Playing Atari with Six Neurons (Extended Abstract*)

Giuseppe Cuccu[1] , **Julian Togelius**[2] and **Philippe Cudré-Mauroux**[1]
[1]eXascale Infolab, Department of Computer Science, University of Fribourg, Switzerland
[2]Game Innovation Lab, Tandon School of Engineering, New York University, NY, USA
giuseppe.cuccu@unifr.ch, julian@togelius.com, philippe.cudre-mauroux@unifr.ch

Abstract

Deep reinforcement learning applied to vision-based problems like Atari games maps pixels directly to actions; internally, the deep neural network bears the responsibility of both extracting useful information and making decisions based on it. By separating image processing from decision-making, one could better understand the complexity of each task, as well as potentially find smaller policy representations that are easier for humans to understand and may generalize better. To this end, we propose a new method for learning policies and compact state representations separately but simultaneously for policy approximation in reinforcement learning. State representations are generated by an encoder based on two novel algorithms: Increasing Dictionary Vector Quantization makes the encoder capable of growing its dictionary size over time, to address new observations, and Direct Residuals Sparse Coding encodes observations by aiming for highest information inclusion. We test our system on a selection of Atari games using tiny neural networks of only 6 to 18 neurons (depending on the game's controls). These are still capable of achieving results comparable—and occasionally superior—to state-of-the-art techniques which use two orders of magnitude more neurons.

1 Introduction

In deep reinforcement learning, a large network learns to map complex, high dimensional input to actions for direct policy approximation. When a giant network with hundreds of thousands of parameters learns a relatively simple task (such as playing Qbert) it stands to reason that only a small part of what is learned is the actual policy.

Separating the representation learning from policy learning allows in principle for higher component specialization, enabling smaller networks dedicated to policy learning to address problems typically tackled by much larger networks. Separating the policy network from image parsing also allows us to better understand how network complexity contributes to accurately representing the policy. Yet another reason to investigate how to learn smaller policy networks by addressing image processing with a separate component is that smaller networks may offer better generalization.

The key contribution of this paper is a new method for learning policy and features *simultaneously* but *separately* in a complex reinforcement learning setting. This is achieved through two novel algorithms: Increasing Dictionary Vector Quantization (IDVQ) and Direct Residuals Sparse Coding (DRSC).

IDVQ maintains a dictionary of centroids in the observation space, which can then be used for encoding. The two main differences with standard VQ are that the centroids are (i) trained online by (ii) disregarding reconstruction error. The dictionary trained by IDVQ is then used by DRSC to produce a compact code for each observation. This code is used in turn by the neural network (policy) as input to select the next action. The code is a binary string: a value of '1' indicates that the corresponding centroid contains information also present in the image, and a limited number of centroids are used to represent the totality of the information.

As the training progresses and more sophisticated policies are learned, complex interactions with the environment result in increasingly novel observations; the dictionary reflects this by *growing in size*, including centroids that account for newly discovered features.

With the goal of minimizing the network size while maintaining comparable scores, experimental results show that this approach can effectively learn both components simultaneously, achieving state-of-the-art performance on several ALE games while using a neural network of *only 6 to 18 neurons*, i.e. **two orders of magnitude smaller** than any known previous implementation. This extended abstract is based on the original AAMAS 2019 publication [Cuccu *et al.*, 2019].

2 Related Work

Video games are frequently used for AI research [Yannakakis and Togelius, 2018]. The Arcade Learning Environment (ALE), based on an emulation of the Atari 2600, is a very popular benchmark set [Bellemare *et al.*, 2013]. In the most common setup, the raw pixel output of the ALE framework is used as inputs to a neural network, and the outputs are interpreted as commands for playing the game. Using this setup,

*Original work published at AAMAS 2019 [Cuccu *et al.*, 2019].

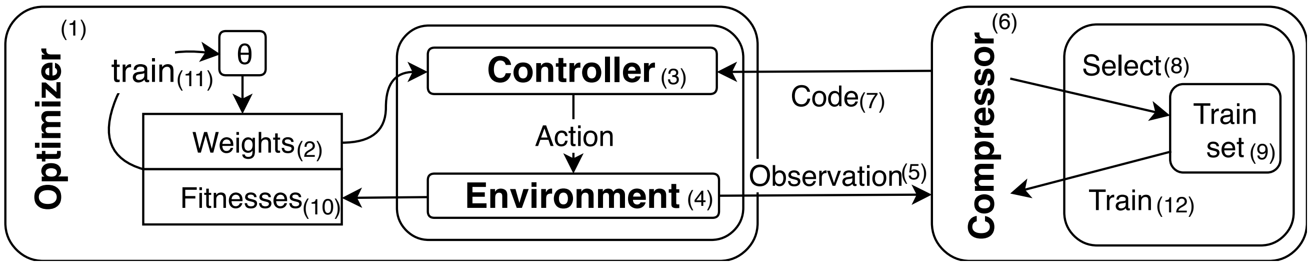

Figure 1: System diagram. At each generation the optimizer (1) generates sets of weights (2) for the neural network controller (3). Each network is evaluated episodically against the environment (4). At each step the environment sends an observation (5) to an external compressor (6), which produces a compact encoding (7). The network uses that encoding as input. Independently, the compressor selects observations (8) for its training set (9). At the end of the episode, the environment returns the fitness (cumulative reward; 10) to the optimizer for training (neuroevolution; 11). Compressor training (12) takes place in between generations.

Mnih et al. reached above human level results on a majority of 57 Atari games that come with the standard distribution of ALE [Mnih *et al.*, 2015]. Since then, a number of improvements have been suggested that have improved game-playing strength on most of these games [Hessel *et al.*, 2017; Justesen *et al.*, 2019].

Neuroevolution uses evolutionary algorithms to train neural networks [Floreano *et al.*, 2008; Yao, 1999; Igel, 2003; Risi and Togelius, 2017]. Typically, this means training the connection weights of a fixed-topology neural network, though some algorithms evolve the topology as well [Stanley and Miikkulainen, 2002]. Until recently, reinforcement learning generally relied on low-dimensional features, either by using intrinsically low-dimensional sensors (such as infrared or laser range-finders) or by using hard-coded computer vision techniques. Such hard mappings generalize badly; in order to create a more general reinforcement learning method, the mapping must be automatically constructed or learned.

3 Method

Our system is divided into four main components: i) the Environment is an Atari game, taking actions and providing observations; ii) the Compressor extracts a low-dimensional code from the observation, while being trained online with the rest of the system; iii) the Controller is our policy approximator, i.e. the neural network; finally iv) the Optimizer is our learning algorithm, improving the performance of the network over time, in our case an Evolution Strategy. Each component is described in more detail below.

3.1 Environment

We test our method on the Arcade Learning Environment (ALE), interfaced through the OpenAI Gym framework [Brockman *et al.*, 2016]. As discussed above, ALE is built on top of an emulator of the Atari 2600, with all the limitations of that console. In keeping with ALE conventions, the observation consists of a $[210 \times 180 \times 3]$ tensor, representing the RGB pixels of the screen input.

3.2 Compressor

The role of the compressor is to provide a compact representation for each observation coming from the environment, enabling the neural network to entirely focus on decision making. This is done through unsupervised learning on the very same observations that are obtained by the network interacting with the environment, in an *online learning* fashion.

We propose a new algorithm based on Vector Quantization (VQ) named Increasing Dictionary VQ (IDVQ), coupled with a new Sparse Coding (SC) method named Direct Residuals SC (DRSC). The following section gives a step-by-step guide on the application of IDVQ+DRSC. The full derivations and pseudocode algorithms are available in the main publication [Cuccu *et al.*, 2019].

Step-by-Step Breakdown of IDVQ and DRSC

IDVQ is used to train a dictionary, which can then be used by *DRSC* to encode (i.e. extract features from) an observation (image). To understand how these two algorithms work together, let us hypothesize a working starting dictionary and see how DRSC produces an encoding.

Initialization. Two components need to b initialized: the *code*, as an arrays of zeros of the same size as the dictionary, and the *residual information* which still needs encoding, initially the whole original image.

Next centroid. The algorithm then loops to select centroids to add to the encoding, based on how much of the residual information they can encode. To select the most similar centroid, the algorithm computes the differences between the residual information and each centroid in the dictionary, aggregating each of these differences by summing all values. The centroid with the smallest aggregated difference is thereby the most similar to the residual information, and is chosen to be included in the encoding.

Encoding. The corresponding bit in the binary code is flipped to '1', and the residual information is updated by subtracting the new centroid.

Interpreting residual information. The signs of the values in the updated residual information (old residual minus new centroid, the order matters) are now significant: (i) values equal to zero mean a perfect correspondence between the pixel information in the old residual and the corresponding value in the new centroid; (ii) positive values correspond to information that was present in the old residual but not covered by the new centroid; (iii) negative values correspond to information present in the new centroid, but absent (or of

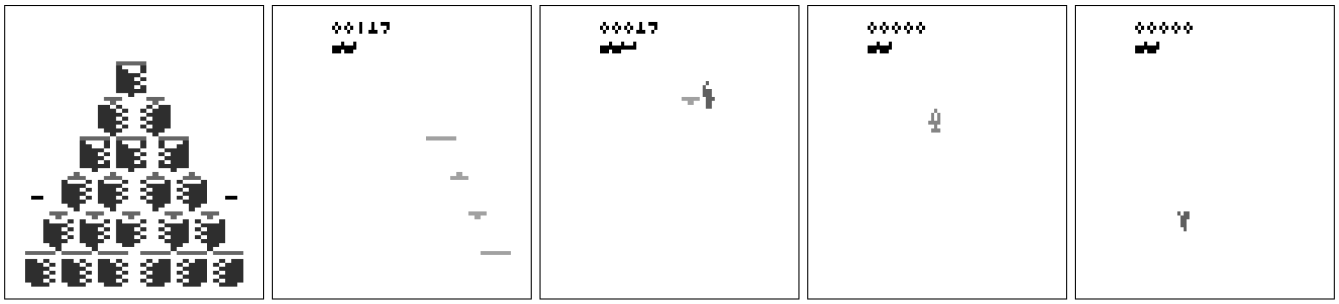

Figure 2: Trained centroids. A few centroids trained with IDVQ during a run of the game Qbert. Notice how the first captures the initial state of the game (background), while the others build features as subsequent residuals: lit cubes, avatar and enemy. Colors are inverted for printing purposes.

smaller magnitude) in the old residual. This is crucial towards the goal of fully representing the totality of the original information, and to this end the algorithm is free to disregard *reconstruction artifacts* as found in (iii).

Loop. The algorithm then keeps looping and adding centroids until the (aggregated) residual information is lower than a threshold, corresponding to an arbitrary precision in capturing the information in the original image.

Dictionary training. The dictionary is trained with IDVQ by adding new centroids to minimize leftover residual information in the encoding. The training begins by selecting an image from the training set and encoding it with DRSC, producing the binary code as described above. A dot product between the code and the dictionary (i.e. summing the centroids selected by the code, since it is binary) produces a reconstruction of the original image, similarly to other dictionary-based algorithms.

Residual information. The difference between the training image and the reconstruction then produces a reconstruction error (-image), where the sign of the values once again correspond to their origin: positive values are leftover information from the image which is not encoded in the reconstruction, while negative values are reconstruction artifacts with no relation to the original image. This reconstruction error image is then aggregated (with a sum) to estimate the quantity of information missed by the encoding.

Adding new centroids. If it is above a given threshold, a new centroid should be added to the dictionary to enable DRSC to make a more precise reconstruction. But in that case the residual itself makes for the perfect centroid, as it exactly captures the information missed by the current encoding, and is then added to the dictionary.

3.3 Controller

The controller for all experiments is a single-layer fully-connected recurrent neural network (RNN). The number of inputs is equal at any given point in time to the size of the code coming from the *compressor*. As the compressor's dictionary grows in size, so does the network's input. In order to ensure continuity in training (i.e. the change needs to be transparent to the training algorithm), it is necessary to define an invariance across this change, where the network with expanded weights is equivalent to the previous one. This is done by setting the weights of all new connections to zero, making the new network mathematically equivalent to the previous one, as any input on the new connections cancels out.

Game	HyperNeat	OpenAI	**Ours**	# n
DemonAttack	3590	1166.5	325	6
FishingDerby	-49	-49	-10	18
Frostbite	2260	370	300	18
Kangaroo	800	11200	1200	18
NameThisGame	6742	4503	920	6
Phoenix	1762	4041	4600	8
Qbert	695	147.5	1250	6
Seaquest	716	1390	320	18
SpaceInvaders	1251	678.5	830	6
TimePilot	7340	4970	4600	10

Table 1: Game scores. Scores on a sample of Atari games (sorted alphabetically), compared to results from HyperNeat and OpenAI ES. Column '*# n*' indicates how many neurons were used in our work, in a single layer (output), for each game. The number of neurons corresponds to the number of available actions in each game, i.e. no neurons are added for performance purpose.

The number of neurons in the output layer is kept equal to the dimensionality of the action space for each game, as defined by the ALE simulator. No hidden layers nor extra neurons were used in any of the presented results.

3.4 Optimizer

The optimizer used in the experiments is a variation of Exponential Natural Evolution Strategy(XNES; [Glasmachers *et al.*, 2010]) tailored for evolving networks with dynamic size.

Since the parameters are interpreted as network weights in *direct encoding* neuroevolution, changes in the network structure need to be reflected by the optimizer in order for future samples to include the new weights.

In order to respect the network's invariance, the expected value of the distribution (μ) for the new dimension should be zero. As for Σ, we need values for the new rows and columns in correspondence to the new dimensions.

Example. Take for example a one-neuron feed-forward network with 2 inputs plus bias, totaling 3 weights. Let us select a function mapping the optimizer's parameters to the weights in the network structure (i.e. the *genotype to phenotype* func-

	HN	Others	**Ours**
# neurons	˜3034	˜650	**˜18**
# hidden layers	2	3	**0**
# connections	˜906k	˜436k	**˜3k**

Table 2: Results. Our proposed approach achieves comparable scores (sometimes better) using up to *two orders of magnitude* fewer neurons, and no hidden layers. The following numbers refer to networks for games with the largest action set (18). Column *HN* is for HyperNeat, column *Others* is for OpenAI ES, GA (1B), and NSRA-ES. Column *Ours* presents the proposed method of IDVQ, DRSC and XNES.

tion), as to first fill the values of all input connections, then all bias connections. Extending the input size to 4 requires the optimizer to consider two more weights before filling in the bias:

$$\mu = \begin{bmatrix}\mu_1 & \mu_2 & \mu_b\end{bmatrix} \quad \rightarrow \quad \begin{bmatrix}\mu_1 & \mu_2 & 0 & 0 & \mu_b\end{bmatrix}$$

$$\Sigma = \begin{bmatrix}\sigma_1^2 & c_{12} & c_{1b} \\ c_{21} & \sigma_2^2 & c_{2b} \\ c_{b1} & c_{b2} & \sigma_b^2\end{bmatrix} \quad \rightarrow \quad \begin{bmatrix}\sigma_1^2 & c_{12} & 0 & 0 & c_{1b} \\ c_{21} & \sigma_2^2 & 0 & 0 & c_{2b} \\ 0 & 0 & \epsilon & 0 & 0 \\ 0 & 0 & 0 & \epsilon & 0 \\ c_{b1} & c_{b2} & 0 & 0 & \sigma_b^2\end{bmatrix}$$

with c_{ij} being the covariance between parameters i and j, σ_k^2 the variance on parameter k, and ϵ being arbitrarily small (0.0001 here). The evolution can pick up from this point on as if simply resuming, and learn how the new parameters influence the fitness.

4 Experimental Setup

The following covers the most important choices in our experimental setup. Please refer to the main paper for a detailed description [Cuccu *et al.*, 2019].

- Population size and learning rates are dynamically adjusted based on the number of parameters, based on the XNES minimal population size and default learning rate [Glasmachers *et al.*, 2010].
- The average dictionary size by the end of the run is around 30-50 centroids, but games with many small moving parts tend to grow over 100. In such games there seems to be direct correlation between higher dictionary size and performance, but our reference machine performed poorly over 150 centroids.
- Every individual is evaluated on 5 environment initializations to reduce fitness variance.
- Experiments are allotted a mere 100 generations, which averages to 2 to 3 hours of run time on our reference machine.

These restrictions are **extremely tight** compared to what is typically used in studies utilizing the ALE framework. Limited experimentation indicates that relaxing any of them, i.e. by accessing the kind of hardware usually dedicated to modern deep learning, consistently improves the results on the presented games. The full implementation is available on GitHub under MIT license[1].

[1]https://github.com/giuse/DNE/tree/six_neurons

5 Results

The goal of this work is not to propose a new generic feature extractor for Atari games, nor a novel approach to beat the best scores from the literature. Our declared goal is to show that **dividing feature extraction from decision making enables tackling hard problems with less resources**, and that the deep networks typically dedicated to this task can be substituted for simple encoders and tiny networks while maintaining comparable performance. Table **??** emphasizes our findings in this regard. Under these assumptions, Table **??** presents comparative results over a set of 10 Atari games from the hundreds available on the ALE simulator.

The resulting scores are compared with recent papers that offer a broad set of results across Atari games on comparable settings, namely HyperNeat [Hausknecht *et al.*, 2014], OpenAI ES [Salimans *et al.*, 2017], GA (1B) [Such *et al.*, 2017], and NSRA-ES [Conti *et al.*, 2018], showing comparable results (and sometimes better). Notably, our setup achieves high scores on *Qbert*, arguably one of the harder games for its requirement of strategic planning.

The real results of the paper are however highlighted in Table **??**, which compares the number of neurons, hidden layers and total connections utilized by each approach. Our setup uses up to **two order of magnitude fewer neurons**, two orders of magnitude fewer connections, and is the only one using only one layer (no hidden layer, equivalent to a simple linear controller).

6 Conclusions

We presented a method to address complex learning tasks, such as learning to play Atari games, by decoupling policy learning from feature construction, learning them independently but simultaneously to further specializes each role. Features are extracted from raw pixel observations coming from the game using a novel and efficient sparse coding algorithm named Direct Residual Sparse Coding. The resulting compact code is based on a dictionary trained online with yet another new algorithm called Increasing Dictionary Vector Quantization, which uses the observations obtained by the networks' interactions with the environment as policy search progresses. Finally, tiny neural networks are evolved to decide actions based on the encoded observations, achieving results comparable with the deep neural networks typically used for these problems while being *two orders of magnitude* smaller.

The implication is that feature extraction on some Atari games is not as complex as often considered. On top of that, the neural network trained for policy approximation is also very small in size, showing that the decision making itself can be done by relatively simple functions. One goal of this paper is to clear the way for new approaches to learning, and to call into question a certain orthodoxy in deep reinforcement learning, namely that image processing and policy should be learned together *(end-to-end)*.

Acknowledgments

This work received support from the Swiss NSF under grant number 407540_167320.

References

[Bellemare *et al.*, 2013] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[Brockman *et al.*, 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[Conti *et al.*, 2018] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5032–5043, 2018.

[Cuccu *et al.*, 2019] Giuseppe Cuccu, Julian Togelius, and Philippe Cudré-Mauroux. Playing Atari with six neurons. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 998–1006. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[Floreano *et al.*, 2008] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[Glasmachers *et al.*, 2010] Tobias Glasmachers, Tom Schaul, Sun Yi, Daan Wierstra, and Jürgen Schmidhuber. Exponential natural evolution strategies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 393–400. ACM, 2010.

[Hausknecht *et al.*, 2014] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[Hessel *et al.*, 2017] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[Igel, 2003] Christian Igel. Neuroevolution for reinforcement learning using evolution strategies. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 4, pages 2588–2595. IEEE, 2003.

[Justesen *et al.*, 2019] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *IEEE Transactions on Games*, 2019.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[Risi and Togelius, 2017] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2017.

[Salimans *et al.*, 2017] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[Stanley and Miikkulainen, 2002] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[Such *et al.*, 2017] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[Yannakakis and Togelius, 2018] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. http://gameaibook.org.

[Yao, 1999] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.