J. Yaghob (Ed.): ITAT 2015 pp. 127–134
Charles University in Prague, Prague, 2015

Platform for Rapid Prototyping of AI Architectures

Peter Hroššo, Jan Knopp, Jaroslav Vítků, and Dušan Fedorčák

GoodAI, Czech Republic
contact author: peter.hrosso@keenswh.com

Abstract: Researching artificial intelligence (AI) is a big endeavour. It calls for agile collaboration among research teams, fast sharing of work between developers, and the easy testing of new hypotheses. Our primary contribution is a novel simulation platform for the prototyping of new algorithms with a variety of tools for visualization and debugging. The advantages of this platform are presented within the scope of three AI research problems: (1) motion execution in a complex 3D world; (2) learning how to play a computer game based on reward and punishment; and (3) learning hierarchies of goals. Although there are no theoretical novelties in (1,2,3), our goal is to show with these experiments that the proposed platform is not just another ANN simulator. This framework instead aims to provide the ability to test proactive and heterogeneous modular systems in a closed-loop with the environment. Furthermore, it enables the rapid prototyping, testing, and sharing of new AI architectures, or their parts.

1 Introduction and Related Work

The recent boom in the field of artificial intelligence (AI) was brought on by advances in so-called narrow AI, represented by highly specialized and optimized algorithms designed for solving specific tasks. Such programs can even sometimes surpass human performance when solving the single problem for which they were created. But these narrow AI programs lack one feature which has been so far widely omitted, partly due to its overwhelming difficulty: generality.

In order to compensate for this deficiency, the field of artificial general intelligence (AGI) is bringing the focus back to broadening the range of solvable tasks. The ultimate goal of AGI is therefore the creation of an agent which can perform well (at human level or better) at any task solvable by a human. For a more detailed description of AI/AGI, see e.g. [23].

Pursuing such a goal is a hard task. According to the scientific method – the only guideline we have – we need to come up with new theories, design experiments for testing them, and evaluate their results. Such a cycle needs to be repeated often, because it can be expected to reach more dead ends than breakthroughs. We don't know how to increase the rate of coming up with new ideas, but what can be improved is the efficiency of research. What we need is better tools which will simplify the implementation of new theories, speed up experiments, and help us understand the results better by visualizing obtained data.

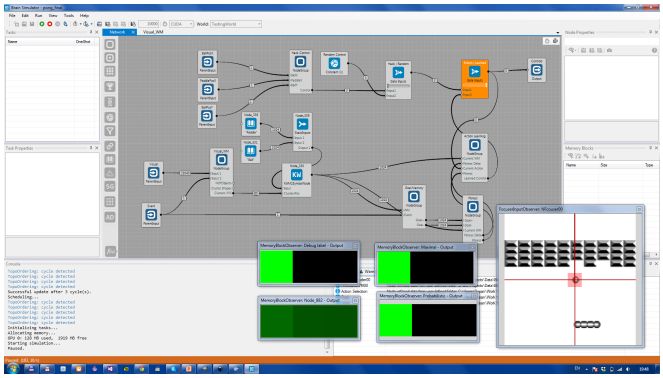


Figure 1: Print-screen of our simulation platform.

In this article, we would like to present our attempt to create such a tool. We here introduce a platform which allows:

- Easy prototyping of new models and fast sharing of existing ones (Sec. 4.1)
- Control of an agent in an environment on top of classic data processing (Sec. 3.3)
- Modular approach – seamless connecting of models inside a greater architecture (Sec. 2.1)
- Various tools for the visualization of data (Fig. 2)
- Simplified debugging (Sec. 2.2)
- User Friendly GPU programming (Sec. 2.1)
- Scalable due to GPU parallel computation
- Support of several scenarios such as an agent in an environment, classification, tic-tac-toe *etc.*

Our platform also includes a variety of visualization tools and enables easy access to diverse data sets not only for tasks such as image classification or recognition, but also scenarios where an agent interacts with its environment. Last but not least, it is open source and freely available under a non-commercial license[1]. The primary goal of this tool is easy collaboration among both specialists and laymen for developing novel algorithms, especially in the field of AI.

There are tools, languages, and libraries that are good in particular areas. In the research community, widely

[1]The platform is available as Brain Simulator at http://www.goodai.com/brainsimulator

used are Matlab [9] and Python [17] for prototyping by code, platforms that aim at high level graphical modeling (Simulink [20], Software Architect [8]), data analysis (Azure [10], Rapid Miner [18]), advanced visualization (ParaView [15]), rich graphical user interface (Blender [2], Maya [1]), modular computation (ROS [19]), or specific libraries for sharing [7] and parallel computation [3]. Each of these instruments is important in their specific domain, but there is none which would cover under one roof the most prominent features of all of those mentioned. Our platform is an attempt to fill this niche, and offers both high-level graphical coding and possibly, but not necessarily, also low-level (i.e. CUDA [3]) programming.

There are several tools for simulating neural networks (NN). Nengo [11] or OpenNN [14] focus on experimenting with all possible modifications of NN. Unfortunately, they either lack in visualization or focus on over-specific design approaches. Moreover, usage of these tools often requires extensive programing knowledge and installation of extension packages [4, 13]. In contrast, other tools that provide rich visualization focus purely on the functions of our brain. For example, PSICS [16] uses 3D synapse visualization to show data flow in parts of the brain, DigiCortex [6] nicely visualizes spike activations of the whole brain in time, and Cx3D [5] simulates growth of the cortex in 3D. Extensive comparison of various neural network simulators, including our platform (Brain Simulator), can be found in [12].

It is worth noting that the proposed platform is not limited to the design of neural networks only. Any algorithm useful for AI, machine learning, or control can be incorporated (various mathematical transformations, filters, PID controller, image segmentation, hashing functions, dictionary, *etc.*, are already included). The heterogeneous character of the platform is its main advantage.

Throughout this work, we describe our platform in the following Sec. 2. In Sec. 3, tasks where we show advantages of the platform are introduced. In Sec. 4, we discuss the experience with our tool and its advantages and weakness in the testing scenarios. The paper is concluded in Sec. 5.

2 Simulation Platform

Our modus operandi reflects our goals - we are aiming for a modular cognitive architecture, so we needed an environment which would efficiently support the whole life-cycle of experiments, starting with the testing of already existing algorithms, going through the design of a new algorithm, and ending with a results evaluation. We developed a platform where various algorithms from machine learning and narrow AI are available. It is easy to pick some, connect them, and start experimenting effortlessly. Agile development requires frequent testing of new hypotheses, which is facilitated by an easy way of prototyping new modules for the platform as well as their fast training and evaluation (accelerated on GPU) on data. After the experiment is

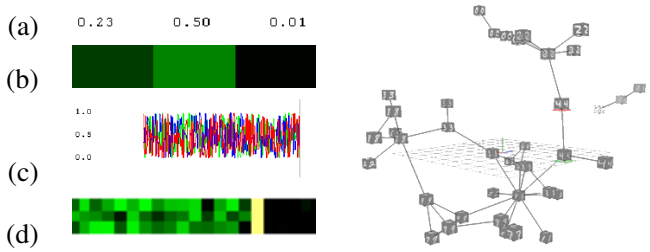


Figure 2: Visualization of data. Left: memory block of a 1×3 matrix can be visualized as: (a) value, (b) color-map, (c) value of each element in time, or (d) value as a color-map in time. Right: more complicated task-specific visualizations can be implemented too, i.e.growing neural gas [26].

running, it often happens that its outcome is not what was expected. Such a simulation platform could not be imagined without a tool for **runtime analysis of algorithms**. For this purpose various data observers can be displayed so the experimenter can visualize the computed data, evaluate performance of the model, and **change its parameters during runtime if needed**.

Our platform is tailored to suit two different points of view of the architecture development process:

- A user who desires quick architecture modeling and needs fast access to already existing state-of-the-art modules (such as PCA, NN, image pre-processing *etc.*) to experiment with. This perspective **requires no coding** and it's done through graphical modeling. Furthermore, it is often crucial to have good insight into the running model, and thus a large set of visualization tools is available (Fig. 2).

- On the other hand, a researcher/developer often requires the creation of a new module or the import of an already existing library. Our API provides an easy way for such a **module to be created and added** to the inner shared repository. Moreover, the API offers an opportunity to hook the code to the GUI and bring needed interactivity. Finally, the API defines a rigid interface, ensuring that the new module will be compatible with other modules.

The platform was designed to meet both needs. It is important to distinguish between them as a user can be a person interested in machine learning, but less experienced in programming. Our platform can be a good starting point, and the learning curve should therefore be smooth enough to bring the person in effortlessly.

From the experienced researcher/developer point of view, the platform should provide a convenient set of tools that can help with the development of novel algorithms and/or be able to envelope existing work into module that can be easily shared among a team.

Finally, the community-driven approach renders itself very powerful and we believe that it can speed up the research vastly. For this reason, we are **planning to in-build**

a "module market" to allow for the sharing of state-of-the-art research results between many co-working teams.

2.1 Platform Meta Model

There are three basic concepts defined in the meta model: *a node, a task and a memory block*. The node encapsulates a functional block or algorithm that can "live" on its own (e.g. matrix operations, data transformations, various machine learning models, *etc.*). A node needs a memory for its function. The memory is organized into a set of memory blocks that are aggregated inside the node. Some of these memory blocks can be designated as output blocks and others as input blocks. The connection between input and output memory blocks is provided by the user.

From the functional point of view, the node behaviour can usually be divided into a set of tasks where each task is a part of the realized algorithm. Both nodes and tasks can define a set of parameters. Usually, node parameters describe structural properties (i.e. size of memory blocks) whereas task parameters affect behavior. At present, the memory model is constant during the simulation, and therefore structural properties are editable only in design time. On the other hand, it is useful to **change task parameters during simulation and observe changes** in behavior of the algorithm/node.

Memory blocks are located at GPU (device memory) and every task can be seen as a collection of kernel calls (methods executed on GPU). If two nodes are connected in the GUI, it means that they have a pointer to the same memory block (input in one node, output in the other).

If one requires dynamically allocated memory, the user can either define a memory block that is large enough, or implement the node only for the CPU (which is more flexible than GPU) using all data structures supported by C#. The only mandatory requirement is usage of input/output memory blocks.

All concepts described above can be easily implemented through rich API that is provided. The actual implementation relies heavily on annotated code describing various aspects of the model (UI interactivity, constraints, persistence, *etc.*). It allows the user to be extremely efficient in creating model prototypes. Sometimes, this can lead to unreadable, over-annotated code which is hard to maintain but this can be eliminated by applying standard software design patterns like MVC when needed.

2.2 Computation

As described above, the prototyped model forms an oriented graph with nodes and data connection edges. As the connections between nodes can be any of $M \rightarrow N$ and recurrent connections are also possible, the resulting graph can be very complex. Moreover, the usual model is connected to the world node from which "perception" inputs are taken, and control outputs are passed, forming the main loop of the simulation.

Before running the simulation, the order of nodes execution needs to be evaluated. There are other aspects that level the problem up (e.g. inner cycles, clustering and balancing of the model in HPC environment) but it usually boils down to various forms of dependency ordering, cycle detection, or the job shop problem [34]. Solving these tasks is automated and user/developer assistance is usually discouraged, but there are use cases where user aid is necessary or can simplify the problem substantially.

There is also another view of the problem of execution order when faced in the area of machine learning. It turns out that many of ML methods are surprisingly noise resistant (i.e.neural nets). Therefore, *if approached with caution*, one can run the model asynchronously and let inner parts of the model deal with sometimes temporally inconsistent data. We made some experiments and the preliminary results show that relatively complex models can be run completely without synchronization.

Another aspect of the model execution is GPU enhanced computation which can speed up the simulation substantially. The main purpose of our simulation platform is fast prototyping and testing of hypotheses. With increasing generality the efficiency usually decreases, so one should not expect top execution speed from our simulation platform. The devised practice is to design, test, and analyze new architectures, and once the final model is tested and working, it can be replaced by a specialized, highly optimized implementation still within the platform environment. Finally, it can be argued that the overall time necessary to get from an idea to the final product is much shorter compared to the classic approach of writing a specific program from scratch for each new experiment.

An important part of the development process is the easy visualization of what is happening at each part of the designed system. This is especially important for debugging as the most frequent problem is due to the difference between what the programmer thinks the program should do and what it does in reality. In addition to the variety of observers that have already been discussed (Fig. 2), the platform contains its own debugger, where one can walk through the execution of all components used in the model.

3 Testing Scenarios

Whether the ultimate goal of AGI (a general autonomous machine) is achievable or not [35], researchers focus on its sub-goals such as learning how to play games [37], *etc.* One of the prominent building blocks for these subgoals are neural networks in the form of deep learning and CNN, and which have recently made big progress in speech recognition [40], computer vision [33], medical analyses [43], or language translation [28]. Le and colleagues [36] used a deep network to learn in unsupervised manner what an ordinary "cat" looks like only by watching youtube videos. While NN can also learn how to play simple games [32, 37], they usually fail in structured

problems which demand learning hierarchies or chains of goals. From this perspective, it seems promising to focus on machines which can control another machine, such as NN that learn how to control a Turing Machine [27]. They designed a neural network which learns a procedure to control a Turing Machine to sort numbers.

As the goal of this paper is to provide a tool that shortcuts the research path to an autonomous machine, we will show how it performs on three selected AI tasks solved by our team: learning motion control, game playing of the Atari game Breakout, and learning hierarchies of goals. Our solutions are highly inspired by current machine learning literature with a stress on the usage of neural networks, which are one of the basic building blocks for bigger architectures.

The first experiment (Sec. 3.1) will demonstrate how our platform can be connected to an external source of input data and how various modules of narrow AI can be combined together to form a functioning system which can drive a robot in a virtual world with simulated physics.

The second experiment (Sec. 3.2) will be situated in much simpler simulation environment – an Atari [38] game called Breakout. In this experiment a more advanced adaptive system will be showcased. The system works directly on raw image input. It takes advantage of the semantic pointer architecture [25] for representing its perceptions and for converting them into a long term memories such as goals. This knowledge is then used for learning necessary actions for playing the game.

In the third experiment (Sec. 3.3), we move a bit higher in the level of abstraction. The presented problem consists of an agent in a simple 2D environment which needs to satisfy a chain of preconditions before reaching a reward, such as if the agent wants to turn on a light, it needs to press a switch, but to get to the switch he also needs to overcome an obstacle (a door controlled by another switch). The task is solved by hierarchical reinforcement learning [30].

To clarify why we selected these experiments, one could imagine the three systems as parts of a future higher-order cognitive architecture where they will work together. The system from the first experiment could be thought of as a basic motoric and sensory system driven by reflexes and higher-level commands. These would come from the system used in the second experiment, which would allow the agent to learn how to reach a specific goal. And finally, the third system should discover the hierarchy of goals and preconditions, and thus could resemble a simplified version of the agent's central executive. Such connection of the systems remains for our future work.

3.1 SE Robot

We took advantage of a sandbox game called Space Engineers [21], which provides a physically realistic 3D environment where various structures can be built. We built a six-legged robot within the game and connected it bi-directionally with our simulation platform. In one direction the game sends visual data from the robot's view and a description of the state of the robot's body. In the other direction motoric commands from our control module inside the simulation platform are sent to the robot, which executes them in the game.

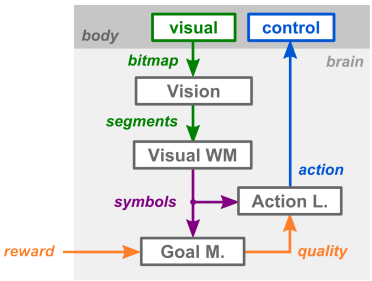
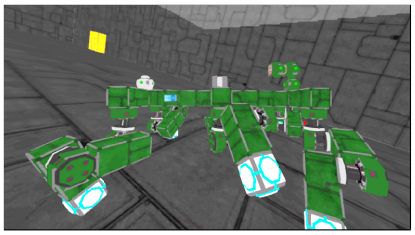


Figure 3: Overall architecture. Raw visual signals are processed into symbols, which are then added to the working memory. States corresponding to reward and punishment are accumulated and later used as teaching signals for training the action selection network.

The control module was trained to associate visual input with motor commands in a supervised way. The associative memory was implemented with a Self-Organizing Map [31], which found the most similar representative of the received input in the visual memory and returned the associated high-level motoric command (turn left/right, move forward/backward). These high-level commands were then unrolled into sequences of body states consisting of joint angles of all of the robot's limbs using a recurrent neural network (RNN) [39]. These body states were afterwards used as waypoints for a control RNN which was trained to act as an inverse dynamics model of the robot's body. In order to reach a specified waypoint, the control network generated full motoric inputs to the robot - the desired angular velocities of joints.



The training phase consisted of a mentor leading the robot from various starting locations towards a goal location in the environment, which was identified by an easily distinguishable 3D symbol located on that position. The mentor was implemented by a hard-coded navigation system. In this way the hexapod was trained to look for the goal symbol and when it appeared in the robot's field of view, to navigate successfully towards the destination through the environment.

3.2 Atari Game

The Breakout game was chosen as our second testing scenario. The game consists of a ball, a paddle, and

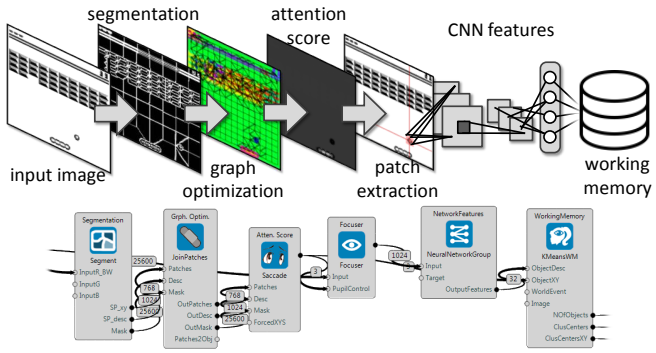
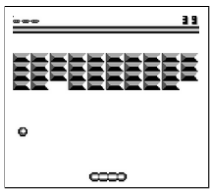


Figure 4: Top: Image processing. First, an input image is segmented into super-pixels (SP) using SLIC [22]. Second, each SP is connected with its neighbors and nearby SP are assigned into the same object id. Third, the attention score (s_A) is estimated for each object. Fourth, features are estimated for the object with the highest s_A by a hierarchy of convolutions and fully-connected layers. Fifth, the object features are clustered into Visual Words [41] to constitute a "Working Memory". Bottom: Corresponding implementation in our platform.

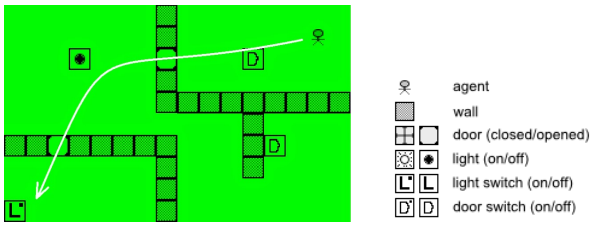
bricks. The ball bounces from walls, can destroy bricks, and can fall to the ground, for which the player is penalized by losing a life. After losing 4 lives, the game is over. When all bricks are destroyed, the player successfully finishes the level and enters the next one consisting of a different arrangement of bricks. The player has three actions available which accelerate the paddle to the right, to the left, or decelerate. Even though our modular approach uses pure unstructured data input (raw image as in [37]), it later extracts the structure, so we can understand the inner workings of the model as opposed to the cited work. The architecture of the system consists of four main parts: image processing, working memory, accumulators of reward and penalty, and an action selection network (Fig. 3).



Relevant information about the objects is extracted from the raw bitmap in the Vision System (Fig. 4).

Working memory (WM) is the agent's internal representation of the environment. It contains all of the objects detected by vision. WM is kept up to date by adding new objects which haven't been seen yet, and by updating those already seen. The identity of objects is detected through a comparison of visual features. Contents of the working memory are transformed into a symbolic representation and passed to the goals memory and action selection network.

The goals memory is trained by accumulating states associated with reward and punishment in their respective semantic pointers, goal$^+$ and goal$^-$. These are then used for evaluating the quality of game-states, which is necessary for training the action selection network.



(a) The goal (b) Legend

Figure 5: Multiple goals. (a) The agent's current goal is to reach the light switch and turn on the lights. (b) objects of the environment

Details of the vision system and semantic point architecture are described in Appendix II.

3.3 2D World with Hierarchical Goals

In previous testing scenarios we wanted to test if our system was able to coordinate complex motoric commands in a 3D environment, learn simple goals, and act towards maximizing the received reward. Our goal for the third scenario is to increase the generality of the designed system to enable identification and satisfaction of chained preconditions before the final goal can be reached.

We present a task, consisting of a simple 2D world, where a single source of reward is located – a light bulb, which starts in the "off" position and should be turned on by the agent. This can be achieved by pressing a switch, but the switch is hidden behind a locked door. The door can be unlocked through a switch, but this switch is hidden behind another locked door. It would be possible to chain the preconditions further in this manner, but without the loss of generality we use only two locked doors with two matching switches. The setup can be seen in Fig. 5.

We approached this problem by employing HARM (Hierarchical Action Reinforcement Motivation system) [30]. It is an approach based on a combination of a hierarchical Q-learning algorithm [42] and a motivation model. The system is able to learn and compose different strategies in order to create a more complex goal.

Q-learning is able to "spread information about the reward" received in a specific state (e.g. the agent reaching a position on the map) to the surrounding space, so the brain can take proper action by climbing the steepest gradient of the Q function later. However, if the goal state is far away from the current state, it might take a long time to build a strategy that will lead to that goal state. Also, a high number of variables in the environment can lead to extremely long routes through the state space, rendering the problem almost unsolvable.

There are several ideas that can improve the overall performance of the algorithm. First, this agent *rewards itself for any successful change to the environment*. The motivation value can be assigned to each variable change so the agent is constantly motivated to change its surroundings.

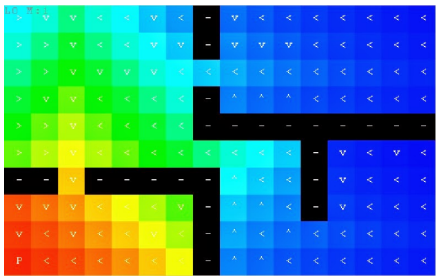


Figure 6: Learnt strategy. Visualization of the agent's knowledge for a particular task, which changes the state of the lights. It tells the agent what to do in order to change the state of the lights in all known states of the world. The heat map corresponds to the expected utility ("usefulness") of the best action learned in a given state. A graphical representation of the best action is shown at each position on the map.

Second, for each variable that the agent is able to change, it *creates a Q-learning module* assigned to the variable (e.g. changing the state of a door). Therefore, it can learn an underlying strategy defining how this change can be made again. In such a system, a whole network of Q-learning modules can be created, where each module learns a different strategy.

Third, in order to lower the complexity of each subproblem (strategy), the brain can analyze its "experience buffer" from the past and eventually drop variables that are not affected by its actions or are not necessary for the current goal (i.e. strategy to fulfill the goal).

A mixture of these improvements creates a hierarchical decision model that is built online (first, the agent is left to (semi-)randomly explore the environment). After a sufficient amount of knowledge is gathered, we can "order" the agent to fulfill a goal by manually raising the motivation that corresponds to a variable that we want to change. The agent then will execute the learned abstract action (strategy) by traversing the network of Q-learning modules and unrolling it into a chain of primitive actions that lie at the bottom.

4 Discussion

Throughout the work on the testing scenarios (Sec. 3) we have observed several advantages and weakness of the platform. In this section, our experience with the usage of the platform is discussed. The discussion is focused especially on the end-user experience, i.e. experience of a person that did not develop the platform but wants to use it for solving her problem.

First, a list of identified features is presented, and then further experience is described.

+ Fast and easy online observation and interaction with the simulation.

+ Created modules can be easily understood and shared with collaborators due to the same interface. Moreover, the persistence capabilities allow easy sharing of whole models (projects) and merging of them together.

+ It is easy to replace an existing module with its improved version, as the architecture is separated from the implementation. As the backward compatibility becomes crucial at this point, the inner versioning system was implemented.

+ Provided interface drives users to follow design patterns when developing low-level optimized modules.

- Current version runs on MS Windows only, but a port to MacOS and Linux is planned for the future.

- The user has an option to either develop optimized modules in code (CUDA [3] or C#) or use the graphical interface for connecting existing modules into bigger architectures. There is no middle layer which would support scripting.

4.1 Experience of Newcomers

The expertise of people that have started to use our platform varies from C++ experts to Matlab users only. We found that users with a very short training can connect existing modules into simple architectures (like neural network MNIST image recognizer) as the graphical modeling is somehow natural and easy to understand. The linear learning curve of newcomers is supported by a video tutorial as well as several examples of how to implement simple and more advanced tasks[2].

For development of new modules, it is necessary to understand a programming language (C#, C++, CUDA) at the basic level at least. Once the definitions of inputs, outputs, tasks, and kernels (four lines of code each) are understood, developers soon start creating their own nodes. Their learning curve then equals learning how to use a new library.

4.2 Our Observations on the Testing Scenarios

In the first scenario (Sec. 3.1), we have shown that our platform successfully connects with the open source game Space Engineers [21]. Modules created in the platform controlled the hexapod in the world of the game. It was the understanding of the game's communication module that took the most time in this case. Otherwise the development of the controller did not raise any challenges for the platform.

The second scenario (Sec. 3.2) consisted of several modules that were developed independently. We found it extremely useful that each module communicates with others using only the pre-defined interface (memory blocks) that correspond to a sketched diagram (i.e. Fig. 4). Modules were merged into one big architecture right before the deadline without any complications. As the final model was quite large and performance-demanding,

[2]Documentation available at `http://docs.goodai.com/brainsimulator/`

we were forced to profile, find bottlenecks, and optimize in the process. It was extremely useful to visualize data flowing between (and inside) the modules.

In the third scenario (Sec. 3.3), HARM constituted a single module with complex insides. Therefore this scenario presented an ideal example for designing a number of task-specific visualization tools (for example, the agent's knowledge in Fig. 6).

5 Conclusion

We have presented a platform for prototyping AI architectures. The platform is tailored both for users with no mathematical/programming background but with a high desire to experiment with AI modules, and for researchers/developers who want to improve and experiment with their existing state-of-the-art techniques.

To show the usage of our platform we have presented three development scenarios: linkage with a 3D game world and controlling an agent there; playing an Atari game using the raw bitmap input processed by computer vision techniques, attention model, and semantic pointer architecture; and learning a complex hierarchy of goals.

The proposed platform opens up possibilities to share ideas not only within the community but also with non-experts who can boost the research via rapid testing, or utilize fresh, out-of-the-box solutions. There is also the prospect of support from the open source community - if not directly in the development, then at least in assessing missing features, so we can incorporate them and thus provide a tool that can be used at many levels of expertise.

We believe that by providing an open platform for AI and ML experiments along with a smooth learning curve, we can bring together many enthusiasts across different fields of interest, potentially leading to unexpected advancements in research.

Acknowledgement. This material is based upon work supported by GoodAI and Keen Software House.

References

[1] Autodesk maya. Available at http://www.autodesk.com/products/maya/overview.

[2] Blender. Available at https://www.blender.org/.

[3] CUDA. Available at https://developer.nvidia.com/cuda-zone.

[4] CVX: Software for Disciplined Convex Programming. Available at http://cvxr.com/.

[5] Cx3D: Cortex simulation in 3D. Available at http://www.ini.uzh.ch/~amw/seco/cx3d/.

[6] DigiCortex: Biological neural network simulator. Available at http://www.dimkovic.com/node/1.

[7] GitHub. Available at https://github.com/.

[8] IBM Rational Software Architect. Available at http://www.ibm.com/developerworks/downloads/r/architect/index.html.

[9] MATLAB: The language of technical computing. Available at http://www.mathworks.com/products/matlab/.

[10] Microsoft Azure. Available at http://azure.microsoft.com/en-us/.

[11] The nengo neural simulator. Available at http://nengo.ca/.

[12] Neural networks simulators. Available at https://goo.gl/hRf4KA.

[13] OpenCV: Open source computer vision. Available at http://opencv.org/.

[14] OpenNN: Open neural networks library. Available at http://www.intelnics.com/opennn/.

[15] ParaView. Available at http://www.paraview.org/.

[16] PSICS: The parallel stochastic ion channel simulator. Available at http://www.psics.org/.

[17] Python. Available at https://www.python.org/.

[18] Rapid miner. Available at https://rapidminer.com/.

[19] ROS. Available at http://www.ros.org/.

[20] Simulink: Simulation and model-based design. Available at http://www.mathworks.com/products/simulink/.

[21] Space Engineers, open source code. Available at https://github.com/KeenSoftwareHouse/SpaceEngineers.

[22] Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., Susstrunk, S.: Slic superpixels compared to state-of-the-art superpixel methods. PAMI (2012), 2274–2282

[23] Ben, G.: Artificial general intelligence: concept, state of the art, and future prospects. Journal of Artificial General Intelligence **5** (2014), 1–48

[24] Bishop, C. M.: Pattern recognition and machine learning. Springer, 2006

[25] Eliasmith, C.: How to build a brain: a neural architecture for biological cognition (Oxford Series on Cognitive Models and Architectures). Oxford University Press, 2013

[26] Fritzke, B.: A growing neural gas network learns topologies. In: NIPS, 1995

[27] Graves, A., Wayne, G., Danihelka, I.: Neural turing machines. CoRR, 2014

[28] He, X., Gao, J., Deng, L.: Deep learning for natural language processing and related applications (tutorial at ICASSP). ICASSP, 2014

[29] Huang, F. J., Boureau, Y. L., Lecun, Y.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. In: CVPR, 2007

[30] Kadlecek, D., Nahodil, P.: Adopting animal concepts in hierarchical reinforcement learning and control of intelligent agents. In: Proc. 2nd IEEE RAS & EMBS BioRob, 2008

[31] Kohonen, T., Schroeder, M. R., Huang, T. S.: Self-organizing maps. 3rd edition, 2001

[32] Koutník, J., Cuccu, G., Schmidhuber, J., Gomez, F.: Evolving large-scale neural networks for vision-based reinforcement learning. In: GECCO, 2013

[33] Krizhevsky, A., Sutskever, I., Hinton, G. E.: Imagenet classification with deep convolutional neural networks. In: NIPS, Curran Associates, Inc., 2012

[34] Tsaia, M. -J., Chang, H.-Y., Huang, K. -C., Huanga, T -C., Tung, Y. -H.: Moldable job scheduling for hpc as a service with application speedup model and execution time information. Journal of Convergence, 2013

[35] Kurzweil, R.: The singularity is near: when humans transcend biology, 2006

[36] Le, Q., Ranzato, M.'A., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., Ng, A.: Building high-level features using large scale unsupervised learning. In: ICML, 2012

[37] Volodymyr et al. Mnih: Human-level control through deep reinforcement learning. Nature **518** (2015), 529–533

[38] Naddaf, Y.: Game-independent AI agents for playing Atari 2600 console games. Masters, University of Alberta, 2010

[39] Rojas, R.: Neural networks: a systematic introduction. Springer-Verlag New York, Inc., New York, NY, USA, 1996

[40] Sak, H., Vinyals, O., Heigold, G., Senior, A., McDermott, E., Monga, R., Mao, M.: Sequence discriminative distributed training of long short-term memory recurrent neural networks. In: Interspeech, 2014

[41] Sivic, J., Zisserman, A.: Video Google: A text retrieval approach to object matching in videos. In: ICCV **2** (2003), 1470–1477

[42] Sutton, R. S., Barto, A. G.: Introduction to reinforcement learning. MIT Press, Cambridge, MA, USA, 1st edition, 1998

[43] Xu, Y., Mo, T., Feng, Q., Zhong, P., Lai, M., Chang, E. I.-C.: Deep learning of feature representation with multiple instance learning for medical image analysis. ICASSP, 2014

Appendix I: Details of Image Processing

Unlike the major stream of Computer Vision, our approach has to be *unsupervised* without any training data. Thus, we have *no* prior knowledge and the system has to learn everything on-the-fly. Our system is a pipeline visualized and implemented in Fig. 4. It consists of the following parts: The input image is first segmented into a set of super-pixels [22] (SP). Then, each SP is connected to its vicinity constituting a graph where nodes are SPs and edges connect neighboring SPs. SPs with similar color are merged into connected components. Note that while more SP speeds up the segmentation, it slows down the graph optimization algorithm, see Fig. 7. Once we have object proposals, we estimate an attention score (s_A) for each object, $s_A(\mathbf{o}_i) = \psi_{time}(\mathbf{o}_i) + \psi_{move}(\mathbf{o}_i)$, where $\psi_{time}(\mathbf{o}_i)$ is time since we have focused on the object $\mathbf{o}_i$, $\psi_{move}(\mathbf{o}_i)$ is the object's movement. The object with the highest s_A is selected[3] and its position together with its size define an image patch. The image patch is processed into a CNN features [29], and this feature representation

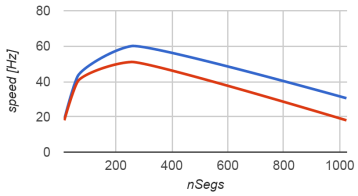


Figure 7: Performance of SLIC (blue) and SLIC+graph optimization (red) w.r.t the number of segments.

[3] Once the object is selected, its ψ_time is decreased and then it won't be selected in the next time step.

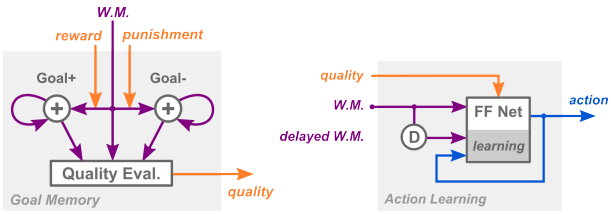


Figure 8: Left: **Goal$^+$ and goal$^-$.** States associated with received reward and punishment are accumulated in semantic pointers goal$^+$ and goal$^-$, respectively. Right: **Action learning.** Actions selection is learnt in a supervised way with fitness computed from goal$^+$ and goal$^-$ as a teaching signal.

is then clustered into a "Working Memory" (WM). The WM stores feature id together with the object position for the last 10 seen objects.

For CNN features, we used two convolutions layers of 8 and 5 neurons and patch sizes 5×5 followed by a fully-connected layer of 16 neurons. Learning converged in 6K iterations. We observed no performance improvement with bigger networks. WM was implemented as a simple K-means [24].

Appendix II: Semantic Pointer Architecture

As was already mentioned in section 1, one of the main features of our method is the semantic pointer architecture (SPA), which merges the symbolic and connectionist approach. Artificial neural networks are very powerful adaptive tools, but their usage usually comes at the expense of losing detailed insight into how exactly the task is solved. Such a drawback can be mitigated by using the SPA and its variable binding. It introduces composite symbols of the form X bind x, where X is the name of the variable and x is its value. It is then possible to train a network to perform complicated transforms such as:

$$V \otimes (X \otimes x + Y \otimes y) \rightarrow C \otimes (Y \otimes x) \qquad (1)$$

which could be interpreted as an action selection rule for the pong game:

$$Visual \otimes (Ball \otimes x_1 + Paddle \otimes x_2) \rightarrow \\ \rightarrow Move \otimes (Paddle \otimes x_1) \qquad (2)$$

If ball is seen at position x_1 and paddle at x_2, execute command 'move paddle to position x_1'. Without SPA it would be much harder to maintain understanding of the transformed symbols, if not entirely impossible.

Goals Memory. The accumulated states g^+ and g^- are used for calculation of quality q of the state x, using dot product '$\cdot$', $q = g^+ \cdot x - g^- \cdot x$, see Fig. 8 left.

Action Learning. Training of action selection (Fig. 8 right) is delayed by one simulation step to allow the system to observe results of its actions. Actions leading to higher quality states are labeled as correct, otherwise incorrect.