Online Evolution of Deep Convolutional Network for Vision-Based Reinforcement Learning

Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez

IDSIA, USI-SUPSI
Galleria 2
Manno-Lugano, CH 6928
{hkou,juergen,tino}@idsia.ch

Abstract. Dealing with high-dimensional input spaces, like visual input, is a challenging task for reinforcement learning (RL). Neuroevolution (NE), used for continuous RL problems, has to either reduce the problem dimensionality by (1) compressing the representation of the neural network controllers or (2) employing a pre-processor (*compressor*) that transforms the high-dimensional raw inputs into low-dimensional features. In this paper we extend the approach in [16]. The Max-Pooling Convolutional Neural Network (MPCNN) compressor is evolved online, maximizing the distances between normalized feature vectors computed from the images collected by the *recurrent neural network* (RNN) controllers during their evaluation in the environment. These two interleaved evolutionary searches are used to find MPCNN compressors and RNN controllers that drive a race car in the TORCS racing simulator using only visual input.

Keywords: deep learning; neuroevolution; vision-based TORCS; reinforcement learning; computer games

1 Introduction

Most approaches to scaling neuroevolution to tasks that require large networks, such as those processing video input, have focused on indirect encodings where relatively small neural network descriptions are transformed via a complex mapping into networks of arbitrary size [4, 7, 10, 13, 15, 23].

A different approach to dealing with high-dimensional input which has been studied in the context of single-agent RL (i.e. TD [25], policy gradients [24], etc.), is to combine action learning with an unsupervised learning (UL) preprocessor or "compressor" which provides a lower-dimensional feature vector that the agent receives as input instead of the raw observation [5, 8, 12, 17, 19–21]. The UL compressor is trained on the high-dimensional observations generated by the learning agent's actions, that the agent then uses as a state representation to learn a value function.

In [3], the first combination of UL and evolutionary reinforcement learning was introduced where a single UL module is trained on data generated by entire population as it interacts with environment (which would normally be discarded) to build a representation that allows evolution to search in a relatively low-dimensional feature space. This approach attacks the problem of high-dimensionality from the opposite direction compared to indirect encoding: instead of compressing large networks into short genomes, the inputs are compressed so that smaller networks can be used.

In [16], we scaled up this Unsupervised Learning – Evolutionary Reinforcement Learning (UL-ERL) approach to the challenging reinforcement learning problem of driving a car in the TORCS simulator using vision from the driver's perspective as input. The high-dimensional images were compressed down to just three features by a Max-Pooling Convolutional Neural Network (MPCNN; [2,22]) that allowed an extremely small (only 33 weights) recurrent neural network controller to be evolved to drive the car successfully. The MPCNN was itself trained separately off-line using images collected previously while driving the car manually around the training track.

In this paper, the feature learning and control learning are interleaved as in [3]. Both the MPCNN, acting as the sensory preprocessor (compressor), and the recurrent neural network controllers are evolved simultaneously in separate populations, with the images used to train the MPCNNs being taken from the driving trials of the evolving controllers rather than being collected manually *a priori*.

The next section describes the MPCNN architecture that is used to compress the high-dimensional vision inputs. Section 3 covers our method—the UL-ERL framework applied to visual TORCS race car driving domain. Section 4 presents the experiments in the TORCS race car driving, which are discussed in section 5.

2 Max-Pooling Convolutional Neural Networks

Convolution Neural Networks [6, 18] are deep hierarchical networks that have recently become the state-of-the-art in image classification due to the advent of fast implementations on graphics card multiprocessors (GPUs) [1]. CNNs have two parts: (1) a deep feature detector consisting of alternating *convolutional* and *down-sampling* layers, and (2) a classifier that receives the output of the final layer of the feature detector.

Each convolutional layer ℓ, has a bank of $m^\ell \times n^\ell$ filters, F^ℓ, where m^ℓ is the number input maps (images), I^ℓ, to the layer, and n^ℓ is the number of output maps (inputs to the next layer). The i-th output map is computed by:

$$I_i^{\ell+1} = \sigma\left(\sum_{j=1}^{m^\ell} I_j^\ell * F_{ij}^\ell\right), \quad i = 1..n^\ell, \ell = 1, 3, 5...,$$

where $*$ is the convolution operator, F_{ij}^ℓ is the i-th filter for the j-th map and σ is a non-linear squashing function (e.g. sigmoid). Note that ℓ is always odd because of the subsampling layers between each of the convolutional layers.

The downsampling layers reduce resolution of each map. Each map is partitioned into non-overlapping blocks and a value from each block is used in the output map. The *max-pooling* operation used here subsamples the map by simply taking the maximum value in the block as the output, making these networks Max-Pooling CNNs [2, 22].

The stacked alternating layers transform the input into progressively lower dimensional abstract representations that are then classified, typically using a standard feed-forward neural network that has an output neuron for each class. The entire network is usually trained using a large training set of class-labeled images via backpropagation [18]. Figure 2 illustrates the particular MPCNN architecture used in this paper. It should be clear from the figure that each convolution layer is just a matrix product where the matrix entries are filters and convolution is used instead of multiplication.

3 Method: Online MPC-RNN

The Online MPC-RNN method for evolving the controller and the compressor at the same time is overviewed in figure 1. The controller is evolved in the usual way (i.e. neuroevolution; [26]), but instead of accessing the observations directly, it receives feature vectors of much lower dimensionality provided by the unsupervised compressor, which is also evolved.

The compressor is trained on observations (images) generated by the actions taken by candidate controllers as they interact with the environment. Here, unlike in [16] where the compressor was pre-trained off-line, the loop is closed by simultaneously evolving the compressor using the latest observations (images) of the environment that are provoked by the actions taken by the evolving controllers.

For the compressor, a deep *max-pooling convolutional neural network* (MPCNN) is used. These networks are normally trained to perform image classification through supervised learning using enormous training sets. Of course, this requires *a priori* knowledge of what constitutes a class. In a general RL setting, we may not know how the space of images should be partitioned. For example, how many classes should there be for images perceived from the first-person perspective while driving the TORCS car? We could study the domain in detail to construct a training set that could then be used to train the MPCNN with backpropagation. However, we do not want the learning system to rely on task-specific domain knowledge, so, instead, the MPCNN is *evolved* without using a labeled training set. A set of k images is collected from the environment, and then MPCNNs are evolved to maximize the fitness:

$$f_{\mathrm{k}} = \min(D) + \mathrm{mean}(D), \tag{1}$$

where D is a list of all Euclidean distances,

$$d_{i,j} = \|\mathbf{f}_i - \mathbf{f}_j\|, \forall i > j,$$

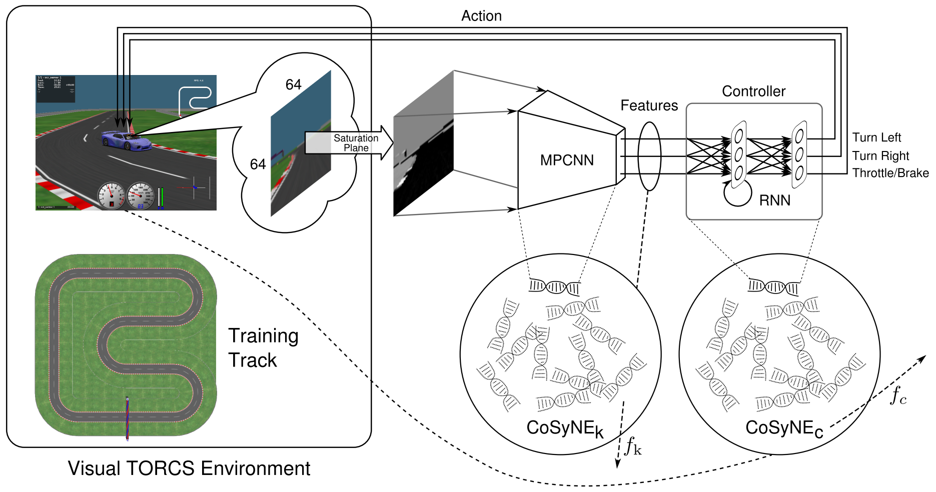

Fig. 1. Overview of the online TORCS controller evolution in the online UL-ERL scheme. At each time-step, a raw 64×64 grayscale (saturation plane) pixel image, taken from the driver's perspective, is transformed to features by the Max-Pooling Convolutional Neural Network (MPCNN) compressor. The features are used by the candidate RNN controller to drive the car by steering, accelerating and braking. Two evolutionary algorithms are interleaved to simultaneously optimize both the controller and compressor. The current implementation uses the CoSyNE algorithm: CoSyNE$_k$ for the MPCNN, CoSyNE$_c$ for the RNN controllers, using the best MPCNN found by CoSyNE$_k$.

between k normalized feature vectors $\{\mathbf{f}_1 \ldots \mathbf{f}_k\}$ generated from k images in the training set by the MPCNN encoded in the genome.

This fitness function forces the evolving MPCNNs to output feature vectors that are spread out in feature space, so that when the final, evolved MPCNN processes images for the evolving controllers, it will provide enough discriminative power to allow them to take correct actions.

4 Visual TORCS Experiments

The goal of the task is to evolve a recurrent neural network controller and MPCNN compressor that can drive the car around a race track.

The visual TORCS environment is based on TORCS version 1.3.1. The simulator had to be modified to provide images as input to the controllers (a detailed description of modifications is provided in [16]). The most important changes involve decrease of the control frequency from 50 Hz to 5 Hz, and removal of the "*3-2-1-GO*" sequence from the beginning of each race.

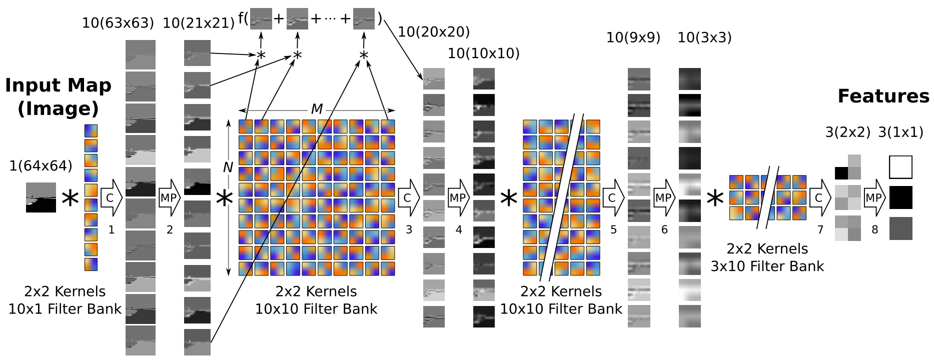

Fig. 2. Example Max-Pooling Convolutional Neural Network (MPCNN) with 8 layers alternating between convolution (C) and downsampling (MP; using max-pooling). The first layer convolves the input 64×64 pixel image with a bank of 10×10 filters producing 10 maps of size 63×63, that are down-sampled to 21×21 by MP layer 2. Layer 3 convolves each of these 10 maps with a filter, sums the results and passes them through the nonlinear function f, producing 10 maps of 20×20 pixels each, and so on until the input image is transformed to just 3 features that are passed to the RNN controller, see Figure 1.

4.1 Setup

The MPCNN compressors and RNN controllers are evolved in a coupled system, alternating every four generations between two separate CoSyNE [9] algorithms, denoted CoSyNE$_k$ for the MPCNNs, and CoSyNE$_c$ for the RNN controllers.

The process starts by first evolving the controllers. In each fitness evaluation, the candidate controller is tested in two trials, one on the track shown in figure 1, and one on its mirror image. A trial consists of placing the car at the starting line and driving it for 25 s of simulated time, resulting in a maximum of 125 time-steps at the 5 Hz control frequency. At each control step a raw 64×64 pixel image (saturation plane only), taken from the driver's perspective is passed through an initially random MPCNN compressor which generates a 3-dimensional feature vector that is fed into a simple recurrent neural network (SRN) with 3 hidden neurons, and 3 output neurons (33 total weights). The first two outputs, o_1, o_2, are averaged, $(o_1 + o_2)/2$, to provide the steering signal (-1 = full left lock, 1 = full right lock), and the third neuron, o_3, controls the brake and throttle (-1 = full brake, 1 = full throttle). All neurons use sigmoidal activation functions.

The fitness of the controllers use by CoSyNE$_c$ is computed by:

$$f_c = d - \frac{3m}{1000} + \frac{v_{max}}{5} - 100c \ , \qquad (2)$$

where d is the distance along the track axis measured from the starting line, v_{max} is maximum speed, m is the cumulative damage, and c is the sum of squares of

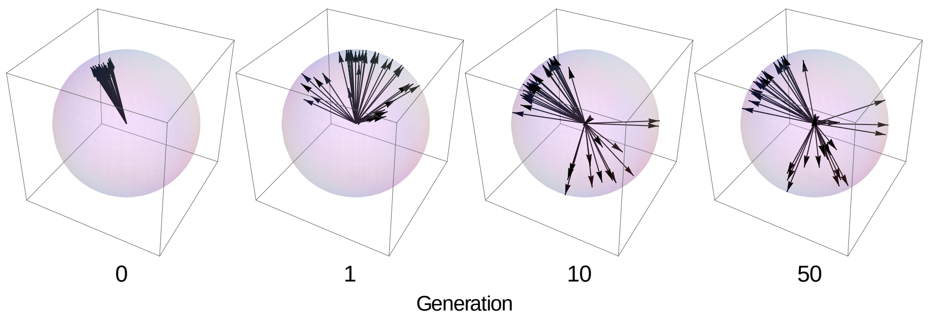

Fig. 3. Evolving MPCNN features. Each plot shows the feature vectors for each of the 40 training images collected at the given generation of CoSyNE$_k$ on the unit sphere. Initially (generation 0), the features are clustered together. After just a few generations spread out so that the MPCNN discriminates more clearly between the images. The features stabilize after generation 50.

the control signal differences, divided by the number of control variables, 3, and the number simulation control steps, T:

$$c = \frac{1}{3T} \sum_{i}^{3} \sum_{t}^{T} [o_i(t) - o_i(t-1)]^2. \tag{3}$$

The maximum speed component in equation (2) forces the controllers to accelerate and brake efficiently, while the damage component favors controllers that drive safely, and c encourages smoother driving. Fitness scores roughly correspond to the distance traveled along the race track axis. Each individual is evaluated both on the track and its mirror image to prevent the RNN from blindly memorizing the track without using the visual input. The original track starts with a left turn, while the mirrored track starts with a right turn, forcing the network to use the visual input to distinguish between tracks. The final fitness score is the minimum of the two track scores (equation 2).

After four generations of CoSyNE$_c$, CoSyNE$_k$ starts. First, a population of MPCNNs with 8 layers, alternating between convolution and max-pooling operations (see figure 2 and Table 1) is initialized with random kernel weights uniformly distributed between -1.5 and 1.5). The MPCNNs are then evolved for four generations using 40 randomly selected images from the previous CoSyNE$_c$ phase to compute the fitness function f_{k} (equation 1).

Each 64×64 pixel image is processed by a candidate MPCNN by convolving it with each of the 20 filters in layer 1, to produce 20 63×63 feature maps, each of which is reduced down to 21×21 by the first max-pooling layer (2). These 20 features are convolved again by layer 3, max-pooled in layer 4, and so on, until the image is reduced down to just 3 1-dimensional features which are fed to the controller. This architecture has a total of 3583 kernel coefficients (weights). All MPCNN layers used scaled *tanh* transfer function.

Layer	Type	m	n	#maps	map size
1	C	1	20	20	63×63
2	MP	-	-	20	21×21
3	C	20	20	20	20×20
4	MP	-	-	20	10×10
5	C	20	20	20	9×9
6	MP	-	-	20	3×3
7	C	20	20	3	2×2
8	MP	-	-	3	1×1

Table 1. MPCNN topology. The table summarizes the MPCNN architecture used; type of a layer, where C is for convolutional, MP for max-pooling, dimensions of the filter bank m and n, number of output maps and their resolution.

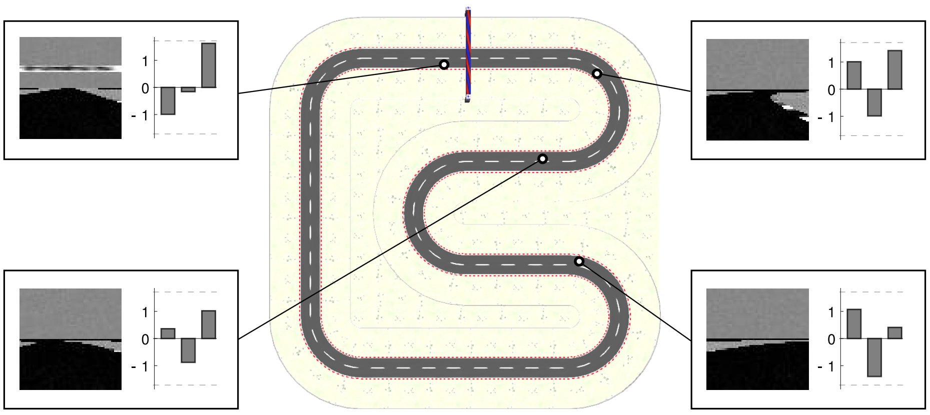

Fig. 4. Evolved visual features. Each inset shows the image at a particular point on the track and the 3-dimensional feature vector produced by the MPCNN after its evolution finished.

At the end of the four generations, the best compressor in the population becomes the compressor for the controllers in the next round of CoSyNE$_c$. CoSyNE$_k$ and CoSyNE$_c$ continue to alternate until a sufficiently fit controller is found, with 20% of the training images being replaced by new ones selected at random in each iteration of CoSyNE$_k$. Both MPCNN compressors and controllers are directly encoded into real-valued genomes, and the population size of both CoSyNEs was 100, with a mutation rate of 0.8.

4.2 Results

We report just 2 runs because a single run of 300 generations of both CoSyNE$_c$ and CoSyNE$_k$ takes almost 80 hours on an 8-core machine[1] (running 8 evaluations in parallel). The fitness, f_c, reached 509.1 at generation 289 for run no.

[1] AMD FX 8120 8-core, 16 GB RAM, nVidia GTX-570

controller	d **[m]**	v_{max} **[km/h]**
olethros	570	147
bt	613	141
berniw	624	149
tita	657	150
inferno	682	150
visual RNN [14]	625	144
MPC-RNN [16]	547	97
Online MPC-RNN	489	91

Table 2. Maximum distance, d, in meters and maximum speed, v_{max}, in kilometers per hour achieved by hand-coded controllers that come with TORCS which enjoy access to the state variables (the five upper table entries), a million-weight RNN controller that drives using pre-processed 64×64 pixel images as input, evolved indirectly in the Fourier domain, MPC-RNN, where the MPCNN is trained offline from manually collected images [16] and the Online MPC-RNN agent that collects the MPCNN training images automatically.

1 and 495.5 at generation 145 for run no. 2, where a generation refers to one generation of each CoSyNE. Table 2 compares the distance travelled and maximum speed of the best controller with the offline-evolved MPCNN controller [16], a large RNN controller evolved in frequency domain [14], and the hand-coded controllers that come with the TORCS package.

The performance of Online MPC-RNN is not as good as its offline variant, but the controllers still approach a fitness of 500, which allows them to complete a lap and continue driving without crashing. The controllers with the pre-trained MPCNN drive slightly better because of possibly two reasons (1) the MPCNN compressor, that improves online does not reach the optimum and (2) as the compressor evolves together with the RNN controller, the weights of the controller have to be updated after each compressor change due to different features that it provides. The hand-coded controllers are much better since they enjoy an access to the car telemetry and features like distances to the track edges.

Figure 3 shows the evolution of the MPCNN feature vectors for each of the 40 images in the training set, in one of the two runs. As the features evolve they very quickly move away from each other in the feature space. While simply pushing the feature vectors apart is no guarantee of achieving maximally informative compressed representations, this simple, unsupervised training procedure provides enough discriminative power in practice to get the car safely across the finish line.

5 Discussion

The results show that it is not necessary to undertake the complicated procedure of collecting the images manually by driving the car and training the compressor beforehand. The MPCNN compressor, trained online from images gathered is good enough for a small RNN with 33 weights to be evolved efficiently to solve the driving control task.

Another approach would be to combine the controllers and compressors within a single directly-encoded genome, but one can expect this to run even longer. Also, instead of using an MPCNN with fitness f_k, the collected images could be used to train autoencoders [11] that would be forced to generated suitable features.

The presented framework is more general than just a TORCS driving system. It remains to be seen whether the agents can be plugged into some other environment, collect the images on the fly and train the controllers to perform the desired task. The TORCS control signal is only 3-dimensional, future experiments will apply Online MPC-RNN to higher-dimensional action spaces like the 41-DOF iCub humanoid, to perform manipulation tasks using vision.

Acknowledgments

This research was supported by Swiss National Science Foundation grant #138219: "Theory and Practice of Reinforcement Learning 2", and EU FP7 project: "NAnoSCaleEngineering for Novel Computation using Evolution" (NASCENCE), grant agreement #317662.

References

1. D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, 2010.
2. D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1237–1242, 2011.
3. G. Cuccu, M. Luciw, J. Schmidhuber, and F. Gomez. Intrinsically motivated evolutionary search for vision-based reinforcement learning. In *Proceedings of the IEEE Conference on Development and Learning, and Epigenetic Robotics*, 2011.
4. D. B. D'Ambrosio and K. O. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th Conference on Genetic and Evolutionary Computation*, (GECCO), pages 974–981, New York, USA, 2007. ACM.
5. F. Fernández and D. Borrajo. Two steps reinforcement learning. *International Journal of Intelligent Systems*, 23(2):213–245, 2008.
6. K. Fukushima. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.
7. J. Gauci and K. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, (GECCO), pages 997–1004. ACM, 2007.

8. L. Gisslén, M. Luciw, V. Graziano, and J. Schmidhuber. Sequential Constant Size Compressors and Reinforcement Learning. In *Proceedings of the Fourth Conference on Artificial General Intelligence*, 2011.
9. F. J. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.
10. F. Gruau. Cellular encoding of genetic neural networks. Technical Report RR-92-21, Ecole Normale Superieure de Lyon, Institut IMAG, Lyon, France, 1992.
11. G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
12. S. R. Jodogne and J. H. Piater. Closed-loop learning of visual control policies. *Journal of Artificial Intelligence Research*, 28:349–391, 2007.
13. H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.
14. J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Amsterdam, 2013.
15. J. Koutník, F. Gomez, and J. Schmidhuber. Evolving neural networks in compressed weight space. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*, 2010.
16. J. Koutník, J. Schmidhuber, and F. Gomez. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 Genetic and Evolutionary Computation Conference (GECCO)*. ACM Press, 2014.
17. S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *International Joint Conference on Neural Networks (IJCNN), Barcelona, Spain*, 2010.
18. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
19. R. Legenstein, N. Wilbert, and L. Wiskott. Reinforcement Learning on Slow Features of High-Dimensional Input Streams. *PLoS Computational Biology*, 6(8), 2010.
20. D. Pierce and B. Kuipers. Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92:169–229, 1997.
21. M. Riedmiller, S. Lange, and A. Voigtlaender. Autonomous reinforcement learning on raw visual input data in a real world application. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Brisbane, Australia, 2012.
22. D. Scherer, A. Müller, and S. Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Proceedings of the International Conference on Artificial Neural Networks*, ICANN, 2010.
23. J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.
24. R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12 (NIPS)*, pages 1057–1063, 1999.
25. G. Tesauro. Practical issues in temporal difference learning. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 4 (NIPS)*, pages 259–266. Morgan Kaufmann, 1992.
26. X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.