Continually Adding Self-Invented Problems to the Repertoire: First Experiments with PowerPlay

Rupesh Kumar Srivastava, Bas R. Steunebrink, Marijn Stollenga and Jürgen Schmidhuber
The Swiss AI Lab IDSIA
University of Lugano & SUPSI
Galleria 2, 6928 Manno-Lugano, Switzerland
Email: {rupesh, bas, marijn, juergen}@idsia.ch

***Abstract*—Pure scientists do not only invent new methods to solve given problems. They also invent new problems. The recent PowerPlay framework formalizes this type of curiosity and creativity in a new, general, yet practical way. To acquire problem solving prowess through playing, PowerPlay-based artificial explorers by design continually come up with the fastest to find, initially novel, but eventually solvable problems. They also continually simplify or speed up solutions to previous problems. We report on results of first experiments with PowerPlay. A self-delimiting recurrent neural network (SLIM RNN) is used as a general computational architecture to implement the system's solver. Its weights can encode arbitrary, self-delimiting, halting or non-halting programs affecting both environment (through effectors) and internal states encoding abstractions of event sequences. In open-ended fashion, our PowerPlay-driven RNNs learn to become increasingly general problem solvers, continually adding new problem solving procedures to the growing repertoire, exhibiting interesting developmental stages.**

I. Introduction

In traditional computer science, formally defined tasks are typically solved by searching a space of candidates until a solution is found and verified. To automatically construct an increasingly general problem solver, we expand the traditional search space in an unusual way. The recent PowerPlay framework [22] incrementally searches the space of possible pairs of (1) new tasks (from the set of all computable tasks), and (2) modifications of the current problem solver. The search continues until the first pair is discovered for which (a) the current solver cannot solve the new task, and (b) the modified solver provably solves all previously learned tasks plus the new one. Here the new task may actually be to simplify, compress, or speed up previous solutions.

As a concrete implementation of the solver, we use a special neural network (NN) [2] architecture called the Self-Delimiting NN or SLIM NN [23]. It is designed for incremental learning with auto-modularization. Given a SLIM NN that can already solve a finite known set of previously learned tasks, an asymptotically optimal program search algorithm [7], [24], [18], [19] can be used to find a new pair that provably has properties (a) and (b). Once such a pair is found, the cycle repeats itself. This results in a continually growing set of tasks solvable by an increasingly more powerful solver. The resulting repertoire of self-invented problem-solving procedures can be exploited at any time to solve externally posed tasks.

The SLIM NN can be broken into modifiable components, namely, its connection weights. By keeping track of which tasks are dependent on each connection, PowerPlay can reduce the time required for testing previously solved tasks. In effect, only the tasks that depend on the changed connections need to be retested. If the most recent task does not require changes of many weights, and if the changed connections do not affect many previous tasks, then validation may be very efficient. Hence PowerPlay prefers to invent tasks whose validity check requires less computational effort. This implicit incentive to generate modifications that do not impact many previous tasks, leads to a natural decomposition of the space of tasks and their solutions into more or less independent regions. Thus, divide and conquer strategies are natural by-products of PowerPlay.

Unlike our first implementations of curious / creative / playful agents from the 1990s [15], [26], [16] (*cf.* [1], [4], [11], [9]), PowerPlay provably (by design) does not have any problems with online learning—it cannot forget previously learned skills, automatically segmenting its life into a sequence

of clearly identified tasks with explicitly recorded solutions. Unlike the task search of *theoretically optimal* creative agents [20], [21], POWERPLAY's task search is greedy, yet practically feasible. This paper presents the first experimental application of this framework to pattern recognition and motor control tasks demonstrating the above features.

II. NOTATION & ALGORITHMIC FRAMEWORK FOR POWERPLAY (VARIANT II) [22]

B^* denotes the set of finite bitstrings over the binary alphabet $B = \{0, 1\}$, $\mathbb{N}$ the natural numbers, $\mathbb{R}$ the real numbers. The computational architecture of POWERPLAY's problem solver may be a deterministic universal computer, or a more limited device such as a feedforward NN. All problem solvers can be uniquely encoded [5] or implemented on universal computers such as universal Turing Machines (TM) [27]. Therefore, without loss of generality, we can assume a fixed universal reference computer whose inputs and outputs are elements of B^*. User-defined subsets $\mathcal{S}, \mathcal{T} \subset B^*$ define the sets of possible problem solvers and task descriptions. For example, $\mathcal{T}$ may be the infinite set of all computable tasks, or a small subset thereof. $\mathcal{P} \subset B^*$ defines a set of possible programs which may be used to generate or modify members of $\mathcal{S}$ or $\mathcal{T}$. If our solver is a feedforward NN, then $\mathcal{S}$ could be a highly restricted subset of programs encoding the NN's possible topologies and weights, $\mathcal{T}$ could be encodings of input-output pairs for a supervised learning task, and $\mathcal{P}$ could be an algorithm that modifies the weights of the network.

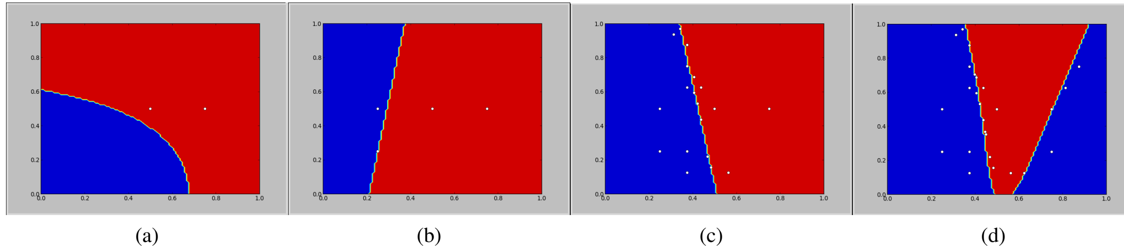
The problem solver's initial program is called s_0. A particular sequence of unique task descriptions $T_1, T_2, \ldots$ (where each $T_i \in \mathcal{T}$) is chosen or "invented" by a search method (examples below) such that the solutions of $T_1, \ldots, T_i$ can be computed by s_i, the i-th instance of the program, but T_i cannot be solved by s_{i-1}. Each T_i consists of a unique problem identifier that can be read by s_i through some built-in mechanism (e.g., input neurons of an NN as in Sec. III and IV), and a unique description of a deterministic procedure for deciding whether the problem has been solved. For example, a simple task may require the solver to answer a particular input pattern with a particular output pattern. Or it may require the solver to steer a robot towards a goal through a sequence of actions. Denote $T_{\leq i} = \{T_1, \ldots, T_i\}$; $T_{<i} = \{T_1, \ldots, T_{i-1}\}$. A valid task T_i $(i > 1)$ may require solving at least one previously solved task T_k $(k < i)$ more efficiently, by using less resources such as storage space, computation time, energy, etc. quantified by the function $Cost(s, T)$. The algorithmic framework (Alg. 1) incrementally trains the problem solver by finding $p \in \mathcal{P}$ that increase the set of solvable tasks. For more details, the reader is encouraged to refer to the original report [22].

Algorithm 1 POWERPLAY Framework (Variant II)

Initialize s_0 in some way
for $i := 1, 2, \ldots$ **do**
  Declare new global variables $T_i \in \mathcal{T}$, $s_i \in \mathcal{S}$, $p_i \in \mathcal{P}$, $c_i, c_i^* \in \mathbb{R}$ (all unassigned)
  repeat
    Let a search algorithm (e.g., Section III) set p_i, a new candidate program. Give p_i limited time to do:
    * TASK INVENTION: Unless the user specifies T_i, let p_i set T_i.
    * SOLVER MODIFICATION: Let p_i set s_i by computing a modification of s_{i-1}.
    * CORRECTNESS DEMONSTRATION: Let p_i compute $c_i := Cost(s_i, T_{\leq i})$ and $c_i^* := Cost(s_{i-1}, T_{\leq i})$
  until $c_i^* - c_i > \epsilon$ (minimal savings of costs such as time/space/etc on all tasks so far)
  Freeze/store forever $p_i, T_i, s_i, c_i, c_i^*$
end for

III. EXPERIMENT 1: SELF-INVENTED PATTERN RECOGNITION TASKS

In our first experimental investigation of POWERPLAY we examined pattern-classification tasks. In this setup, s encodes an arbitrary set of weights for a fixed-topology multi-layer perceptron (MLP). The MLP maps two-dimensional, real-valued input vectors from the unit square to binary labels; i.e., s: $[0, 1) \times [0, 1) \rightarrow 0, 1$. The output label is 0 or 1 depending on whether or not the real-valued activation of the MLP's single output neuron exceeds 0.5. Binary programs $p \in \mathcal{P}$ compute tasks and modify s as follows. If p^1 (the first bit of p) is 1, this indicates that the current task is to simplify s by weight decay, under the assumption that smaller weights are simpler. But if p^1 is 0, then the target label of the current task candidate T is given by the next bit p^2, and T's two-dimensional input vector is uniquely encoded by the remainder of p's bit string, $p^3 p^4 \ldots p^n$, according to a prewired coding scheme. A pseudo-random sequence generator is used to decode the tasks using the bitstrings in this experiment. It is re-seeded by the same seed every time a new task search begins, thus ensuring a deterministic search order. Since we only have two labels in this experiment, we do not need p^2 as we can choose the target label to be different from the label currently assigned by the MLP to the encoded input. To run p for t steps (on



(a) (b) (c) (d)

Fig. 1. (a) The MLP finds it easiest to learn to give the same label to the first patterns. (b) After giving the same label to the entire space (which is helped by the drive to compress), the MLP soon invents pattern classifications involving the opposite class. The compression drive still forces the decision boundary to be linear. (c) As more associations are invented, it becomes harder and harder to learn new ones that break the previous solver's generalization ability, while maintaining a linear boundary. (d) The decision boundary becomes non-linear, and increasingly so, as more and more associations are invented and learned.

a training set of i patterns so far) means to execute $\lfloor t/2i \rfloor$ epochs of gradient descent on the training set and check whether the patterns are correctly classified. One step always means the processing of one pattern (either a forward or backward pass), regardless of the task.

Assume now that POWERPLAY has already learned a version of s called s_{i-1} able to classify $i-1$ previously invented training patterns ($i > 1$). Then the next task is defined by a simple enumerative search in the style of universal search [8], [24], [19], which combines task simplification and systematic run-time growth (see Alg. 2).

Algorithm 2 POWERPLAY implementation for experiment 1

```
Initialize s_0 in some way
for i := 1, 2, ... do
  for m := 1, 2, ... do
    for all candidate programs p s.t. L(p) ≤ m do
      Run p for at most 2^{m−L(p)} steps;
      if (p creates s_i from s_{i−1} correctly classifying
      all i training patterns so far) and (s_i either is
      substantially simpler than s_{i−1}, or can also
      classify a newly found pattern misclassified
      by s_{i−1}) then
        Set p_i := p (store the candidate)
        exit m loop;
      end if
    end for
  end for
end for
```

Since the compression task code is the single bit '1', roughly half of the total search time is spent on simplification, the rest is spent on the invention of new training patterns that break the MLP's current generalization ability.

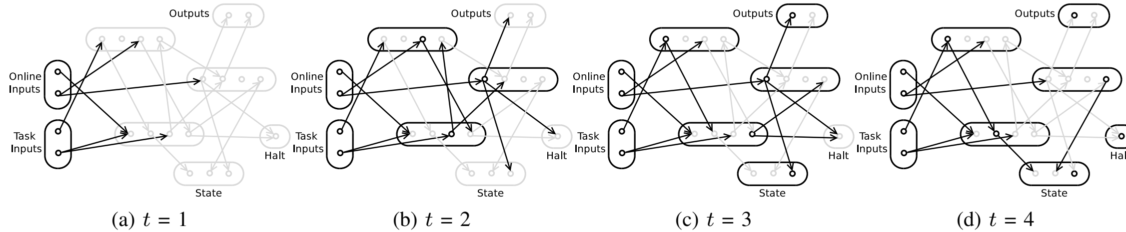
After each successful search for a new task, the labels of grid points are plotted in a rather dense grid on the unit square (Fig. 1), to see how the MLP maps $[0, 1) \times [0, 1)$ to $0, 1$, thus monitoring the evolution of its generalization map. As expected, the experiments show that in the beginning POWERPLAY prefers to invent and learn simple linear functions. However, there is a phase transition to more complex non-linear functions after a few tasks, indicating a new developmental stage [12], [17], [10]. This is a natural by-product of the search for simple tasks—they are easier to invent and verify than more complex non-linear tasks. As learning proceeds, we observe that the decision boundary becomes increasingly non-linear, which happens because the system has to come up with tasks which the solver cannot solve yet, but the solver becomes increasingly more powerful, so the system has to invent increasingly harder tasks.

IV. EXPERIMENT 2: SELF-INVENTED TASKS INVOLVING MOTOR CONTROL AND INTERNAL ABSTRACTIONS

A. *Self-Delimiting (SLIM) Programs Running on A Recurrent Neural Network (RNN)*

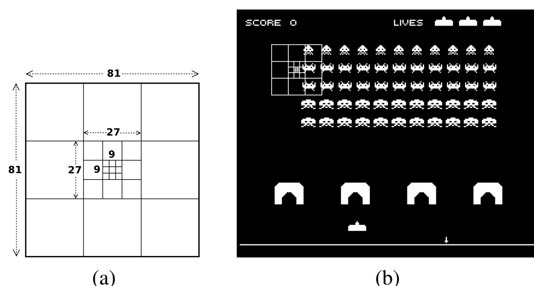
Here we demonstrate how a POWERPLAY-based RNN continually invents novel sequences of actions in an environment, over time becoming a more and more general solver of self-invented problems. RNNs are general computers that allow for both sequential and parallel computations. Given enough neurons and an appropriate weight matrix, an RNN can compute any function computable by a standard PC [14]. We use a particular RNN named SLIM RNN [23] to define $\mathcal{S}$ for our experiment. Its salient features are explained here.

The k-th computational unit or *neuron* of our SLIM RNN is denoted u^k ($0 < k \leq n(u) \in \mathbb{N}$). w^{lk} is the real-valued *weight* on the directed connection c^{lk} from u^l to u^k. At discrete time step $t = 1, 2, \ldots, t_{end}$ of a finite interaction sequence with the environment, $u^k(t)$ denotes the real-valued *activation* of u^k. There are designated neurons



(a) t = 1 (b) t = 2 (c) t = 3 (d) t = 4

Fig. 2. SLIM RNN activation scheme. At various time steps, active/winning neurons and their outgoing connections are highlighted. At each step, at most one neuron per WITAS can become active and propagate activations through its outgoing connections.



(a) (b)

Fig. 3. (a) Fovea design. Pixel intensities over each square are averaged to produce a real valued input. The smallest squares in the center are of size 3x3. (b) The RNN controls the fovea movement over a static image, such as a still from the Space Invaders game.

serving as *online inputs* which read real valued observations from the environment and *outputs* whose activations are passed to the environment for interpretation of actions, e.g., the movement commands for a robot. We initialize all $u^k(1)$ = 0 and compute $u^k(t+1) = f^k(\sum_l w^{lk} u^l(t))$ where f may be of the form $f^k(x) = 1/(1+e^{-x})$, or $f^k(x) = x$, or $f^k(x) = 1$ if $x \geq 0$ and 0 otherwise. To program the SLIM RNN means to set the weight matrix $\langle w^{lk} \rangle$.

A special feature of the SLIM RNN is that there is a single *halt* neuron in the network which has a fixed *halt-threshold*, such that if at any time t the activation at the neuron exceeds the *halt-threshold*, the network's computation stops. Thus, any network topology in which there exists a path from the online or task inputs to the halt neuron can run self-delimiting programs [8], [3], [24], [19] studied in the theory of Kolmogorov complexity and algorithmic probability [25], [6]. Inspired by a previous architecture [13], neurons other than the inputs and outputs in our RNN are arranged in winner-take-all subsets (WITAS) of n_{witas} neurons each ($n_{witas} = 4$ was used for this experiment). At each time step t, $u^k(t)$ is set to 1 for the winning neuron in each WITAS (the one with the highest activation), and for other neurons it is set to 0. This feature gives SLIM RNN a special modularization ability, since neurons can act as *gates* to different regions of the network. By regulating the information flow through the network, it becomes possible to use only a fraction of the weights $\langle w^{lk} \rangle$ for each task the network learns. Since POWERPLAY methodically increases search time and devotes half the search time to simplification, this feature encourages the network to invent novel tasks that do not require many changes of weights used by many previous tasks. Sec. IV-C shows that it tends to modify the weight matrix such that over time novel tasks depend on fewer weights.

Aside from the online input, output and halt neurons, a fixed number n_{ti} of neurons are set to be *task inputs*. These inputs remain constant for $1 \leq t < t_{end}$ and serve as self-generated task labels. Finally, there is a subset of n_s *internal state* neurons whose activations are considered as the final outcome when the program halts. Thus a task is: Given a particular task input, interact with the environment (read online inputs, produce outputs) until the network halts and produces a particular internal state, which is read from the internal state neurons. Essentially arbitrary computable tasks can be represented in this way by the SLIM RNN. Fig. 2 illustrates the network activation on a particular task. A more detailed discussion of SLIM RNNs and their efficient implementation can be found in a previous paper [23].

Our SLIM RNN implementation efficiently resets activations computed by the numerous unsuccessful tested candidate programs. We keep track of used connections and active (winner) neurons at each time step, to reset activations such that tracking/undoing effects of programs essentially does not cost more than their execution.

B. RNN-Controlled Fovea Environment

The environment for this experiment consists of a static image which is observed by the RNN through a fovea, whose movement it can control at

each time step. The size of the fovea is 81×81 pixels; it produces 25 real valued online inputs (normalized to $[0, 1]$) by averaging the pixel intensities over regions of varying sizes such that it has higher resolution at the center and lower resolution in the periphery (Fig. 3). The fovea is controlled using 8 real-valued outputs of the network, and a parameter *win-threshold*. Out of the first four outputs, the one with the highest value greater than the *win-threshold* is interpreted as a movement command: up, down, left, or right. If none of the first four outputs exceeds the threshold, the fovea does not move. Similarly, the next four outputs are interpreted as the extent of movement of the fovea over the image (3, 9, 27 or 81 pixels in case of exceeding the threshold, 1 pixel otherwise).

C. Results

The SLIM RNN described in Sec. IV-A is trained on the fovea environment using the POWERPLAY framework in a manner similar to Alg. 2. The difference lies in the encoding of task inputs and the definition of 'inventing and learning' a task. The bitstring p now encodes a set of n_{ti} real numbers between 0 and 1 which denote the constant task inputs for this program. A new task is considered learned if given a new set of task inputs, the network halts and reaches an internal state after interacting with the environment, and is also able to reproduce the saved internal states of all previously learned tasks. If the network cannot halt within a chosen fraction of the time budget dictated by $L(p)$, the remaining budget is used for trying to learn the task using a simple mutation rule, by modifying a few weights of the network.

In this way, the network's internal states abstract from its trajectories through the fovea environment as it invents more and more tasks without forgetting previously learned ones. A SLIM RNN consisting of 40 WITAS, with 4 neurons in each WITAS, invented 46 action sequences for guiding the fovea before halting, within 24 hours on a standard PC. The action sequences exploring the environment were of varying lengths (3–10 steps).

As a result of the network design, we observe that the SLIM NN uses partially overlapping subsets of connection weights for generating self-invented trajectories. Fig. 4 shows that not all connections are used for all tasks. The *usage ratio* on the y-axis is defined as the number of tasks the connection is used for, divided by the number of tasks learned so far. The ratio is 1 for the first 200 connections, which are frequently used outgoing connections from task and online inputs. On the whole, as more tasks get learned, the number of connections with high usage ratio falls, indicating that the network is becoming more modular (many tasks do not depend on the same connections). That is, it becomes possible to modify connections with a lower ratio to invent a new task without affecting many previously learned ones.

V. CONCLUSION

POWERPLAY for SLIM RNN represents a greedy implementation of central aspects of the Formal Theory of Fun and Creativity [20], [21]. It permits general yet practically feasible, curious/creative agents that learn hierarchically and modularly. This paper reports on the first known general yet practical implementation. Each new task invention either breaks the solver's present generalization ability, or compresses the solver, or speeds it up.

We can know precisely what is learned by our POWERPLAYing SLIM NN. The self-invented tasks are clearly defined by inputs and internal outcomes / results. Human interpretation of the NN's weight changes, however, may be difficult, a bit like with a baby that generates internal representations and skills or skill fragments during play. What is their "meaning" in the eyes of the parents, to whom the baby's internal state is a black box? In case of the fovea tasks, for example, the learner invents certain input-dependent movements as well as abstractions of trajectories in the environment (limited by its vocabulary of internal states). The RNN weights at any stage encode the agent's present (possibly limited) understanding of the environment and what can be done in it.

POWERPLAY has no problems with noisy inputs from the environment. However, noisy versions of an old, previously solved task must be considered as new tasks, because in general we do not know what is noise and what is not. But over time POWERPLAY can automatically learn to generalize away the "noise," eventually finding a compact solver that solves all "noisy" instances seen so far.

Our first experiments focused on developmental stages of a purely creative system, and did not involve any externally posed tasks yet. Future work will test the hypothesis that systems that have been running POWERPLAY for a while will be faster at solving many user-provided tasks than systems without such purely explorative components. This hypothesis is inspired by babies who creatively seem to invent and learn many skills autonomously, which then helps them to learn additional teacher-defined external tasks.

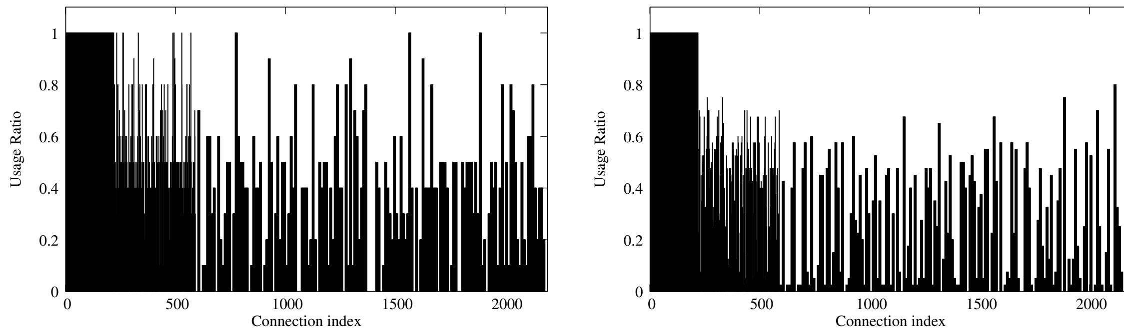


Fig. 4. Connection usage ratios for all SLIM RNN connections after 10 (left) and 40 (right) self-invented tasks have been learnt. The network learns to better utilize its own architecture by using different connections for different tasks, thus reducing the number of connections with high usage ratio. Such modularization can be exploited to speed up task search in later stages.

ACKNOWLEDGMENTS

POWERPLAY [22] and self-delimiting recurrent neural networks (SLIM RNN) [23] were developed by J. Schmidhuber and implemented by R.K. Srivastava and B.R. Steunebrink. We thank M. Stollenga and N.E. Toklu for their help with the implementations. This research was funded by the following EU projects: IM-CLeVeR (FP7-ICT-IP-231722) and WAY (FP7-ICT-288551).

REFERENCES

[1] A. Barto. Intrinsic motivation and reinforcement learning. In G. Baldassarre and M. Mirolli, editors, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer, 2012. In press.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] G. J. Chaitin. A theory of program size formally identical to information theory. *J. of the ACM*, 22:329–340, 1975.

[4] P. Dayan. Exploration from generalization mediated by multiple controllers. In G. Baldassarre and M. Mirolli, editors, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer, 2012. In press.

[5] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[6] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

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

[8] L. A. Levin. Laws of information (nongrowth) and aspects of the foundation of probability theory. *Problems of Information Transmission*, 10(3):206–210, 1974.

[9] U. Nehmzow, Y. Gatsoulis, E. Kerr, J. Condell, N. H. Siddique, and T. M. McGinnity. Novelty detection as an intrinsic motivation for cumulative learning robots. In G. Baldassarre and M. Mirolli, editors, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer, 2012. In press.

[10] H. Ngo, M. Ring, and J. Schmidhuber. Compression progress-based curiosity drive for developmental learning. In *Proc. of the 2011 IEEE Conf. on Development and Learning and Epigenetic Robotics ICDL-EPIROB*. 2011.

[11] P.-Y. Oudeyer, A. Baranes, and F. Kaplan. Intrinsically motivated learning of real world sensorimotor skills with developmental constraints. In G. Baldassarre and M. Mirolli, editors, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer, 2012. In press.

[12] J. Piaget. *The Child's Construction of Reality*. London: Routledge and Kegan Paul, 1955.

[13] J. Schmidhuber. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412, 1989.

[14] J. Schmidhuber. Dynamische neuronale Netze und das fundamentale raumzeitliche Lernproblem. Dissertation, Institut für Informatik, Technische Univ. München, 1990.

[15] J. Schmidhuber. Curious model-building control systems. In *Proc. of the Internat. Joint Conf. on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE press, 1991.

[16] J. Schmidhuber. Artificial curiosity based on discovering novel algorithmic predictability through coevolution. In P. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and Z. Zalzala, editors, *Congress on Evolutionary Computation*, pages 1612–1618. IEEE Press, 1999.

[17] J. Schmidhuber. Exploring the predictable. In A. Ghosh and S. Tsuitsui, editors, *Advances in Evolutionary Computing*, pages 579–612. Springer, 2002.

[18] J. Schmidhuber. Bias-optimal incremental problem solving. In S. Becker, S. Thrun, and K. Obermayer, eds., *Adv. in Neural Information Processing Systems 15 (NIPS 15)*, pages 1571–1578, Cambridge, MA, 2003. MIT Press.

[19] J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.

[20] J. Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006.

[21] J. Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010.

[22] J. Schmidhuber. POWERPLAY: Training an Increasingly General Problem Solver by Continually Searching for the Simplest Still Unsolvable Problem. Technical Report arXiv:1112.5309v1 [cs.AI], IDSIA, 2011.

[23] J. Schmidhuber. Self-delimiting neural networks. Technical report IDSIA-08-12, arXiv:1210.0118v1 [cs.NE], IDSIA, 2012.

[24] J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130, 1997.

[25] R. J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

[26] J. Storck, S. Hochreiter, and J. Schmidhuber. Reinforcement driven information acquisition in non-deterministic environments. In *Proceedings of the International Conference on Artificial Neural Networks, Paris*, volume 2, pages 159–164. EC2 & Cie, 1995.

[27] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society, Series 2*, 41:230–267, 1936.