of truly self-learning systems, for any practical reasons it would be infeasible to start each time with an empty system without any inductive bias whatsoever. Rather, in practical applications, the EVM system is seeded with previously solved problems and does in fact re-uses previously gathered learning biases, based on a choice of particular C. Unlike GA, where fitness function, crossover, mating etc., are all shared among all different search domains, and the expert bias is encoded almost entirely all within the E, in the EVM the expert bias is entirely encoded in C itself, and E remains to be discovered. The crucial difference between truly self-learning and typical machine learning systems lies in the adaptive ability to modify pre-existing search biases of C during the evolution of the learning process itself (E discovery process).

One example of a set of truly self-learning algorithms that has inspired some of the characteristics of the EVM is the environment-independent reinforcement acceleration (EIRA) approach proposed by Schmidhuber.

[5]There are many different reasons why such a transformation is useful: consider for example two computations computing the number Pi up to 2 million decimal places. One computation is expressed by a short program, but takes many iterations to calculate the expansion. The other computation is expressed as a long (over 2 million bytes) lookup that takes just one instruction cycle to execute.

VII. Environment-Independent Reinforcement Acceleration – EIRA

Environment-Independent Reinforcement Acceleration (EIRA) is a reinforcement learning mechanism proposed by Schmidhuber [18], [19]. EIRA is an environment-independent, single-life (lifelong) reinforcement learning mechanism that guarantees lifelong performance improvement. EIRA is an example of a truly self-learning system: there is no initial human-expert bias provided initially to the system. The system will learn all biases exclusively from the binary feedback mechanism. The executing program monitors its own performance, and it continuously tries different learning policies. Some of the policies that perform worse than any other are discarded, and the system keeps only the *useful* policies: those that bring more reward per time unit than all the previously tested policies. One of the objectives in designing EVM assembly language was to provide easy mechanisms for plug-in reinforcement learning techniques for lifelong continuous adaptation and performance improvement. The built-in probability distribution manipulation instructions are inspired by EIRA. The EVMA can be seen as a super set of EIRA conceptual learning mechanism. It is important to remember that EVMA extends the learning mechanisms by meta-learning strategies that can work concurrently within the same EVMA grid together with EIRA. Unlike EIRA, EVMA also provides specialised mechanisms for dynamic updating (forgetting) previously learned or established patterns and program structures.

Reinforcement Acceleration Criterion (RAC). In EIRA, a system policy is an arbitrary modifiable algorithm that maps environmental inputs and internal states to outputs and new internal states. In dynamic and unknown environments in the real world, each *policy modification process* (PMP) occurring during system lifetime may have unpredictable influence on environmental states, rewards, and PMPs at later times. Note that this relates to the notion of multiple interactive machines.

Schmidhuber proposed the *reinforcement acceleration criterion* (RAC) as a way of measuring performance improvements. At a given time, the RAC is satisfied if the beginning of each completed PMP that computed a currently valid policy modification has been followed by long-term acceleration of average reward intake. In other words, the rewards rate of a new policy must be higher than those of the previous policies. The system keeps only those probability modifications computed by useful self-modification programs: those which bring more payoff per time unit compared to all the previous self-modification programs.

At special instants, called *checkpoints*, EIRA will restore all the previous PMPs until it finds one that fulfills RAC.

There are different ways of implementing EIRA system. Here we describe one possibility to apply it to cell specialisation in the EVM system. Let every EVM cell contain one program P, that is executed in a continuous fashion (in a never-ending loop). P can call up its neighbours and during its own program execution, it can call and execute the neighbour programs too (finite number of iterations of the loop, in most cases just one iteration). P has the ability to modify itself, through self-referencing. Because P has the ability to modify the way it modifies itself (*meta-learning*), there is no intrinsic need for any other learning mechanism to be added on top, other than to provide convenient expert knowledge or initial search bias. The system is able to shift its inductive bias in a universal way. In other words, there is no explicit difference between learning, meta-learning, and other kinds of information processing. As a consequence, the system can reproduce all traditional EC mechanisms as special cases (extensions).

To achieve such a broad unification within the learning process, P needs to have special meta-instructions that modify P. Executing a meta-instruction corresponds to a PMP, as discussed above.

The checkpoints can be done periodically by the environment itself (for instance every 100 time steps, if the expected solution to the hardest task is less than 100 time steps). It is more interesting, however, to introduce a special instruction, called *checkpoint-instruction*, that triggers a checkpoint. As P can modify itself, the frequency of checkpoints is also dynamically and automatically tuned. In theory, the checkpoint frequency may well reflect the difficulty of the tasks. This mechanism will reflect also the dynamism and changeability of the environment itself.

A policy's usefulness is defined as the rewards/time ratio (or rewards rate) from the beginning of the policy. The system has to maintain a stack of PMPs. At checkpoints, it keeps only useful PMPs. A PMP is considered useful if it fulfills the RAC, i.e.:

- if there is no previous policy and the rewards rate of its policy is greater than the global rewards rate from the beginning.
- if the policy introduced by this PMP has a higher rewards rate than all the previous policies.

All PMPs are thus restored until one is found that satisfies RAC. For more detail about the implementation of that stack and efficient memory management, please refer to [19].

There are inherent benefits of lifelong reinforcement learning models such as EIRA. They mimic closely biological learning and provide an attractive theoretical and practical framework for experimentation.

EIRA is designed to make optimal use of its computational time/space resources, by exploiting *arbitrary* task-specific regularities (if there are any) [18]. These are the objectives for the EVMA, too. The EVM system should be able to:

1) develop arbitrary problem-specific representations,
2) run arbitrary learning algorithms,
3) find *good*, problem-specific learning algorithms, as quickly as possible,
4) find algorithms for finding learning algorithms.

Implementation of policies. For every instruction of P, there is a probability distribution over its possible values. That is, for every instruction of P there is an associated probability

distribution from which that instruction is being sampled. The EVMI maintains a table of probabilities as depicted in Figure 3. Values for P's instructions are selected according to their probability distribution. The meta-instructions of EVM assembly language,`incP` and `decP`, that allow PMPs, consist in increasing or decreasing entries in that table[6].

instructions of program p

possible values	0	1	2	3	4	5	6	7
add	0.9	0	0.1	0.01	0.2	0	0.8	0
incP	0.01	0.3	0.1	0.01	0.01	0	0.01	0
decP	0	0.01	0.02	0.02	0.6	0	0.01	0.1
const_2	0.05	0.5	0.04	0.01	0.1	0.1	0.05	0.7
goto	0.01	0.02	0.6	0.01	0.05	0	0.1	0.04
mul	0	0.07	0	0.2	0.01	0.9	0.02	0.03
...	...	...	...	...	...	...	...	...

Fig. 3. Policies are managed by maintaining probability distributions over the possible values for P's instructions. Instructions `incP` and `decP` modify these probability distributions. Because `incP` and `decP` can also modify their own probability, we speak here of *meta-learning*. This table is modified by the meta-instructions. At checkpoints, EIRA undoes the modifications that have not been followed by an acceleration of the rewards intake. This table, though, is never reset to its initial state.

The described mechanism expects P to have fixed length. To enable variable length programs, we introduce a special *halt* instruction that will halt an executing program. However, P may take too long to execute or it may never terminate. For practical reasons programs operate in a time-bound computational model. That means that after a predefined time-limit, the program is terminated. Further studies related to more elaborate estimations of program terminations will be a subject of further studies.

When a neighbour asks for a given cell's program, the EVM cell sends the program *without* the meta and checkpoint instructions. The outer looping instructions (if present) are also stripped out. The purpose of these special instructions is to accumulate bias information about certain instructions of the cell's program, so as to increase their probabilities of being used. The bias mechanism works exclusively on a given cell's program alone, and not on the one of its neighbours.

In our initial EVM implementation [15], we used the list-stack and all the EVM instructions to manipulate it. The list-stack simply becomes the cell's machine. That way, the program is able to generate machines of any dimensions (i.e. any number of programs and any number of instructions per program). The most interesting application are in the area of executing programs on SIMD processors, such as on the modern multicore processors and GPUs.

VIII. Summary and Future Work

This article presents the main concepts behind lifelong reinforcement learning, multi-task learning, and self-learning systems. It also provides a general description of the EVM architecture and EVM instantiation. We have discussed how the multi-cellular computational EVM model mimics biological systems and how it can be used for lifelong reinforcement learning. For detailed description of the model together with the experimental results please refer to [13].

Future work will consists of further extensions to the model. We are planning to use the EVM learning and program discovery in the context of speculative execution and code optimisation for parallel programming architectures, such as those based on Single Instruction Multiple Data. It is our belief that automated code generation might assist in more robust and flexible runtime computing environments deployed on SIMD and multicore systems.

[6]We need two meta-instructions, `incP` and `decP`, which respectively increase and decrease by δ the probability of having value v for instruction i of P. δ, v, and i are passed as arguments of `incP` and `decP`. Note that these two instructions can be collapsed into a unique one if we allowed negative increments.

References

[1] Jonathan Baxter. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

[2] M.A. Bozarth, G.J. Gerber, and R.A. Wise. Intracranial self-stimulation as a technique to study the reward properties of drugs of abuse. *Pharmacology, Biochemistry and Behaviour*, 13(Suppl I):245–247, 1980. PMID: 7195575.

[3] Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[4] F. H. George. *Cybernetics and Biology*. Oliver & Boyd, Department of Psychology, University of Bristol, first edition, 1965. Sci Q-320-GA82.

[5] Christophe Giraud-Carrier, Ricardo Vilalta, and Pavel Brazdil. Introduction to the special issue on meta-learning. *Mach. Learn.*, 54(3):187–193, 2004.

[6] Hermann Haken. *Synergetics, An Introduction: Nonequilibrium Phase Transitions and Self-Organization in Physics, Chemistry, and Biology*. Springer-Verlag, Berlin, 3^{rd} revised and enlarged edition edition, 1983.

[7] John H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. ORA Projects 01252 and 08226, University of Michigan, Dept. of Computer Science and Communication Sciences, Ann Arbor, 1968.

[8] John H. Holland. Adaptation in natural and artificial systems, 1975.

[9] M. Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002.

[10] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[11] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[12] T. Mitchell, P. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter Chapter 6, pages 163–190. Springer-Verlag, 1984.

[13] Mariusz Nowostawski. *"Evolvable Virtual Machines"*. PhD thesis, "Information Science Department, University of Otago, Dunedin, New Zealand, 12 2008.

[14] Mariusz Nowostawski, Lucien Epiney, and Martin Purvis. Self-Adaptation and Dynamic Environment Experiments with Evolvable Virtual Machines. In S.Brueckner, G.Di Marzo Serugendo, D.Hales, and F.Zambonelli, editors, *Proceedings of the Third International Workshop on Engineering Self-Organizing Applications (ESOA 2005)*, pages 46–60. Springer Verlag, 2005.

[15] Mariusz Nowostawski, Martin Purvis, and Stephen Cranefield. An architecture for self-organising evolvable virtual machines. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, *Engineering Self Organising Sytems: Methodologies and Applications*, number 3464 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2004.

[16] I. P. Pavlov. *Conditioned Reflexes*. Oxford University Press, 1927. http://psychclassics.yorku.ca/Pavlov/.

[17] Juergen Schmidhuber. Self-referential learning, or on learning how to learn: The meta-meta-... hook. Diploma thesis, Institut fuer Informatik, Technische Universitaet Muenchen,, 1987. http://www.idsia.ch/ juergen/diploma.html.
[18] Juergen Schmidhuber. On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, 1994.
[19] Juergen Schmidhuber. Environment-independent reinforcement acceleration. Technical Note IDSIA-59-95, IDSIA, Lugano, 1995.
[20] Juergen Schmidhuber. A general method for incremental self-improvement and multiagent learning. In X. Yao, editor, *Evolutionary Computation: Theory and Applications*, chapter 3, pages 81–123. Scientific Publishers Co., Singapore, 1999.
[21] Juergen Schmidhuber. Optimal ordered problem solver. Technical Report IDSIA-12-02, IDSIA, 31 July 2002.
[22] Juergen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.
[23] E. L. Thorndike. *Animal Intelligence*. Macmillan, New York, 1911.
[24] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In D. Touretzky and M Mozer, editors, *Advances in Neural Information Processing Systems (NIPS) 8*, pages 640–646, Cambridge, MA, 1996. MIT Press.
[25] Sebastian Thrun and L. Pratt. *Learning to Learn*. Kluwer Academic Publishers, 1998.
[26] Alan Mathison Turing. Intelligent machinery. Technical Report Machine Intelligence 5, National Physical Laboratory, Edinburgh, 1948.
[27] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. A Bradford Book, MIT Press, Cambridge, Massachusetts/London, England, 1999.
[28] J. B. Watson. Kinesthetic and organic sensations and their role in the reaction of the white rat to the maze. *Psychological Review Monograph Supplement*, 8(33):1–100, 1907.
[29] Y. Xue, X. Liao, L. Carin, and B. Krishnapuram. Multi-Task Learning for Classification with Dirichlet Process Priors. *The Journal of Machine Learning Research*, 8:35–63, 2007.