Reinforcement Learning using Reservoir Computing with Optical Co-Processors

Deep Learning Project Report

MVA

January 24, 2021

Authors :
Yan Monier
Raphaël Lafargue

Last update
January 24, 2021

Abstract

Thanks to its recent breakthroughs, Reinforcement Learning (RL) has become a major field of Machine Learning. In RL, an agent learns to interact in its environment. The environment we chose is an OpenAI game called CarRacing-v0. In our case, the task consists in extracting features from the displayed image of a game and take actions to control a car in the best way possible. In this game, the car is driven on randomly generated roads and has to finish the track. We use Convolutional Neural Networks (CNN) to extract "instantaneous features" and a recently developed method of RCRC (Reinforcement Learning with Convolutional Reservoir Computing). This method includes a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) for time-dependent feature. CNN and RNN can be demanding to train, we therefore use random and fixed ones. The last layer is based on the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). It consists in generating matrices and selecting them with a form of "genetic algorithm". It is the only trained layer, the others (CNN +RNN) are random and fixed. We could reproduce some of the results from the original paper on RCRC in which the authors implemented this methodology. Moreover, we indented to use Optical Processing Units (OPUs) to increase the dimensionality of feature space while having a faster and energy efficient architecture. This device can perform very high-dimensional random (but consistent) nonlinear projections. It is therefore particularly suited to be implemented in our program.

Acknowledgement

We would like to thank LightOn for their friendly support. We would like to thank especially Ruben Ohana, PhD. Student at LightOn and INRIA, for his helpful support and advice during this project. He and Iacopo Poli, lead machine learning engineer at LightOn, suggested us some articles to choose for a good project. LightOn also gave us access for free to their Co-Processing Optical devices.

Contents

1 Introduction & Motivation **2**
- 1.1 About us . . . 2
- 1.2 CarRacing-v0 . . . 2
  - 1.2.1 General Overview . . . 2
  - 1.2.2 The reward system . . . 3
- 1.3 What are we trying to demonstrate ? Why is it relevant ? Applications . . . 3

2 Related Work **4**

3 Methodology **4**
- 3.1 Reservoir Computing . . . 5
- 3.2 Optical Processing Units (OPUs) . . . 6
  - 3.2.1 Binarization of the input . . . 7
  - 3.2.2 Optical Reservoir Computing recursive equation . . . 7

4 Challenge encountered and choices made during the programming of the project **7**
4.1 Problem of memory leak 7
4.2 Problem of device calculation 8
4.3 Choice of parameters 8

5 Implementation **8**
5.1 main.py 8
5.2 game.py 9
5.3 Network.py 9

6 Description of behaviors and results **9**

7 Conclusion **10**

8 References **11**

1 Introduction & Motivation

1.1 About us

Both of us were really amazed by what Reinforcement Learning enabled. As a future intern at LightOn (Raphaël), I thought it would be of interest to work with them already so I decided to ask them for suggestions. When we discovered this article on RCRC [1], we were thrilled and enthusiastic to combine it with the powerful features of Optical Processing Units (OPUs). We also thought it was a nice first approach to Reinforcement Learning.

1.2 CarRacing-v0

1.2.1 General Overview

CarRacing-v0 is an OpenSource game designed by OpenAI as a benchmark test for Reinforcement Learning methods. The Figure 1 shows how the player interface looks like. At each frame or timestep a three dimensional vector named "action" is used. "action" will be denoted $\vec{a} \in [-1, 1]^3$. The first dimension is the steering wheel, then the accelerator then the brake.

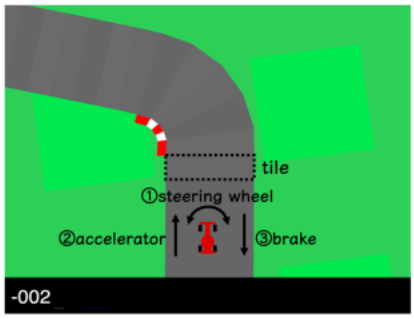

Figure 1: CarRacing-v0 player interface. 1, 2 and 3 are the different degrees of liberty the player has.

Listing 1: Code in the CarRacing class

```
...
    def step(self, action):
        if action is not None:
            self.car.steer(-action[0])
            self.car.gas(action[1])
            self.car.brake(action[2])
...
```

The Road is randomly generated and therefore takes various turns. Consequently, it is not possible for a player to use any other strategy than just watching the screen and adapting the driving behavior accordingly.

1.2.2 The reward system

In order to give a mark to a simulation and differentiate it with other, a reward function was implemented based on how well the car followed the road. The road is divided into little sections that have for value of 10 points: each time the car passes above one of this road section, the total reward is increased by 10. At each step of simulation, the total reward is decreased by 0.1 points in order to lower the score if the car is standing still or not following the road during simulation.

1.3 What are we trying to demonstrate ? Why is it relevant ? Applications

Firstly, we were interested in reproducing the impressive results shown in [1]. We were also trying to show that the use of Optical hardware would considerably lower the cost of computation. This is perfectly in line with the choice of Random Matrices both for the CNN and the RNN part. Reinforcement Learning has many application especially in Robotics and Self-Driving cars. The electric consumption of these systems might drastically increase with advanced AI algorithms running on it. These hurdles could be overpassed using partly untrained architectures and novel hardware designed for high-dimensional data handling.

2 Related Work

Reinforcement learning is an active field of research in artificial intelligence due to its application in a huge number of domains (game theory, simulation-based optimization, multi agent system, swarm intelligence, etc...). In a more concrete way, Reinforcement learning is well adapted for the development of automated systems such as autonomous vehicles, robot manipulation and video games [2]. The use of CNN to extract features out of an image is very common. One could argue that it was initiated by AlexNet [3]. The use of untrained CNN came later with the surprising use of Reservoir Computing for image recognition. [4]. Reservoir Computing (RC) is a simple and efficient ML used for time-dependent information processing. RC was introduced in the 2000s as the fusion of Echo State Networks (ESNs) [5] in ML and Liquid State Machines (LSMs) [6] in Computational Neurosciences. RC is based on a Recurrent Neural Network (RNN) architecture (nodes of the same layer are connected). RNNs' training is known for being challenging [7]. However, RC does not require training over all the network parameters while keeping excellent performance. Only the last layer of weights is trained with a simple linear regression thus making it far more computationally affordable. It also results in the property that the training problem is always convex. Furthermore, its overall minimal construction rules are well suited for hardware implementations. RC has already been applied on low-dimensional systems such as Rössler or Lorenz oscillators (3D) to predict chaotic time series or infer other variables from a single time series [8]. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is now very commonly used for non-convex optimization problems. It is the only trained part of our project and acts as the last layer of our reservoir computer.

The roots of step size choice were discovered in 1968 [9]. In an evolution strategy, new candidate solutions are sampled according to a multivariate normal distribution. The "best" candidates yield the necessmore efficient exploration of the high-dimensional parameter space because local minima can be overpassed. The successful World Model [10] in RL inspired this framework in the sense that feature extraction and action decision model are separated on purpose. In [1], it is claimed that "The separation of these two models results in the stabilization of feature extraction and reduction of parameters to be trained based on task-rewards." Only the Action Decision (CMA-ES) [11][12] model is trained. Every other weights and biases are fixed and random.

3 Methodology

1. **Extracting features from each frame.** This step is solved by untrained random convolutional networks. As detailed in figure 2, the RGB image passes through 3 convolutional layers filled with Normally distributed weights. After Conv3. 128 4x4 pixels images are obtained thus forming a 512 dimensional feature vector.

2. **Extracting time-dependant features from the features obtained in (1.)** The Reservoir Computing (RC) is a method based on Recurrent Neural Networks specialized for time dependent tasks. Its main interest is the low computational cost. Indeed, all the weights and biases (input to "reservoir state" and recurrent connections within the "reservoir") are random and fixed. Only the output layer is usualy trained with a ridge

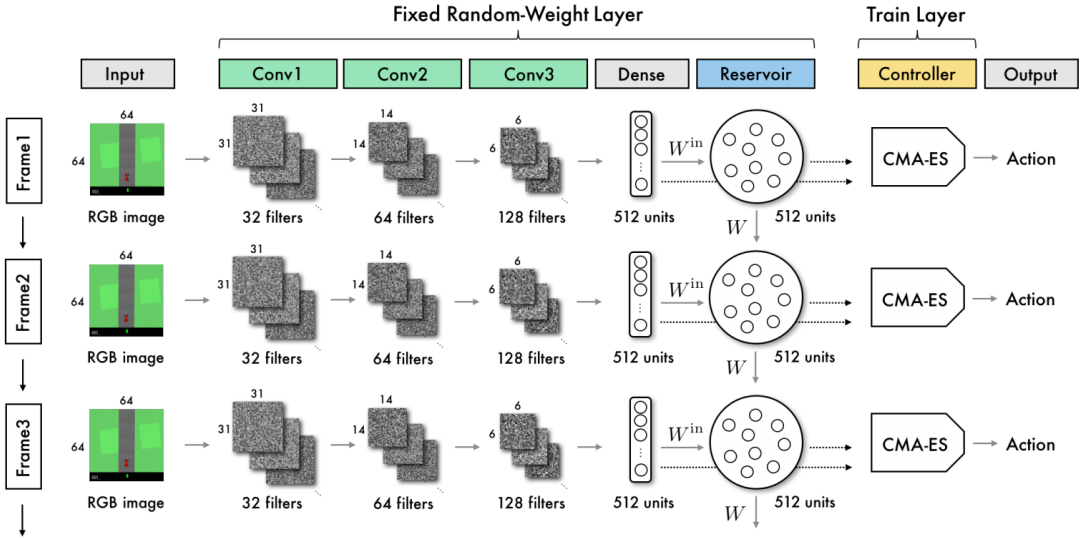

Figure 2: RCRC overview to decide the action for CarRacing-v0: the feature extraction layer are called the convolutional reservoir computing layer, and the weights are sampled from Gaussian distributions and then fixed. For DoomTakeCover-v0 we use the completely identical convolutional reservoir computing layer in CarRacing-v0. In both tasks, we train only a single weight in the controller layer to take task-dependent actions.

regression. It will not even be the case here. The high performance of RC is due to the high dimensionality of its reservoir state and thus the diversity of information processing done in each so-called neuron of the reservoir.

3. **Determining actions from the time-dependant features (2.) and the frame features (1.).** We are looking for a matrix W_{out} to map the {reservoir+frame} state to the action vector. Covariance matrix adaptation evolution strategy (CMA-ES) is the method chosen. This method can be summarized this way. The loss function is *a priori* non convex. Therefore we start by sampling several W_{out} according to a multivariate normal distribution. The fittest according to the scoring function are used to update the new mean and covariance matrix of the distribution. This algorithm can be seen as a *Natural Gradient Descent* in the space of sample distribution.

We started with 96×96 instead of 64×64 pixel as mentioned in [1]. We, therefore, decided to apply a max-pool layer before the convolutional layers. The theory of Convolutional Networks will not be detailed here since it is well-known.

3.1 Reservoir Computing

A nonlinear activation function is applied on the node's states. The different connection weights (with the input and with other nodes) and biases yield a great diversity of nonlinear transformation of the input signal and its last past states. Let $\vec{u} \in \mathbb{R}^u$ be a vector of input. At each time step i, $\vec{u_i}$ is mapped into a higher-dimensional space to form the reservoir state

$\vec{r_i} = f(\vec{u_i}, \vec{r_{i-1}}) \in \mathbb{R}^r$. f is a nonlinear function added to avoid simple proportionality between input and output and thus enables nonlinear transformation. In this study, we choose to use the tanh· function. The mappings from $\vec{r_{i-1}}$ and $\vec{u_i}$ to $\vec{r_i}$ are random and fixed using the matrices W and W^{IN} respectively. $\vec{r_{i-1}}$ introduces a memory of the reservoir past states and $\vec{u_i}$ is the input information. W, the reservoir sparse connectivity matrix, links nodes of the reservoir to other nodes of the reservoir. One should note that a node can be connected to itself (coefficients on the diagonal of W are not always zero). Here, we choose the leaky state transition function for the nodes in the reservoir. This corresponds to the coefficients a_l and $1 - a_l$ and it enables a linear memory.

The implementation in Eq. 1 also forces $\vec{r} \in [-1, 1]^r$.

$$\vec{r_i} = (1 - a_l)\vec{r_{i-1}} + a_l \tanh\left(W\vec{r_{i-1}} + \gamma W^{IN}\vec{u_i} + \vec{\zeta}\right) \tag{1}$$

Description of the hyperparameters:

- a_l is the *Leaking Rate*. $a_l \in]0, 1]$. It sets the speed of update of the reservoir states. It "can be regarded as the time interval in the continuous world between two consecutive time steps in the discrete realization" [**?**] . $(1 - a_l)$ scales the first term of equation 1. This term yields a linear memory or inertia in the reservoir.

- ρ is the *Spectral Radius*. The spectral radius is the largest absolute of W's eigenvalues. The spectral radius is critical since it scales the nonlinear memory in the reservoir. $\rho < 1$ means that $W^i \xrightarrow[i \to +\infty]{} 0_{r,r}$. In the literature it is known as the echo state property : "The state of the reservoir is uniquely defined by a fading history of the input" [**?**].

- D is the *Average Connectivity*. Concretely, it is implemented in the setting of $\frac{D}{r}$ as W's sparsity. It accounts for the average number of connections a node has. The nonlinearity increases with this parameter. Together with r which is the number of nodes, D, and ρ are the only hyperparameters necessary to describe W.

- γ is the *Input Scaling*. It scales the uniform random coefficients of the input matrix W^{IN} from $[-1, 1]$ to $[-\gamma, \gamma]$. This coefficient makes the reservoir dynamics more sensitive to the input and less dependent on its nonlinear memory.

- $\vec{\zeta}$ is the *Bias Vector*. It is a uniform random vector $\vec{\zeta} \in [-\zeta, \zeta]^r$ scaled by the scalar ζ. It allows the reservoir to explore different parts of the tanh· function's nonlinearity. $(\tanh(\mathrm{x}) \to \tanh(\mathrm{x} + \mathrm{z}))$ For example, the saturation effect found for extreme values of x would not be used without the bias. The distribution of values in $\vec{\zeta}$, W^{IN} and W follow a uniform law.

3.2 Optical Processing Units (OPUs)

Reservoir Computing hardware is a very active field of research. The randomness of higher dimension projections makes this algorithm very "novel hardware friendly" [13] [14]. The LightOn company in Paris allowed us to use its Optical Processing Units (OPUs). OPUs enable parallelized, high-dimensional very fast and energy efficient random nonlinear matrix projection. We would like to incorporate the use of this technology in Reinforcement Learning Tasks. [15]. Thanks to their technology, LightOn engineers could already outrun GPUs and

TPUs in terms of speed and energy consumption in certain situations. This was obtained in several usual Machine Learning and Deep Learning tasks with an error rate close to the state-of-the art ones. Using an "off-the-shelf" optical setup, an OPU first encodes information optically. The light is then scattered through a dispersive media which provokes the nonlinear random (but consistent) projection. The nonlinear activation function, in this case is not the hyperbolic tangent but the squared norm of the projected input since the light intensity defines the outcome of an OPU operation. The input must be encoded as a binary vector. Therefore a first dimensionality augmentation is required to avoid any loss of information. This is further details in the section below.

3.2.1 Binarization of the input

Several strategies exist to binarize the input [13]. However we will focus on only one. Let us have a vector of dimension n, $x \in \mathbb{R}^n$. One strategy could be to apply the Heavyside function on the rescaled vector, but there would be a huge loss of information. One can therefore create a random mapping $W_{bin} \in \{-1, 1\}^{d_{bin} \times n}$ to a higher dimensional space with weights of 1 with a probability of 0.5 and -1 with the same probability. The vector becomes $y = W_{bin}x \in \mathbb{R}^{d_{bin}}$. We can now apply the Heavyside function on y $H : \mathbb{R} \to \{-1, 1\}$ element-wise on x. H yields 0 for negative or 0 input and yield 1 for positive input. We therefore obtain $z = H(y) \in \{-1, 1\}^{d_{bin}}$.

3.2.2 Optical Reservoir Computing recursive equation

Let us define the function $OPU : \{-1, 1\}^d \to [\![0\ ;\ 256]\!]^{N_r}$.

Listing 2: Definition of the OPU function

```
from lightonopu import OPU
self.opu= OPU(n_components=Nr)  #defines the output dimensionality
```

Let us now concatenate the binarized feature vector $\tilde{f} \in \{-1, 1\}^{d_f}$ with the binarized reservoir vector $\tilde{r} \in \{-1, 1\}^{d_r}$. As a simplification we will not consider the leaking rate a_l in equation 1. Therefore we get :

$$r(t+1) = OPU([\tilde{r(t)}, \tilde{f(t)}]) \tag{2}$$

Using the binarization protocol again (scaling + higher dimensional mapping + Heavyside), one can compute the next step $\tilde{r}(t+1)$.

4 Challenge encountered and choices made during the programming of the project

4.1 Problem of memory leak

During the development of the program used in this project we encountered a lot of problems to solve regarding the method we used. First of all due to the use of gym, we encountered some memory leak problems that made the program crash after a few hours of simulations. Because the problem comes from the gym environment, it was not possible for us in the time given for

this project to correct this issue. However in order to continue simulation where it stopped and not lose what was calculated, we needed to implement a system to save the important data of simulation. To do so, we needed to correctly understand what was the relevant data and how the pycma library stored it during iterations of the CMA-ES method. That is why we opted to program the loop of the CMA-ES ourselves instead of using the already proposed function in the library fmin2().

4.2 Problem of device calculation

An other issue we struggled with was managing the device of calculation to perform calculations on CPU or on GPU, as some method are not available for GPU calculation and we can't make operation between variables stored in CPU and GPU.

4.3 Choice of parameters

Finally, the main problem encountered was the time of simulation needed to obtain good results. To address this issue we decided to decrease the number of simulation per iteration, however it may have a bad impact on the accuracy of the result. An other issue was the good chose of parameters such as the non linear function converting our command data into acceptable range for the car_racing program (the control values are between -1 and 1), we could either use a sigmoid or a clip function but we didn't see a huge difference between the use of those two functions. Moreover, we needed to address the issue of stopping the simulation when the car is having obviously wrong behavior. This allows to avoid calculating useless steps of simulation when we know that the car is not behaving as wanted. Therefore, we decided to stop the simulation if the car didn't pass a road section during the last 150 step, considering that in this case, the car left the road or will do unexpected behavior until the end of the simulation.

5 Implementation

In order to run the game we used the gym environment developed by openAI. Moreover we programmed the convolutional network and the reservoir computing using pytorch and used the library pycma to implement the CMA-ES in our program. To implement the training of the car controller, we created 3 .py files with each a specific role.

- main.py
- game.py
- network.py

5.1 main.py

The file main.py contains the main loop of our program and the management of saving features calculated such as: C the co-variance matrix, σ (standard deviation scaling factor), μ (mean of multivariate distribution), the number of iteration and the Convolutional Network and Reservoir used.

Before the main loop we can observe the line:

```
es = cma.CMAEvolutionStrategy(mu, 1,{'popsize':5,'ftarget':-50000,'maxiter':100000})
```

It is used to initialize the CMA-ES method, *popsize* is the number of generation of random controller vector W_out per iteration, f_{target} is a score to reach in order to stop the loop and *maxiter* is the number of maximal iteration accepted. Both are high because we never want the program to stop, we want to stop it manually if we see that the behavior of the car is satisfying.
The main loop criteria of stopping is :

```
es.stop()
```

This is a function that verify if the stopping criteria (maxiter or ftarget) are satisfied.
The function

```
es.ask()
```

generates a number popsize of random W_{outi} following a multivariate normal distribution of parameter C (covariance matrix) , σ and μ, variable updated at each iteration of the CMA-ES. It stores those information in a list called W_out.
The function

```
es.tell(Wout, [utils.game.launch_scenarios(Wouti) for Wouti in Wout])
```

is performing an iteration of the CMA-ES with the value W_{outi} and with the reward calculated by the function

```
utils.game.launch_scenarios()
```

this function take as input W_{outi} and generate a set of simulation to return the average reward value of each iteration simulation.

5.2 game.py

game.py is the file containing the function used to perform a set of simulation using the controller vector W_{outi} given in input and return the average reward of those simulations. It has two loop, one for operating each simulation, one to perform each step of simulation, including generating the command given to the car and updating the total reward.

5.3 Network.py

Network.py is the file used to generate the CNN and reservoir used to extract relevant features.

6 Description of behaviors and results

By launching our program we obtained interesting result, the car was nearly about to finish the race with little unwanted behavior. At the beginning of a simulation, the main unwanted behaviors encountered were: the car not moving or turning indefinitely. By the time going, the

car started following the road but sometimes left the road, cut the turn or completely turned around. The reward as it is calculated allows to easily avoid the case of leaving the road during a long time. However, on some iteration it was difficult to remove those behaviors. In one of the simulation the program favored the fact that the car follow the right limit between grass and the road, which lead to an oscillation of the car between the road and the grass, it was then difficult for the program to go out of this configuration. After several attempt, we obtained one simulation that nearly followed the road. However this solution still contained unwanted behavior and whatever the time we continued the simulation, it was not improving, the car was still cuting turns. The unwanted behavior we obtained that weren't displayed in the paper were surely caused by the fact that due to low computational power available, the number of vector W_{outi} generated per iteration was low (5) and the number of simulation per W_{out} were also slow (5) whereas the recommended number of random generation of W_{out} per iteration for an W_{out} of size 3075 was recommended to be 28.

7 Conclusion

We were really enthusiastic to pursue this project and actually program our first Artificial intelligence. It allowed us to better understand the underlying reasoning of artificial intelligence conception, and the different step that need to be done in order to obtain results. We unfortunately didn't manage to implement the OPU in our project due to incompatibilities of the gym environment and the platform used for OPU operations and we lacked the time to address this issue. However we managed to reproduce the methods presented in the paper, with our own parameters and reward calculation and we obtained interesting results, even if it implied letting the computer running during several hours.

8 References

[1] Hanten Chang and Katsuya Futagami. Reinforcement learning with convolutional reservoir computing. *Applied Intelligence*, 50(8):2400–2410, 2020.

[2] Qiang Wang and Zhongli Zhan. Reinforcement learning model, algorithms and its application. *Proceedings 2011 International Conference on Mechatronic Science, Electric Engineering and Computer, MEC 2011*, (1):1143–1146, 2011.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[4] Zhiqiang Tong and Gouhei Tanaka. Reservoir computing with untrained convolutional neural networks for image recognition. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 1289–1294. IEEE, 2018.

[5] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80, 2004.

[6] Wolfgang Maass and Henry Markram. On the computational power of circuits of spiking neurons. *Journal of computer and system sciences*, 69(4):593–616, 2004.

[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[8] Zhixin Lu, Jaideep Pathak, Brian Hunt, Michelle Girvan, Roger Brockett, and Edward Ott. Reservoir observers: Model-free inference of unmeasured variables in chaotic systems. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(4):041102, 2017.

[9] MA Schumer and Kenneth Steiglitz. Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13(3):270–276, 1968.

[10] David Ha and Jurgen Schmidhuber. World models. *arXiv*, 2018.

[11] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[12] Nikolaus Hansen. The CMA Evolution Strategy: A Tutorial. 2016.

[13] Jonathan Dong, Mushegh Rafayelyan, Florent Krzakala, and Sylvain Gigan. Optical reservoir computing using multiple light scattering for chaotic systems prediction. *IEEE Journal of Selected Topics in Quantum Electronics*, 26(1):1–12, 2019.

[14] Guy Van der Sande, Daniel Brunner, and Miguel C Soriano. Advances in photonic reservoir computing. *Nanophotonics*, 6(3):561–576, 2017.

[15] Photonic Computing for Massively Parallel AI. (May):1–17, 2020.

[16] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.