First Experiments with POWERPLAY

Rupesh Kumar Srivastava, Bas R. Steunebrink and Jürgen Schmidhuber
The Swiss AI Lab IDSIA, Galleria 2, 6928 Manno-Lugano
University of Lugano & SUPSI, Switzerland

2012

Abstract

Like a scientist or a playing child, POWERPLAY [24] not only learns new skills to solve given problems, but also *invents* new interesting problems by itself. By design, it continually comes up with the fastest to find, initially novel, but eventually solvable tasks. It also continually simplifies or compresses or speeds up solutions to previous tasks. Here we describe first experiments with POWERPLAY. A self-delimiting recurrent neural network SLIM RNN [25] is used as a general computational problem solving architecture. Its connection weights can encode arbitrary, self-delimiting, halting or non-halting programs affecting both environment (through effectors) and internal states encoding abstractions of event sequences. Our POWERPLAY-driven SLIM RNN learns to become an increasingly general solver of self-invented problems, continually adding new problem solving procedures to its growing skill repertoire. Extending a recent conference paper [28], we identify interesting, emerging, developmental stages of our open-ended system. We also show how it automatically self-modularizes, frequently re-using code for previously invented skills, always trying to invent novel tasks that can be quickly validated because they do not require too many weight changes affecting too many previous tasks.

1 Introduction

To automatically construct an increasingly general problem solver, the recent POWERPLAY framework [24] incrementally and efficiently searches the space of possible pairs of (1) new task descriptions (from the set of all computable task descriptions), and (2) modifications of the current problem solver. The search continues until the first pair is discovered for which (i) the current solver cannot solve the new task, and (ii) the modified solver provably solves all previously learned tasks plus the new one. Here a new task may actually be to simplify, compress, or speed up previous solutions, which in turn may invoke or partially re-use solutions to other tasks. The above process of discovering and solving a novel task can be repeated forever in open-ended fashion.

As a concrete implementation of the solver, we use a special neural network (NN) [2] architecture called the Self-Delimiting NN or SLIM NN [25]. Given a SLIM NN that can already solve a finite known set of previously learned tasks, an asymptotically optimal program search algorithm [9, 26, 20, 21] can be used to find a new pair that provably has properties (i) and (ii). 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 or skills can be exploited at any time to solve externally posed tasks.

The SLIM NN has modifiable components, namely, its connection weights. By keeping track of which tasks depend on each connection, POWERPLAY can reduce the time required for testing previously solved tasks with certain newly modified connection weights, because only tasks that depend on the changed connections need to be retested. If the solution of the most recently invented 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. Since POWERPLAY's efficient search process has a built-in bias towards tasks whose validity check requires little computational effort, there is an implicit incentive to generate weight modifications that do not impact too many previous tasks. This 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.

Note that active learning methods [5] such as AdaBoost [6] have a totally different set-up and purpose: there the user provides a set of samples to be learned, then each new classifier in a series of classifiers focuses on samples badly classified by previous classifiers. In open-ended POWERPLAY, however, all computational tasks (not necessarily classification tasks) can be self-invented; there is no need for a pre-defined global set of tasks that each new solver tries to solve better, instead the task set continually grows based on which task is easy to invent and validate, given what is already known.

Unlike our first implementations of curious / creative / playful agents from the 1990s [17, 29, 18] (*cf.* [1, 4, 13, 11]), 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 [22, 23], POWERPLAY's task search is greedy, yet practically feasible. Here we present first experiments, extending recent work [28].

2 Notation & Algorithmic Framework for POWERPLAY (Variant II)

We use the notation of the original paper [24], and briefly review the basics relevant here. B^* denotes the set of finite bit strings 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 [7] or implemented on universal computers such as universal Turing Machines (TM) [31]. 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 (see 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. 3 and 4), 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 [24].

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 3) 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

3 Experiment 1: Self-Invented Pattern Recognition Tasks

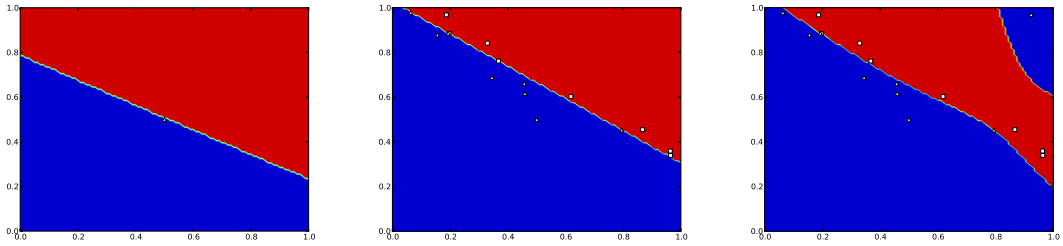
We start with pattern classification tasks. In this setup, s encodes an arbitrary set of weights for a fixed-topology multi-layer perceptron (MLP). The MLP maps two-dimensional, real-valued input vectors from the unit square to binary labels; i.e., s: $[0, 1) \times [0, 1) \rightarrow 0, 1$. The output 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 $p \in \mathcal{P}$ of length $length(p)$ compute tasks and modify s as follows. If p^1 (the first bit of p) is 0, this will specify that the current task is to simplify s by weight decay, under the assumption that smaller weights are simpler. Such programs implement compression tasks. But if p^1 is 1, then the target label of the current task candidate T will be given by the next bit p^2, and T's two-dimensional input vector will be uniquely encoded by the remainder of p's bit string, $p^3 p^4 \ldots p^n$, as follows. The string $p^3 p^4 \ldots p^n$ is taken as the binary representation of an integer N. Then a 2D Gaussian pseudo-random number generator is used to generate numbers $(x_1, y_1), (x_2, y_2), \ldots$, where x and y are used as 2D coordinates in the unit square. Now the task is to label the coordinates (x_N, y_N) as p_2.

The random number generator 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 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. Here one step always refers to the processing of a single pattern (either a forward or backward pass), regardless of the task.

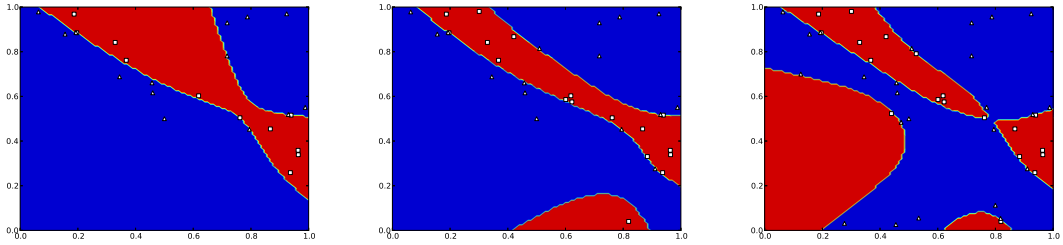
Assume now that POWERPLAY has already learned a version of s called s_{i-1} able to classify $i - 1$ previously invented training patterns ($i > 1$). Then the next task is defined by a simple enumerative search in the style of universal search [10, 26, 21], which combines task simplification and systematic run-time growth (see Alg. 2).

Algorithm 2 POWERPLAY implementation for experiment 1

Initialize s_0 in some way
for $i := 1, 2, \ldots$ **do**
  for $m := 1, 2, \ldots$ **do**
    for all candidate programs p s.t. $length(p) \leq m$ **do**
      Run p for at most $2^{m-length(p)}$ steps
      if p creates s_i from s_{i-1} correctly classifying all i training patterns so far **and** (s_i is substantially simpler than s_{i-1} **or** s_i can 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



(a) After 1 task (b) After 16 tasks (c) After 17 tasks



(d) After 25 tasks (e) After 33 tasks (f) After 43 tasks

Figure 1: Experiment 1. Right after initialization, before the first compressions, the decision boundary may be arbitrary and possibly non-linear. The drive to compress and simplify, however, first encourages linear separability (top row). 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. Eventually this causes the decision boundary to become non-linear (bottom row). The decision boundary becomes increasingly non-linear, as more and more associations are invented and learned.

Since the compression task code is the single bit '0', 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.

To monitor the evolution of the solver's *generalization map*, 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 $0, 1$. 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 [14, 19, 12]. 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, 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. On the other hand, the search time for solutions to harder and harder tasks need not grow over time, since new solutions do not have to be learnt from scratch, but may re-use previous solutions encoded as parts of the previous solver.

4 Experiment 2: Self-Invented Tasks Involving Motor Control and Internal Abstractions

4.1 Self-Delimiting (SLIM) Programs Run on A Recurrent Neural Network (RNN)

Here we describe experiments with a POWERPLAY-based RNN that continually invents novel sequences of actions affecting an external environment, over time becoming a more and more general solver of self-invented problems.

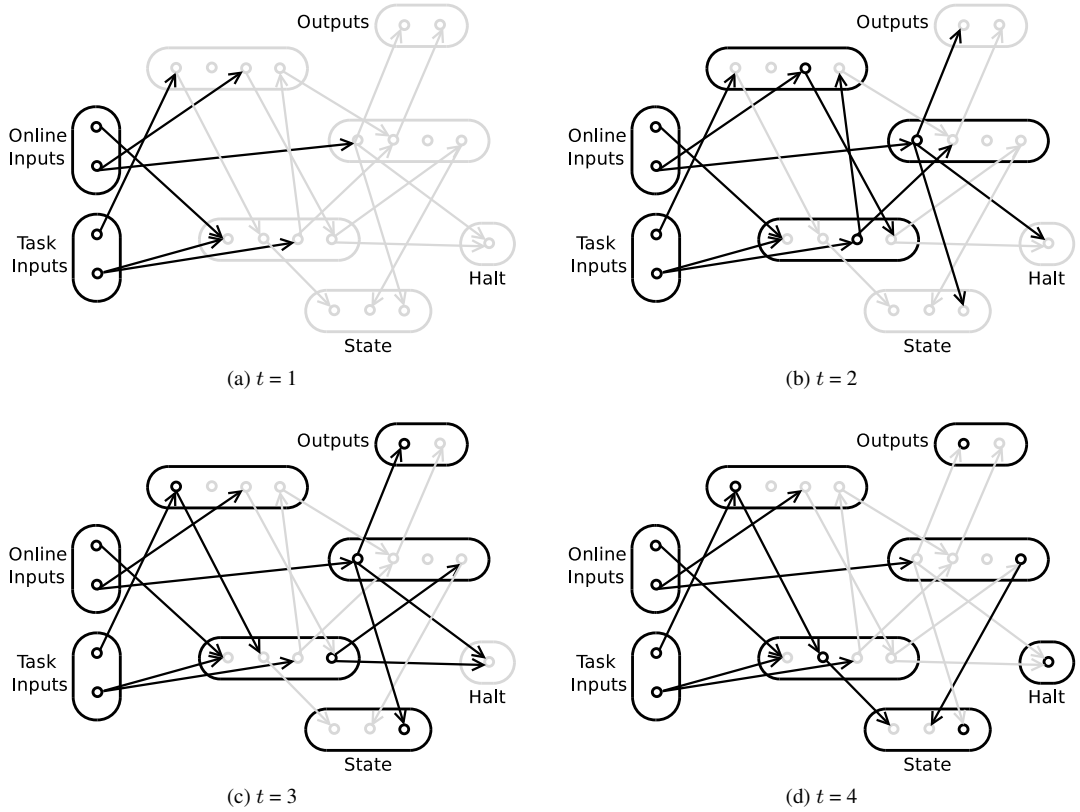


Figure 2: SLIM RNN activation scheme. At various time steps, active/winning neurons and their outgoing connections are highlighted. At each step, at most one neuron per WITAS can become active and propagate activations through its outgoing connections.

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 [16]. We use a particular RNN named SLIM RNN [25] to define $\mathcal{S}$ for our experiment. Here we briefly review its basics.

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 serving as *online inputs*, which read real-valued observations from the environment, and *outputs* whose activations encode actions in the environment, 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 it has a single *halt* neuron with a fixed *halt-threshold*. If at any time t its activation 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 [10, 3, 26, 21] studied in the theory of Kolmogorov complexity and algorithmic probability [27, 8]. Inspired by a previous architecture [15], 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 if u^k is a winning neuron in some WITAS (the one with the highest activation), and to 0 otherwise. This feature gives the SLIM RNN the potential to modularize itself, since neurons can act as *gates* to various self-determined regions of the network. By regulating the information flow, the network may use only a fraction of the weights $\langle w^{lk} \rangle$ for a given task.

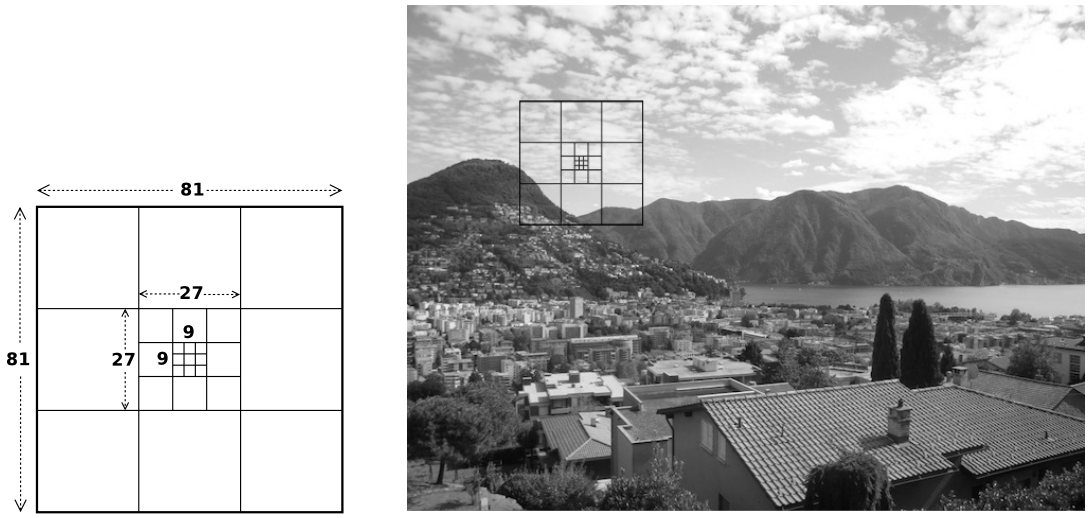
Apart from the online input, output and halt neurons, a fixed number n_{ti} of neurons are set to be *task inputs*. These inputs remain constant for $1 \leq t < t_{end}$ and serve as self-generated task specifications. Finally, there is a subset of n_s *internal state* neurons whose activations are considered as the final outcome when the program halts. Thus a *non-compression* 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—the abstract *goal*—which is read from the internal state neurons. Since the SLIM RNN is a general computer, it can represent essentially arbitrary computable tasks in this way. Fig. 2 illustrates the network's activation spreading for a particular task. A more detailed discussion of SLIM RNNs and their efficient implementation can be found in the original report [25].

The SLIM RNN is trained on the fovea environment described in Sec. 4.2 using the POWERPLAY framework according to Algorithm 3 below. The difference to Algorithm 2 lies in task set-specific details such as the encoding of task inputs and the definition of 'inventing and learning' a task. The bit string p now encodes a set of n_{ti} real numbers between 0 and 1 which denote the constant task inputs for this program. Given a new set of task inputs, the new task is considered learned if the network halts and reaches a particular internal state (potentially after interacting with the environment), and remains able to properly reproduce the saved internal states for all previously learned tasks. This is implemented by first checking if the network can halt and produce an internal state on the newly generated task inputs. Only if the network cannot halt within a chosen fraction of the time budget dictated by $length(p)$, the length of program 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. When p is the single bit '0', the task is interpreted as a *compression* task. Here compression either means a reduction of the sum of squared weights without increasing the total number of connection usages by all previously learned tasks, or a reduction of the total number of connection usages on all previously learned tasks without increasing the sum of squared weights.

Algorithm 3 POWERPLAY implementation for experiment 2

```
Initialize s_0 in some way
for i := 1, 2, ... do
  for m := 1, 2, ... do
    for all candidate programs p s.t. length(p) ≤ m do
      Set time_budget := 2^(m−length(p))
      if p encodes a compression task then
        Set s_temp := s_{i−1}
        while time_budget > 0 do
          Create s_i from s_temp through random perturbation of a few connection weights
          if compression is successful and time_budget ≥ 0 then
            Set s_temp := s_i
          end if
        end while
      else
        while time_budget > 0 do
          Create s_i from s_{i−1} through random perturbation of a few connection weights
          From p generate task k
          if s_{i−1} does not solve k and s_i solves k and s_i solves all previous tasks in the repertoire and
          time_budget ≥ 0 then
            Add the pair (k, internal state) to the repertoire
            exit m loop
          end if
        end while
      end if
    end for
  end for
end for
```

Since our POWERPLAY variant methodically increases search time, half of which is used for compression, it automatically encourages the network to invent novel tasks that do not require many changes of weights used by many previous tasks.



(a) (b)

Figure 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 3×3. (b) The RNN controls the fovea movement over a static image, in our experiments this photo of the city of Lugano.

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.

4.2 RNN-Controlled Fovea Environment

The environment for this experiment consists of a static image which is observed sequentially 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 *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. Similarily, the next four outputs are interpreted as the fovea step size on the image (3, 9, 27 or 81 pixels in case of exceeding the threshold, 1 pixel otherwise).

4.3 Results

The network's internal states can be viewed as abstract summaries of its trajectories through the fovea environment and its parallel "internal thoughts." The system invents more and more novel skills, each breaking the generalization ability of its previous SLIM NN weight matrix, without forgetting previously learned skills. Within 8 hours on a standard PC, a SLIM RNN consisting of 20 WITAS, with 4 neurons in each WITAS, invented 67 novel action sequences guiding the fovea before halting. These varied in length, consuming up to 27 steps. Over time the SLIM NN not only invented new skills to solve novel tasks, but also learned to speed up solutions to previously learned tasks, as shown in Fig. 4. For clarity, all figures presented here depict aspects of this same run, though results were consistent over many different runs.

The SLIM NN also learns to reduce the interactions with the environment. Fig. 5 shows the number of interactions required to solve certain previously learned fovea control tasks. Here an "interaction" is a SLIM NN computation step that produces at least one non-zero output neuron activation. General trend over different tasks and runs: the interactions decrease over time. That is, the SLIM NN essentially learns

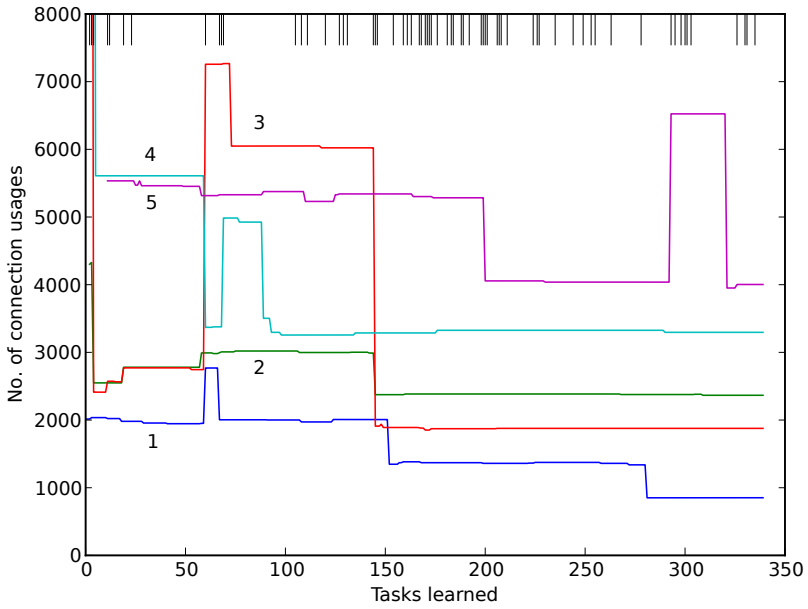


Figure 4: For the first five self-invented non-compression tasks, we plot the number of connection usages per task. In this run, solutions to 340 self-generated tasks were learned. 67 of them were non-compression tasks (marked by small black lines at the top); the rest resulted in successful compressions of the SLIM RNN's weight matrix. Over time, previously learned skills tend to require less and less computational resources, i.e., the SLIM RNN-based solver learns to speed up its solutions to previous self-invented tasks. Although some plot lines occasionally go up, this is compensated for by a decrease of connection usages for dozens of other tasks (not shown here to prevent clutter).

to build internal representations of its interaction with the environment, due to POWERPLAY's continual built-in pressure to speed up and simplify and generalize.

The SLIM NN often uses partially overlapping subsets of connection weights for generating different self-invented trajectories. Fig. 6 shows that not all connections are used for all tasks, and that the connections used to solve individual tasks can become progressively more separated. In general, the variation in degree of separation depends on network parameters and environment.

As expected, POWERPLAY-based SLIM NNs prefer to modify only few connections per novel task. Randomly choosing one to fifteen weight modifications per task, on average only 2.9 weights were changed to invent a new skill—see Fig. 7. Why? Because POWERPLAY is always going for the novel task that is fastest to find and validate, and fewer weight changes tend to affect fewer previously learned tasks; that is, less time is needed to re-validate performance on previous tasks. In this way POWERPLAY avoids a naively expected slowdown linear in the number of tasks. Although the number of skills that must not be forgotten grows all the time, the search time for new skills does not at all have to grow in proportion to the number of previously solved tasks.

As a consequence of its bias towards fast-to-validate solutions, the POWERPLAY-based SLIM NN automatically self-modularizes. The SLIM RNN tested above had 1120 connections. Typically, 600 of them were used to solve a particular task, but on average less than three of them were changed. This means that for each newly invented task, the system re-uses a lot of previously acquired knowledge without modification. The truly novel aspects of the task and its solution often can be encoded within just a handful of bits.

This type of self-modularization is more general than what can be found in traditional (non-inventive) modular reinforcement learning (RL) systems whose action sequences are chunked into macros to be re-used by higher-level macros, like in the options framework [30], or in hierarchical RL [32]. Since the

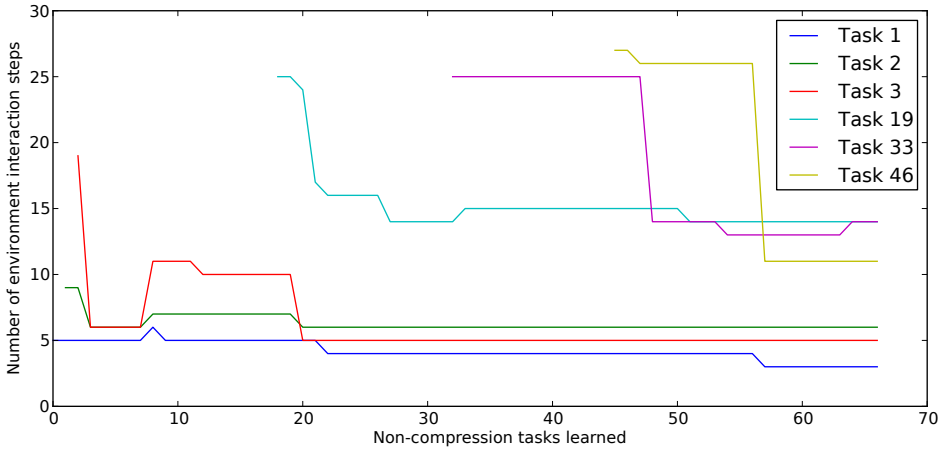


Figure 5: For only six selected tasks (to prevent clutter), we plot the number of interactions with the environment, over a run where 67 novel non-compression tasks were learned, besides numerous additional compression tasks ignored here. Here an interaction is a SLIM NN computation step that produces at least one non-zero output neuron activation. The total number of interactions cannot exceed the number of steps until the halt neuron is activated.

SLIM RNN is a general computer, and its weights are its program, subsets of the weights can be viewed as sub-programs, and new sub-programs can be formed from old ones in essentially arbitrary computable ways, like in general incremental program search [21].

5 Discussion and Outlook

POWERPLAY for SLIM RNN represents a greedy implementation of central aspects of the Formal Theory of Fun and Creativity [22, 23]. The setup permits practically feasible, curious/creative agents that learn hierarchically and modularly, using general computational problem solving architectures. 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 POWERPLAYing SLIM NN. The self-invented tasks are clearly defined by inputs and abstract internal outcomes / results. Human interpretation of the NN's weight changes, however, may be difficult, a bit like with a baby that generates new 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? For example, in case of the fovea tasks 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, a noisy version of an old, previously solved task must be considered as a new task, 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 purely creative systems, 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. We intend to identify conditions under which such knowledge transfer can be expected.

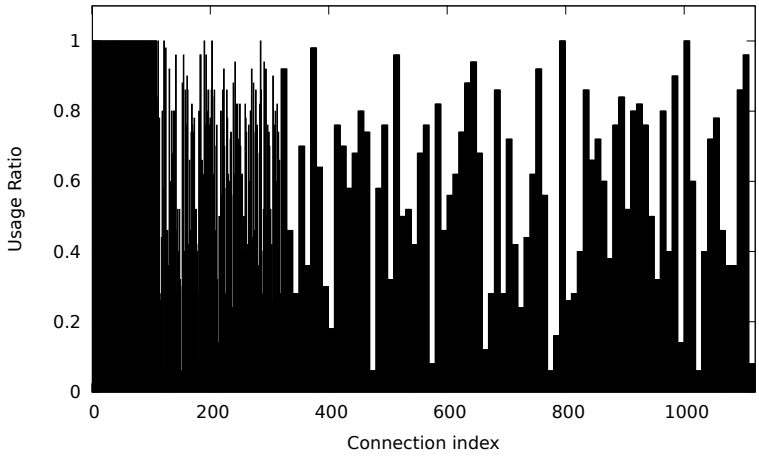


Figure 6: Connection usage ratios for all SLIM RNN connections after learning 227 out of the 340 total self-invented tasks, 50 of them non-compression tasks forming the so-called task repertoire, the rest compression tasks. The *usage ratio* on the y-axis is the number of repertoire tasks using the connection, divided by the number of repertoire tasks. This ratio is 1 for the first 110 connections, which are frequently used outgoing connections from task and online inputs. 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 help to speed up task search in later stages.

A Appendix: Implementation details

The SLIM RNN used for Experiment 2 (fovea control) is constructed as follows:

Let the number of input, output and state neurons in the network be *n_input*, *n_output* and *n_state*, respectively. Let *nb_comp* = number of computation blocks each with *block_size* neurons. Thus there are *nb_comp*×*block_size* computation neurons in the network.

The network is wired as follows. Each task input neuron is connected to *nb_comp* computation neurons at random. Each online input neuron is connected to *nb_comp*/10 neurons at random. Each internal state neuron receives connections from *nb_comp*/2 random computation neurons. The halt neuron recieves connections from *nb_comp*/2 random computation neurons. *nb_comp*×*n_output* random computation neurons are connected to random output neurons. Each neuron in each computation block is randomly connected to *block_size* other computation neurons.

We used *nb_comp = 20*, *block_size = 4*, and *n_state = 3* with *n_input = 25* and *n_output = 8* for the fovea control task. The *halt-threshold* was set to 3, and the WITAS and fovea control *win-threshold*s were set to 0.00001. All connection weights were initialized to random values in $[-1, 1]$. The cost of using a connection (consuming part of the *time_budget*) was set to 0.1 for all connections. The mutation rule was as follows. For non-compression tasks, the network is first run using the new task inputs to check if the task can already be solved by generalization. If not, we randomly generate an integer number m between 1 and 1/50th of all connections used during the unsuccessful run, and randomly change m weights by adding to them a uniformly random number in $[-0.5, 0.5]$. For compression tasks, we randomly generate a number m between 1 and 1/50th of all connections used for any of the tasks in the current repertoire, and randomly modify m of those connections.

The time budget fraction available to check whether a candidate task is not yet solvable by s_{i-1} was chosen randomly between 0 and *time_budget*/2. For compression tasks, the sum of squared weights had to decrease by at least a factor of 1/1000 to be acceptable.

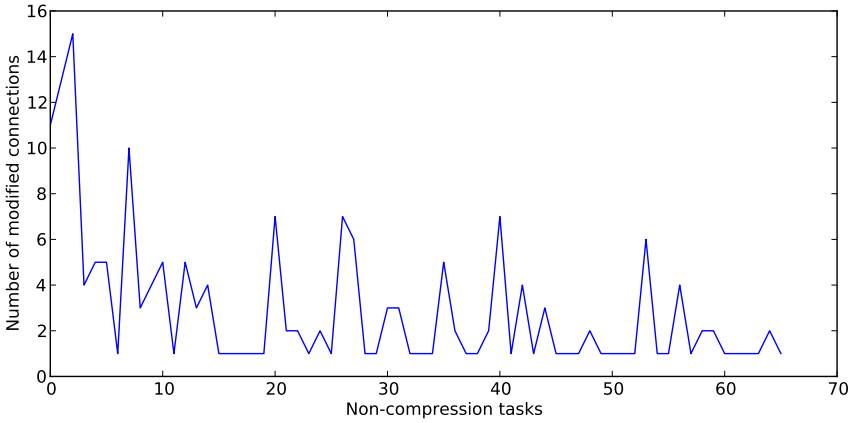


Figure 7: For each self-invented non-compression task, we plot the number of modified SLIM NN weights needed to learn it without forgetting solutions to old tasks. During task search, the number of connections to modify is chosen randomly. Once the growing repertoire has reached a significant size, however, *successfully* learned additional tasks tend to require few weight changes affecting few previous tasks (especially tasks with computationally expensive solutions). This is due to POWERPLAY's bias towards tasks that are fast to find and validate on the entire repertoire. See text for details.

Acknowledgments

POWERPLAY [24] and self-delimiting recurrent neural networks (SLIM RNN) [25] 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. *Journal 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] V. V. Fedorov. *Theory of optimal experiments*. Academic Press, 1972.

[6] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

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

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

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

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

[11] 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.

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

[13] 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.

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

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

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

[17] J. Schmidhuber. Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE press, 1991.

[18] 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.

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

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

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

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

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

[24] 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], 2011.

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

[26] 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.

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

[28] R. K. Srivastava, B. R. Steunebrink, M. Stollenga, and J. Schmidhuber. Continually adding self-invented problems to the repertoire: First experiments with POWERPLAY. In *Proceedings of the 2012 IEEE Conference on Development and Learning and Epigenetic Robotics IEEE-ICDL-EPIROB*. IEEE, 2012.

[29] 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.

[30] R. S. Sutton, D. Precup, and S. Singh. Intra-option learning about temporally abstract actions. In *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufman, 1998.

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

[32] M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1998.