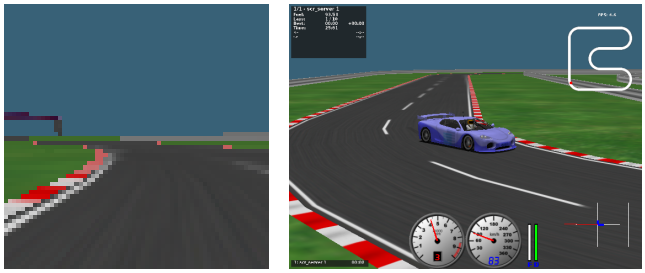
Evolving Large-Scale Neural Networks for Vision-Based TORCS

Jan Koutník Giuseppe Cuccu Jürgen Schmidhuber Faustino Gomez
IDSIA, USI-SUPSI
Galleria 2
Manno-Lugano, CH 6928
{hkou, giuse, juergen, tino}@idsia.ch

ABSTRACT

The TORCS racing simulator has become a standard testbed used in many recent reinforcement learning competitions, where an agent must learn to drive a car around a track using a small set of task-specific features. In this paper, large, recurrent neural networks (with over *1 million* weights) are evolved to solve a much more challenging version of the task that instead uses only a stream of images from the driver's perspective as input. Evolving such large nets is made possible by representing them in the frequency domain as a set of coefficients that are transformed into weight matrices via an inverse Fourier-type transform. To our knowledge this is the first attempt to tackle TORCS using vision, and successfully evolve a neural network controllers of this size.



(a) (b)

Figure 1: Visual TORCS environment. (a) The 1st-person perspective used as input to the RNN controllers (figure 5) to drive the car around the track (figure 6). (b), a 3rd-person perspective of the car.

1. INTRODUCTION

The idea of using evolutionary computation to train artificial neural networks, or *neuroevolution* (NE), has now been around for over 20 years. The main appeal of this approach is that, because it does not rely on gradient information (e.g. backpropagation), it can potentially harness the universal function approximation capability of neural networks to solve reinforcement learning (RL) tasks (i.e. tasks where there is no "teacher" providing targets or examples of correct behavior). Instead of incrementally adjusting the synaptic weights of a single network, the space of network parameters is searched directly according to principles inspired by natural selection: (1) encode a population of networks as strings, or *genomes*, (2) transform them into networks, (3) evaluate them on the task, (4) generate new, hopefully better, nets by recombining those that are most "fit", (5) goto step 2 until a solution is found. By evolving neural networks, NE can cope naturally with tasks that have continuous inputs and outputs, and, by evolving networks with feedback connections (recurrent networks), it can tackle more general tasks that require memory.

Early work in the field focused on evolving rather small networks (hundreds of weights) for RL benchmarks, and control problems with relatively few inputs/outputs. However, as RL tasks become more challenging, the networks required become larger, as do their genomes. The result is that scaling NE to large nets (i.e. tens of thousands of weights) is infeasible using a straightforward, direct encoding where genes map one-to-one to network components. Therefore, recent efforts have focused on *indirect* encodings [1, 4, 5, 13] where relatively small genomes are transformed into networks of arbitrary size using a more complex mapping. This approach offers the potential for evolving very large networks efficiently, by embedding them in a low-dimensional search space.

In previous work [3, 6, 7, 14], we presented a new indirect encoding where network weight matrices are represented as a set of coefficients that are transformed into weight values via an inverse Fourier-type transform, so that evolution-

ary search is conducted in the frequency-domain instead of weight space. The basic idea is that if nearby weights in the matrices are correlated, then this regularity can be encoded using fewer coefficients than weights, effectively reducing the search space dimensionality. For problems exhibiting a high-degree of redundancy, this "compressed" approach can result in an order of magnitude fewer free parameters and significant speedup [7].

With this encoding, networks with over 3000 weights were evolved to successfully control a high-dimensional version of the Octopus Arm task [15], by searching in the space of as few as 20 Fourier coefficients (164:1 compression ratio) [8]. In this paper, the approach is scaled up dramatically to networks with over *1 million weights*, and applied to a new, vision-based version of the TORCS race car driving environment (figure 1). In the standard setup for TORCS—used now for several years in reinforcement learning competitions [9, 10, 11]—a set of features describing the state of the car is provided to the driver. In the version used here, the controllers do not have access to these features, but instead must drive the car using only a stream of images from the driver's perspective; no task-specific information is provided to the controller, and the controllers must compute the car velocity internally, via feedback (recurrent) connections, based on the history of observed images.

To our knowledge this is the first attempt to tackle TORCS using vision, and successfully evolve neural network controllers of this size.

The next section describes the compressed network encoding in detail. Section 3 presents the visual TORCS software architecture. In section 4, we presented our results in the visual TORCS domain, and discuss them in section 5.

2. COMPRESSED NETWORKS

Networks are encoded as a string or *genome*, $\boldsymbol{g} = \{g_1, \ldots, g_k\}$, consisting of k substrings or *chromosomes* of real numbers representing Discrete Cosine Transform (DCT) coefficients. The number of chromosomes is determined by the choice of network architecture, Ψ, and data structures used to decode the genome, specified by $\Omega = \{D_1, \ldots, D_k\}$, where D_m, $m = 1..k$, is the dimensionality of the coefficient array for chromosome m. The total number of coefficients, $C = \sum_{m=1}^{k} |g_m| \ll N$, is user-specified (for a compression ratio of N/C, where N is the number of weights in the network), and the coefficients are distributed evenly over the chromosomes. Which frequencies should be included in the encoding is unknown. The approach taken here restricts the search space to *band-limited* neural networks where the power spectrum of the weight matrices goes to zero above a specified limit frequency, c_ℓ^m, and chromosomes contain all frequencies up to c_ℓ^m, $g_m = (c_0^m, \ldots, c_\ell^m)$.

Each chromosome is mapped to its coefficient array according to Algorithm 1 (figure 2) which takes a list of array dimension sizes, $d = (d_1, \ldots, d_{D_m})$ and the chromosome, g_m,

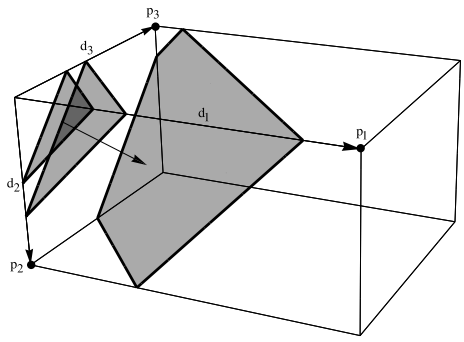


Figure 2: Mapping the coefficients. The cuboidal array (top) is filled with the coefficients from chromosome g according to Algorithm 1, starting at the origin and moving to the opposite corner one simplex at a time.

Algorithm 1: Coefficient mapping(g, d)

$j \leftarrow 0$
$K \leftarrow \text{sort}(\text{diag}(d) - \mathbb{I})$
for $i = 0$ **to** $|d| - 1 + \sum_{n=1}^{|d|} d_n$ **do**
  $l \leftarrow 0$
  $s_i \leftarrow \{e | \sum_{k=1}^{|d|} e_{\xi_j} = i\}$
  while $|s_i| > 0$ **do**
    $ind[j] \leftarrow \underset{e \in s_i}{\text{argmin}} \left\| e - K[l\text{++} \bmod |d|] \right\|$
    $s_i \leftarrow s_i \setminus ind[j\text{++}]$
for $i = 0$ **to** $|ind|$ **do**
  if $i < |g|$ **then**
    coeff_array[$ind[i]$] $\leftarrow c_i$
  else
    coeff_array[$ind[i]$] $\leftarrow 0$

to create a total ordering on the array elements, $e_{\xi_1,\ldots,\xi_{D_m}}$. In the first loop, the array is partitioned into $(D_m - 1)$-simplexes, where each simplex, s_i, contains only those elements e whose Cartesian coordinates, $(\xi_1, \ldots, \xi_{D_m})$, sum to integer i. The elements of simplex s_i are ordered in the `while` loop according to their distance to the corner points, p_i (i.e. those points having exactly one non-zero coordinate; see example points for a 3D-array in figure 2), which form the rows of matrix $K = [p_1, \ldots, p_m]^T$, sorted in descending order by their sole, non-zero dimension size. In each loop iteration, the coordinates of the element with the smallest Euclidean distance to the selected corner is appended to the list *ind*, and removed from s_i. The loop terminates when s_i is empty.

After all of the simplexes have been traversed, the vector *ind* holds the ordered element coordinates. In the final loop, the array is filled with the coefficients from low to high fre-

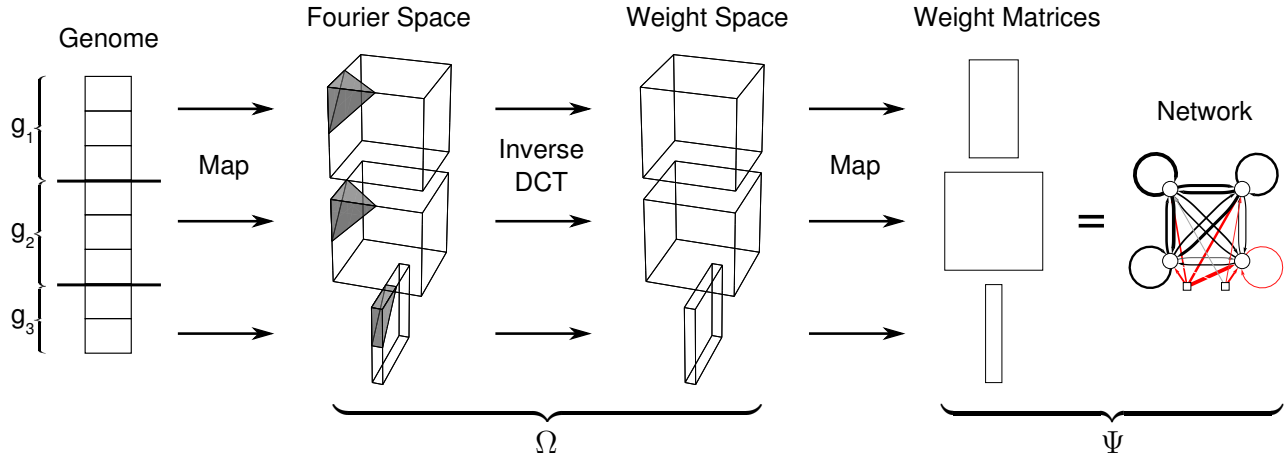


Figure 3: Decoding the compressed networks. The figure shows the three step process involved in transforming a genome of frequency-domain coefficients into a recurrent neural network. First, the genome (left) is divided into k chromosomes, one for each of the weight matrices specified by the network architecture, Ψ. Each chromosome is mapped, by Algorithm 1, into a coefficient array of a dimensionality specified by Ω. In this example, an RNN with two inputs and four neurons is encoded as 8 coefficients. There are $k = |\Omega| = 3$, chromosomes and $\Omega = \{3, 3, 2\}$. The second step is to apply the inverse DCT to each array to generate the weight values, which are mapped into the weight matrices in the last step.

quency to the positions indicated by *ind*; the remaining positions are filled with zeroes. Finally, a D_m-dimensional inverse DCT transform is applied to the array to generate the weight values, which are mapped to their position in the corresponding 2D weight matrix. Once the k chromosomes have been transformed, the network is complete.

Figure 3 shows an example of the decoding procedure for a fully-recurrent neural network (on the right) represented by $k = 3$ weight matrices, one for the input layer weights, one for the recurrent weights, and one for the bias weights. The weights in each matrix are generated from a different chromosome which is mapped into its own D_m-dimensional array with the same number of elements as its corresponding weight matrix; in the case shown, $\Omega = \{3, 3, 2\}$: 3D arrays for both the input and recurrent matrices, and a 2D array for the bias weights.

In [7], the coefficient matrices were 2D, so that the simplexes are just the secondary diagonals; starting in the top-left corner, each diagonal is filled alternately starting from its corners. However, if the task exhibits inherent structure that cannot be captured by low frequencies in a 2D layout, more compression can potentially be gained by organizing the coefficients in higher-dimensional arrays [8].

3. VISUAL TORCS

The visual TORCS environment is based on TORCS version 1.3.1. The simulator had to be modified in order to be usable with vision. Figure 4 describes the software architecture schematically. At each time step during a network evaluation, an image rendered in OpenGL is captured in the car code (C++), and passed via UDP to the client (Java), that contains the RNN controller. The client is wrapped into a Java class that provides methods for setting up the RNN weights, executing the evaluation, and returning the fitness score. These methods are called from Mathematica which is used to decode the compressed networks (figure 3) and the evolutionary search.

The Java wrapper allows multiple controllers to be evaluated in parallel in different instances of the simulator via different UDP ports. This feature is critical for the experiments presented below since, unlike the non-vision-based TORCS, the costly image rendering, required for vision, cannot be disabled. The main drawback of the current implementation is that the images are captured from the screen buffer and, therefore, have to actually be rendered to the screen.

Other tweaks to the original TORCS include changing the control frequency from 50 Hz to 5 Hz, and removing the *3-2-1-GO* waiting sequence from the beginning of each race. The image passed in the UDP is encoded as a message chunk with *image* prefix, followed by *unsigned byte* values of the image pixels. Each image is decomposed into the HSB color space and only the saturation (S) plane is passed in the message.

4. EXPERIMENTS

The goal of the task is to evolve a recurrent neural network controller that can drive the car around a race track using only vision. The challenge for the controller is not only to interpret each static image as it is received, but also to retain information from previous images in order to compute the velocity of the car internally, via its feedback connections.

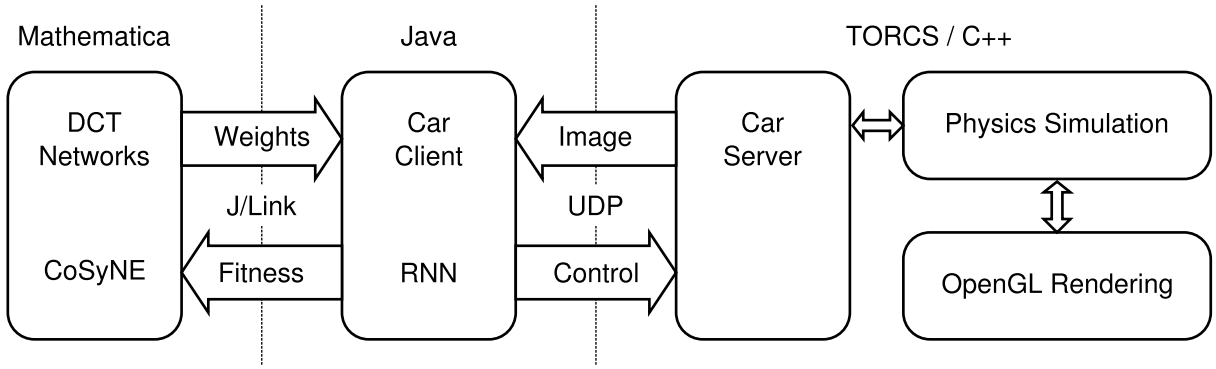


Figure 4: Visual TORCS software platform. The TORCS environment contains a physics simulator connected to the car server that can, in our enhanced version, transmit images from the environment to a client that controls the car using an RNN. The DCT coefficient genomes are evolved and decoded into RNN weights in Mathematica and passed via a J/Link interface to the car client. Fitness is calculated in the client and sent back to Mathematica after each simulation.

4.1 Setup

In each fitness evaluation, the car is placed at the starting line of the track (figure 6), and its mirror image, and a race is run for 25s of a simulated time, resulting in a maximum of 125 time-steps at the 5Hz control frequency. At each control step (figure 5), a raw 64×64 pixel image, taken from the driver's perspective is split into three color planes (hue, saturation and brightness). The saturation plane is passed through Robert's edge detector [12] and then fed into a Elman (recurrent) neural network (SRN) with $16 \times 16 = 256$ fully-interconnected neurons in the hidden layer, and 3 output neurons. The first two outputs, o_1, o_2, are averaged, $(o_1 + o_2)/2$, to provide the steering signal, and the third neuron, o_3 controls the brake and throttle (-1 = full brake, 1 = full throttle). All neurons use sigmoidal activation functions.

With this architecture, the networks have a total of 1,115,139 weights, organized into 5 weight matrices. The weights are encoded indirectly by 200 DCT coefficients which are mapped into 5 coefficient arrays, $\Omega = \{4, 4, 2, 3, 1\}$: (1) a 4D array encodes the input weights from the 2D input image to the 2D array of neurons in the hidden layer, so that each weight is correlated (a) with the weights of adjacent pixels for the same neuron, (b) with the weights for the same pixel for neurons that are adjacent in the 16×16 grid, and (c) with the weights from adjacent pixels connected to adjacent neurons; (2) a 4D array encodes the recurrent weights in the hidden layer, again capturing three types of correlations; (3) a 2D array encodes the hidden layer biases; (4) a 3D array encodes weights between the hidden layer and 3 output neurons; and (5) a 1D array with 3 elements encodes the output neuron biases (see [8] for further discussion of higher-dimensional coefficient matrices).

The coefficients are evolved using Cooperative Synapse NeuroEvolution (CoSyNE; [2]) algorithm with a population size of 64, a mutation rate of 0.8, and fitness being computed by:

$$f = d - \frac{3m}{1000} + \frac{v_{max}}{5} - 100c \ , \tag{1}$$

where d is the distance along the track axis, v_{max} is maximum speed, m is the cumulative damage, and c is the sum of squares of the control signal differences, divided by the number of control variables, 3, and the number simulation control steps, T:

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

The maximum speed component in equation (1) 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 shown in figure 6 and its mirror image to prevent the RNN from blindly memorizing the track without using the visual input (i.e. evolution can find weights which implement a dynamical system that drives the track from the same initial conditions, even with no 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 fitness is the minimum of the two track scores.

4.2 Results

Table 1 compares the distance travelled and maximum speed of the visual RNN controller with that of other, hard-coded controllers that come with the TORCS package. The performance of the vision-based controller is similar to that of the other controllers which enjoy access to the full set of

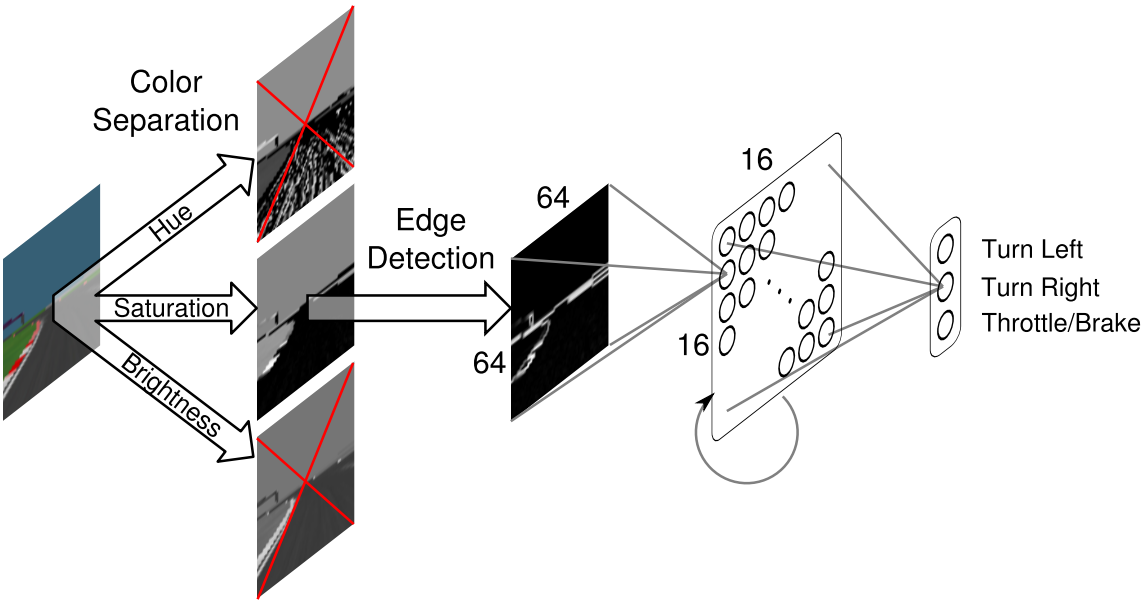


Figure 5: Visual TORCS network controller pipeline. At each time-step a raw 64×64 pixel image, taken from the driver's perspective, is split into three plane (hue, saturation and brightness). The saturation plane is then passed through Robert's edge detector [12] and then fed into the 16×16=256 recurrent neurons of the controller network, which then outputs the three driving commands.

controller	d **[m]**	v_{max} **[km/h]**
olethros	570	147
bt	613	141
berniw	624	149
tita	657	150
inferno	682	150
visual RNN	625	144

Table 1: Maximum distance, d, in meters and maximum speed, v_{max}, in kilometers per hour achieved by the selected hard-coded controllers compared to the visual RNN controller.

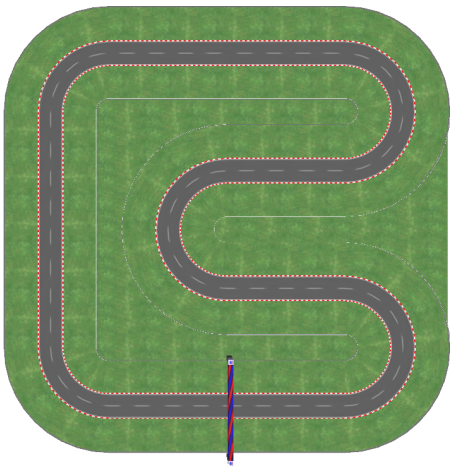


Figure 6: Training race track. The controllers were evolved using a track of length of 714.16m and width of 10m, that consists of straight segments of length 50 and 100m and curves with a radius of 25m. The car starts at the bottom (start line) and has to drive counter-clockwise. The track boundary is 14m wide

pre-processed TORCS features, such as forward and lateral speed, angle to the track axis, position on the track, distance to the track side, etc.

Figure 7 compares the learning curve for the compressed networks (upper curve), and a typical evolutionary run (lower curve) where the network is evolved *directly* in weight space, i.e. using chromosomes with 1,115,139 genes, one for each weight, instead of 200 coefficient genes. Direct evolution makes little progress as each of the weights has to be set individually, without being explicitly constrained by the values of other weights in their matrix neighborhood, as is the case for the compressed encoding.

As discussed above, the controllers were evaluated on two tracks to prevent them from simply "memorizing" a single

sequence of curves. In the initial stages of evolution, a sub-optimal strategy is to just drive straight on both tracks ignoring the first curve, and crashing into the barrier. This is a simple behavior, requiring no vision that produces relatively high fitness, and therefore represents local minima in the fitness landscape. This can be seen in the flat portion of the curve until generation 118, when the fitness jumps from 140 to 190, as the controller learns to turn both left and right. Gradually, the controllers start to distinguish between the two tracks as they develop useful visual feature detectors. From then on the evolutionary search refines the control to optimize acceleration and braking through the curves and straight sections.

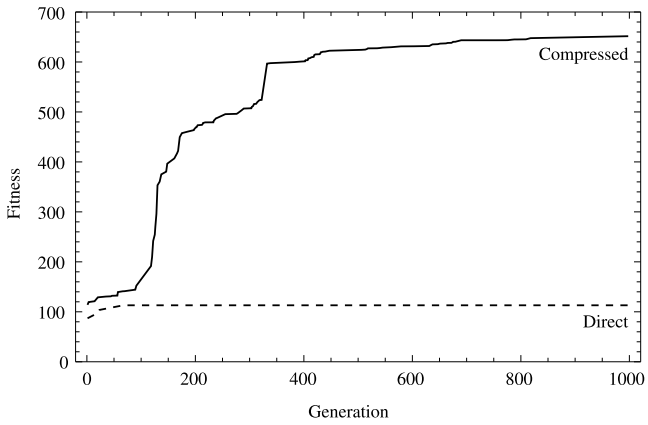


Figure 7: Learning curve. Typical fitness evolution of a compressed (upper curve) and directly encoded (lower curve) controller during 1000 generations. The compressed controller escapes from the local minima at generation 118, but the directly encoded network never learns to distinguish between left and right curve from the visual features.

5. DISCUSSION

While preliminary, the results presented above show that it is possible to evolve neural controllers on an unprecedented scale. The compressed network encoding reduces the search space dimensionality by exploiting the inherent regularity in the environment. Since, as with most natural images, the pixels in a given neighborhood tend to have correlated values, searching for each weight independently is overkill. Using fewer coefficients than weights sacrifices some expressive power (some networks can no longer be represented), but constrains the search to the subspace of lower complexity—but still sufficiently powerful—networks, thereby reducing the search space dimensionality by, e.g. a factor of more than 5000 for the driver networks evolved here.

Figure 8 shows the five weight matrices of a typical RNN controller evolved for the visual TORCS task. Had this network been evolved in weight space, the matrices would look like noise, with no apparent structure. Here, however, the constraints imposed by using a Fourier basis of just 200 coefficients means that the matrices exhibit clear regularity.

The (16×64) blocking seen in the input and recurrent arrays is due to the use of 4D coefficient arrays. So, for example, take the upper-left block in the figure inset, which is also the upper-left block of the entire input matrix. This block represents the weight values of the first row of 16 neurons for the first row of 64 pixels in the input image. The next block to the right corresponds to the weights for the second row in the image for the same set of neurons, which, given the similarity between any two adjacent rows in the image, are only slightly different. As we move left-to-right in the matrix (i.e. moving through the input image from top-to-bottom), the weights change smoothly; as they do when we move along a column of blocks in the matrix: each group of 16 neurons selects slightly different features from a given row in the input image.

Further experiments are needed to compare the approach with other indirect or generative encodings such as HyperNEAT [1]; not only to evaluate the relative efficiency of each algorithm, but also to understand how the methods differ in the type of solutions they produce. Part of that comparison should involve testing the controllers in different conditions from those under which they were evolved (e.g. on different tracks) to measure the degree to which the ability to generalize benefits from the low-complexity representation, as was shown in [8].

The compressed network encoding used here assumes band-limited networks, where the matrices can contain all frequencies up to a predefined limit frequency. For networks with as many weights as those used for visual TORCS, this may not be the best scheme as the limit frequency has to be chosen by the user, and if some specific high frequency is needed to solve the task, then all lower frequencies must be searched as well. A potentially more tractable approach might be Generalized Compressed Network Search (GNCS; [14]) which uses a messy GA to simultaneously determine which arbitrary subset of frequencies should be used as well as the value at each of those frequencies. Our initial work with this method has been promising.

Acknowledgements

This research was supported by Swiss National Science Foundation grants #137736: "Advanced Cooperative NeuroEvolution for Autonomous Control" and #138219: "Theory and Practice of Reinforcement Learning 2".

References

[1] 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 annual conference on Genetic and evolutionary computation*, (GECCO), pages 974–981, New York, NY, USA, 2007. ACM.

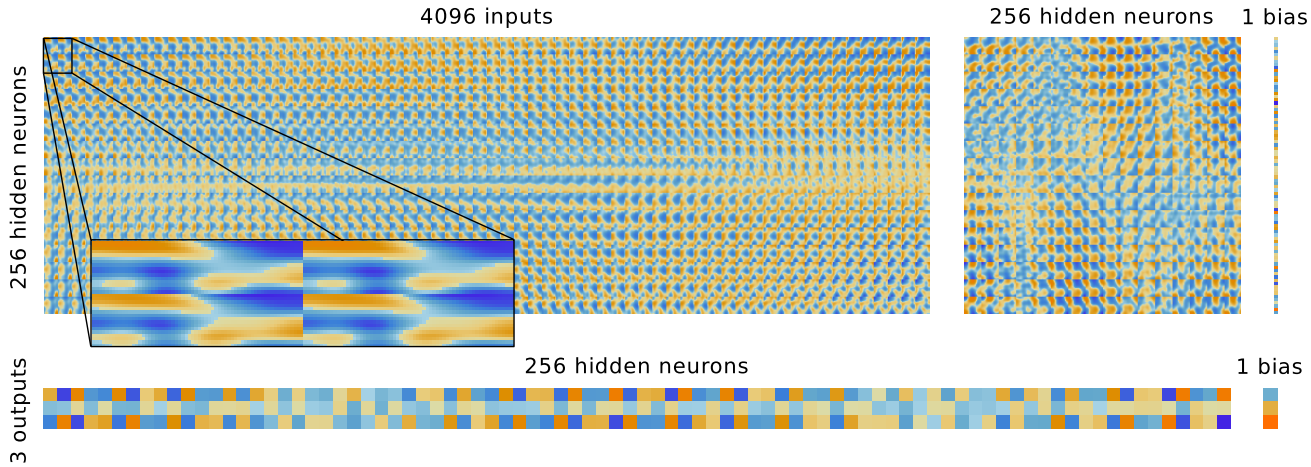


Figure 8: Evolved low-complexity weight matrices. Colors indicate weight value: orange = large positive; blue = large negative. The frequency-domain representation enforces clear regularity on weight matrices that reflects the regularity of the environment.

[2] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

[3] F. Gomez, J. Koutník, and J. Schmidhuber. Compressed network complexity search. In *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN XII, Taormina, IT)*, 2012.

[4] F. Gruau. Cellular encoding of genetic neural networks. Technical Report RR-92-21, Ecole Normale Superieure de Lyon, Institut IMAG, Lyon, France, 1992.

[5] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[6] J. Koutník, F. Gomez, and J. Schmidhuber. Searching for minimal neural networks in Fourier space. In *Proceedings of the 4th Annual Conference on Artificial General Intelligence*, 2010.

[7] 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.

[8] J. Koutník, J. Schmidhuber, and F. Gomez. A frequency-domain encoding for neuroevolution. Technical report, arXiv:1212.6521, 2012.

[9] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, and Y. Saez. The WCCI 2008 simulated car racing competition. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 119–126. IEEE, 2008.

[10] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lönneker, L. Cardamone, D. Perez, Y. Sáez, M. Preuss, and J. Quadflieg. The 2009 simulated car racing championship, 2009.

[11] D. Loiacono, L. Cardamone, and P. L. Lanzi. Simulated car racing championship competition software manual. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 2011.

[12] L. G. Roberts. *Machine Perception of Three-Dimensional Solids*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1963. ISBN 0-8240-4427-4.

[13] J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

[14] R. K. Srivastava, J. Schmidhuber, and F. Gomez. Generalized compressed network search. In *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature (PPSN XII, Taormina, IT)*, 2012.

[15] Y. Yekutieli, R. Sagiv-Zohar, R. Aharonov, Y. Engel, B. Hochner, and T. Flash. A dynamic model of the octopus arm. I. biomechanics of the octopus reaching movement. *Journal of Neurophysiology*, 94(2):1443–1458, 2005.