Design of cluster-computing architecture to improve training speed of the Neuroevolution algorithm

Iaroslav Omelianenko[1,2] (0000-0002-2190-5664)

[1] Institute of Software Systems, Kyiv, Ukraine
[2] NewGround Advanced Research LLC, Kyiv, Ukraine
yaric@newground.com.ua

Abstract. In this paper, we review the key features and major drawbacks of the Neuroevolution of Augmenting Topologies (NEAT) algorithm, such as slow training speed that limits its area of application. The main reason for the performance issues of the NEAT algorithm is the huge number of calculations required at the end of each epoch to estimate the fitness of each organism in the population. We propose a software system architecture that can be implemented to solve NEAT performance problems based on Ray cluster-computing framework. Finally, we demonstrate how fitness estimation computations can be distributed across stateless distributed workers deployed either on-premise or in the cloud using Ray framework.

Keywords: genetic algorithms, neuroevolution of augmenting topologies, ray framework, distributed computing, cluster computing, cloud computing

1 Introduction

Genetic algorithms are a perspective area of machine learning particularly suitable for tasks related to optimization and control [3, 28], reinforcement learning [20, 24], artificial life simulation [8], coevolution of swarm agents [13], etc. One of the most popular and powerful among them is a Neuroevolution of Augmenting Topologies (NEAT) [15, 23]. NEAT algorithm and it's extensions allow us to apply creative forces of evolution to evolve compact and energy efficient topologies of Artificial Neural Networks (ANN). After that, controllers based on such ANN's can be used for inference in very resource constrained environments, such as: robots, UAVs, networking edge devices, IoT devices, etc.

In this paper we start with overview of related work and short definition of NEAT algorithm fundamentals, then we describe its key features and main drawbacks, after that we give a short overview of Ray framework [12], and finally we provide proposal of system architecture that allow us to use Ray cluster-computing features to distribute fitness evaluation during simulation phase of evolutionary process controlled by NEAT algorithm.

2 Related work

The slow training speed of evolutionary algorithms, including NEAT, has troubled researches for a while. Multiple solutions has been proposed to solve this problem.

In [4], the authors proposed using a large network of volunteer computers to perform large-scale evolution of Convolutional Neural Networks (CNN). To achieve this task, they implemented modified version of the NEAT algorithm called Evolutionary Exploration of Augmenting Convolutional Topologies (EXACT). Within two months, about 4500 volunteered computers in the Citizen Science Grid were able to train about 120000 CNNs. The best performing CNNs were able to achieve 98.32% test data accuracy on the MNIST dataset. Evolved CNNs outperformed the CNNs trained using the error gradient backpropagation optimization strategy. In addition, the evolved CNN networks had an interesting "organic" topology that was very different from the standard architectures designed by humans.

Distributed computing based on asynchronous master-worker architecture is a common approach to solving the problem of slow training speed of evolutionary algorithms. This can be implemented relatively easy. However, it suffers from an obvious bottleneck. The master process should schedule tasks for workers and process all results. In [7], a method for improving the performance of the distributed NEAT algorithm is described. The authors proposed an implementation of the NEAT that offloads costly crossover and mutation operations to the remote workers. In order to achieve this, they implemented modular congruence class-based strategy to generate globally unique innovation numbers. The proposed solution does not require any communication between remote workers to generate innovation numbers, thus reducing communication overhead. Using the implemented solution, the authors were able to solve bottleneck of the Evolutionary eXploration of Augmenting Memory Models (EXAMM) neuroevolution algorithm, which prevented scaling to more than 432 cores in the a cluster. They were able to demonstrate effective scaling over 864 cores using the proposed method.

Another approach based on implementation of massively parallel genetic optimizer for High Performance Computing (HPC) systems were proposed in [26]. The authors designed the Propulate framework based on a fully asynchronous island model with independent processing workers. The island model allows you to parallelize the optimization process and distribute the evaluation of the objective function. According to the proposed model, Processing Elements (PEs), a.k.a. workers, are divided into islands. Each worker processes one individual at a time and maintains a population to track evaluated and migrated individuals on their island. In each iteration, each worker evaluates an individual that is added to its population list. It then sends the individual with the evaluation results to all workers on the same island. In return, it gets individuals evaluated by other workers and adds them to its population list. Explicit synchronization is avoided by asynchronous point-to-point communication via the Message Passing Interface (MPI). Each worker dispatches its results as soon as evaluation

is completed and checks incoming message from other workers on the island in non-blocking manner. In the next iteration, worker breeds a new individual by applying evolutionary operations of crossover and mutation to the current population list of already evaluated individuals. After mutual update of all workers, asynchronous migration and pollination between islands occurs. This is done on per worker basis with some probability. The process is repeated up to certain number of generations evaluated by each worker. After that, the population is synchronized among all workers and the best individuals are selected.

In recent works, another promising approach to the implementation of parallel Genetic Algorithms (GA) is proposed - the use of modern Graphics Processing Units (GPU). By its design, the GPU support massively parallel computations over multidimensional data vectors (tensors). However, its performance vanishes when the complexity of GPU programs (kernels) increases. To utilize the inherent parallelization capabilities of GPUs, an evolutionary computation library called EvoTorch was proposed in [27]. The EvoTorch is based on the famous PyTorch library [17], which itself is a deep learning framework designed from the ground up to support hardware accelerators such as GPUs. Using PyTorch as the basis for their library, the authors allow users to utilize an API already well-known in the industry, which has become the de-facto standard for symbolic imperative style programming in the field of Machine Learning (ML). Another big advantage of EvoTorch is that it provides a Python API, which is the most popular programming language among data scientists and ML researchers. The library allows for multiple parallelization modes that can be switched with minimal user efforts. It can easily parallelize computations using all or any GPUs available on a local computer or in a computing cluster. In addition, it supports parallelization on multiple Central Processing Units (CPUs) to address the problem of parallelizing the computations of the fitness function, which in most cases is a black-box component that is too complex to be transferred to GPU (e.g. physics simulation). CPU-bound tasks can be parallelized across multiple CPUs on a single machine or CPUs available in a compute cluster.

Recently, was published several works aimed to utilize the power of commercial computing clouds to parallelize training of the NEAT algorithm. One of the promising solutions was proposed by authors of EvoJAX library [25]. The EvoJAX library is based on JAX [2] framework which allows seamless integration with Google Cloud Platform (GCP) [19]. JAX framework allows to compile and run NumPy [5] programs on GPUs and Tensor Processing Units (TPUs) [6]. In addition, it allows to use just-in-time compilation of user defined functions into XLA (Accelerated Linear Algebra) optimized kernels using a one function API. Being based on JAX the EvoJAX library utilizes its just-in-time compilation to achieve very high performance of evolutionary process computation. The main design goal behind EvoJAX library was to improve the neuroevolution training efficiency by implementing the entire NEAT algorithm pipeline in the modern ML framework that support such crucial features as: auto-vectorization, device-parallelism, just-in-time compilation, etc.

Also, distributed evolutionary algorithms used in the modern manufacturing systems to control and manage production processes. In [1] authors proposed distributed evolutionary algorithm implementation based on Apache Spark [11] for flexible scheduling of evaluation tasks in a cloud manufacturing environment.

Reviewed works usually use implementations of GA based on Python programming language, which makes it suitable for research and development but requires additional efforts to be deployed in a production environment based on micro-service architecture under the control of Kubernetes, for example.

Our proposed solution is based on an implementation of the NEAT algorithm developed using the GO programming language, which allows the trained models to be easily deployed in a micro-service-based production environment without any modifications.

Before proceeding to the description of our solution, we will give a brief description of the main advantages and disadvantages of the NEAT algorithm. After that we give a brief description of the Ray framework fundamentals and go on to describe the design of our solution.

3 NEAT fundamentals

The most important feature of the NEAT algorithm, that defines its potential, is an ability to evolve the topology of Artificial Neural Networks during the learning process. The importance of gradual augmentation of network topology became obvious if we consider how it is implemented by NEAT. When exploring search space of solution, NEAT algorithm starts with simplest basic network topology that comprises only input and output nodes of the solver ANN. After that, with each epoch of evolution, the topology of solvers become more complex and the most complex multidimensional topologies are evaluated only at the final stages of evolutionary process. Furthermore, the most suitable topological structures found during evolution are preserved by NEAT algorithm through generations due to its inherent features.

Such approach of gradual complexification of solver's topology considerably reduces the solution search space during the training. Wide solution's search space is a huge drawback of other evolutionary algorithms which effectively addressed by NEAT due to its key design features, which we consider next.

3.1 Key features of NEAT algorithm

As we already mentioned, the main goal of the NEAT algorithm is to reduce complexity of evolved network topology during the evolution process. And this relates not only to the final product of evolution, but for all intermediate generations of solvers. To achieve this the following key features are implemented in the algorithm:

Topology augmentation through gradual complexification. Learning process starts with basic, simple topology and gradually increases its complexity

based on results of analysis of solution search space. At the same time, NEAT has ability to preserve minimal found topology that is able to generate ultimate solution.

Speciation. Is another key feature of the NEAT that separates population into evolutionary niches using genetic distance between organisms (similarity of genomes). Its main goal is to reduce negative adaptation pressure caused by increased complexity of novel organism when new node or connection added to the genome of offspring. New more complex topology can introduce novel vector for exploration of solution search space leading to success in finding ultimate solution in the future. However, in the current epoch of evolution it can be eradicated by existing evolution champions due to temporal fitness reduction. Thus, to maintain survival pressure we allow evolutionary competition only among organisms within the same niche. This allows us to keep interesting innovations within population even if they are not the most effective within whole population.

Mutation. Mutation operator plays important role in keeping genetic diversity of population during evolution and prevents objective function from stalling at local minima when the chromosomes of organisms in population become too similar. This operator changes one or more genes in chromosomes according to the mutation probability defined by experimenter. Introducing random changes into solver's chromosome, mutation allows evolutionary process to explore new areas in the search space of possible solutions and to find better solutions during evolutionary process. Mutation operator is a common feature of almost all genetic algorithms. However, specific to NEAT implementation of the operator changes not only weights of connections between neurons, but also the very structure of ANN topology.

Innovation numbers. This is the most prominent feature of the NEAT algorithm that allows to solve issue with overlapping parts of ANN solvers' genomes. These parts produce similar topologies that estimate the same function, but were embedded in different genome structures during reproductive crossover.

Crossover (recombination of genomes). Allows for two or more champion organisms within population niche to mate and produce offsprings that inherent important characteristics of both parents. It is during this process that the mentioned above innovation numbers allow us to identify overlapping (matching), excess or disjoint genes. Overlapping genes has the same value of innovation number and encode the same topological structure despite having different values for another parameters (connection weight, etc.). During reproduction, matching genes randomly selected from one of the parents to be inherited by offspring. On the other hand, genes that are present only in genome of one of the parents can be considered as disjoint (within innovation numbers of another

parent) or excess (outside innovation numbers of another parent). Disjoint or excess genes inherited by offsprings from most fit parent or from both parents randomly - this depends on type of crossover operator (multipoint, multipoint average, single-point, or uniform).

Mentioned features of the NEAT algorithm make it a powerful variant of evolutionary algorithms. However, it have a drawbacks which we are going to discuss next.

3.2 Drawbacks of NEAT algorithm

Despite the numerous advantages inherent in the NEAT algorithm, there are some problems that prevent its widespread application to many machine learning tasks.

We can name the following major drawbacks of the NEAT:

- In the NEAT algorithm, the genome of each organism in the population directly encodes a certain topology of the ANN solver. As a result, the population size and the size of encoded ANN are limiting factors because with a larger population, we need to have more computational resources to evaluate the evolutionary process. Thus, we can effectively work only with relatively small ANN topologies and moderately sized populations.
- The chosen objective function applied to calculate the fitness of organisms in a population works well for simple tasks, but often collapses into local minima traps for more complex tasks.
- Slow training associated with large number of calculations at the stage of fitness evaluation of each organism in the population.

The first problem, related to the large population of organisms encoding topologically complex ANN structures, can be addressed using extensions of the original NEAT algorithm, where the structure of the ANN solver is encoded as points in a multidimensional hypercube. The configuration of such structure produced by much smaller ANN named Compositional Pattern Producing Network (CPPN) [21] that learns how to encode large ANN of solver within hypercube. There are at least two extensions of NEAT based on hypercube encoding: HyperNEAT and ES-HyperNEAT [15, 18, 22]. With the help of these extensions, it is even possible to solve tasks related to visual pattern recognition, where deep machine learning methods traditionally shine.

The second problem can be addressed by replacing the goal-oriented objective function with another search optimization method, such as Novelty Search [9, 10]. In this method, a specific goal-oriented objective function is not defined or applied during search for solution. Instead, the novelty of each found solution is directly rewarded during the process of evolution. Thus, the novelty of found solutions directs the evolution towards the ultimate goal. Such approach gives us a chance to use creative forces of evolution regardless of adaptive pressure to expand the solution's search space.

Finally, the last problem can be solved by applying parallel distributed computing to evaluate the fitness of each organism in the population. Next, we are going to consider how this can be implemented using Ray framework for distributed computing [12].

4 Ray framework fundamentals

Ray is a general-purpose distributed cluster-computing framework developed within University of California, Berkley to meet the ever-growing need for distributed parallel computing for Reinforcement Learning tasks. The main goal behind Ray framework was to develop framework which is able to handle heterogenous task in massively distributed manner to reach more than million scheduled tasks per second with milliseconds-level latencies [12]. The heterogeneity of tasks considered both in the terms of execution time (e.g. a task takes milliseconds or hours) and resource usage (e.g. tasks can be executed by GPUs, CPUs, or TPUs).

There are two main abstractions defined in Ray framework to support massive parallel computing:

- **Task** - is to enable stateless distributed computing, suitable for efficient and dynamically load balanced tasks related to simulation (RL rollouts, fitness evaluation of genomes, etc), large inputs processing (e.g. processing videos, images), and allows for easy recovery after failures.
- **Actor** - is to allow stateful distributed computations, such as training an ML model, running stateful simulations, etc.

From programming point of view a *task* represents the execution of remote function on a stateless worker. The outputs of remote functions are determined exclusively by their inputs, that is, the call to such a function is idempotent. The stateless workers suitable for fine-grained load balancing and efficient failure handling because they can be started and stopped without worrying about loosing any computational state.

On the other hand, an *actor* represents a remote object that exposes multiple methods that can be executed sequentially while maintaining computational state between invocations. It is handled by stateful workers, which are coarse-grained load balanced, introduce extra overhead from checkpointing, but allow to perform many small updates without extra overhead.

The call to schedule *task* or *actor* for remote execution by Ray framework returns immediately with a *future* representing result of execution. This gives us the flexibility to build parallel data flows in our applications and introduce different synchronization schemes to achieve our goals.

The core of the Ray framework is implemented using C++ programming language, and at the time of this writing, it provides several APIs:

- **Python API** - the main API to interact with Ray framework, supporting all kinds of operations defined in the framework.

– **Java API** - is an experimental API to interact with Ray framework, supporting only subset of functions.
– **REST API** - to monitor and manage remote jobs, serve deployments, etc.

Such flexibility of the provided APIs allows us to use Ray framework in various contexts and on different platforms. In addition, it's worth noting that Ray provides seamless integration with most popular cloud platforms, such as: AWS, GCP, and Azure. This allows us to deploy our clusters on a virtually unlimited number of computing nodes in the cloud.

Now that we've covered the basics of the Ray framework, we're ready to look at how it can be used to accelerate NEAT training.

5 Distributed training of NEAT algorithm with Ray framework

As we mentioned, simulation to estimate fitness of each organism in the population is the most time-consuming part of training using NEAT. Especially in tasks that require a large population size and a complex fitness assessment environment. Unfortunately, most real-world tasks fall into this category, which greatly hinders the widespread adoption of NEAT algorithm.

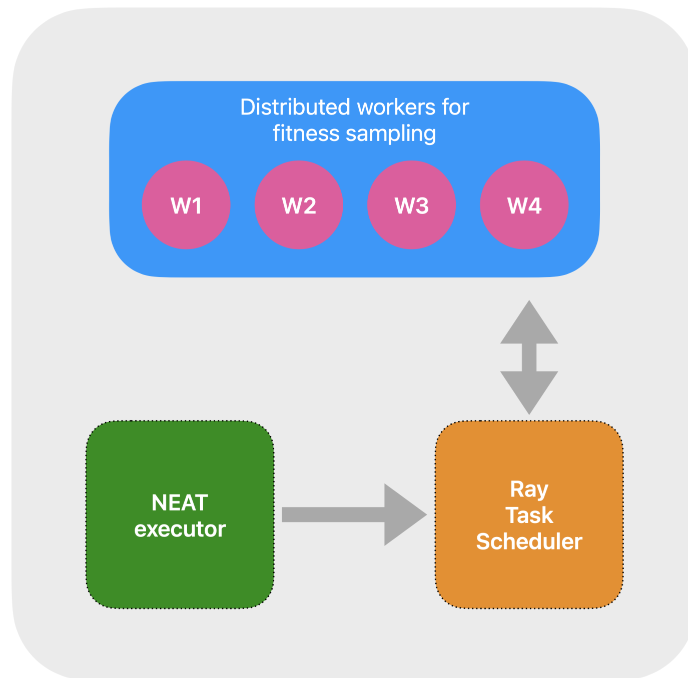

Fig. 1. Overall system architecture

In order to support distributed training using NEAT algorithm we propose architecture shown in the figure 1. Where:

- **NEAT executor** - main process executing evolutionary training using NEAT.
- **Ray Task Scheduler** - Ray driver application scheduling remote tasks executions and exposing RPC API interface for NEAT executor.
- **Distributed workers** - cluster with Ray stateless workers to perform fitness estimation simulation for each organism in the population.

The NEAT algorithm implements a direct genome encoding scheme, which allows generating a solver ANN directly from organism's genome [14, 15, 23]. At the same time, the implemented genome encoding scheme has a very compact textual representation that can be easily encapsulated into standard POST HTTP requests.

An example of the NEAT compact genome coding scheme as following:

```
genomestart 1
trait 1 0.1 0 0 0 0 0 0 0
trait 2 0.2 0 0 0 0 0 0 0
trait 3 0.3 0 0 0 0 0 0 0
node 1 0 1 3 NullActivation
node 2 0 1 1 NullActivation
node 3 0 1 1 NullActivation
node 4 0 0 2 SigmoidSteepenedActivation
gene 1 1 4 0.0 false 1 0 true
gene 2 2 4 0.0 false 2 0 true
gene 3 3 4 0.0 false 3 0 true
genomeend 1
```

Using this encoding scheme, we can easily encode the genomes of one organism or all organisms in a population together. As a result, at the end of each evolutionary epoch, we can send the encoded genomes of all organisms in the population for evaluation by distributed Ray workers in just one network call, which greatly reduces the latency of this operation.

An algorithm describing NEAT training using remote distributed workers for fitness estimation is shown in the Listing 1.

Upon receiving the encoded genomes of all organisms in the population, the *Ray Task Scheduler* reads the encoding of each individual genome and schedules tasks for remote fitness evaluation by *stateless* cluster workers. In the Listing 2, we provide the definition of an algorithm for distributed fitness estimation.

Each remote worker in the Ray cluster receives the encoded genome of the organism as input, decodes it into an ANN solver, and applies it to solve the simulated objective function. The estimated fitness of the organism and a flag indicating if the objective function was solved are then returned to the *Ray Task Scheduler*. An algorithm describing evaluation of fitness of organism's genome by remote worker is shown in the Listing 3.

Algorithm 1: NEAT training with distributed fitness estimation

```
Input: A number of Epochs of evolution to execute.
Result: List of found successfull solvers or empty list.
begin
    1. Initialize randomized Population of simplest organisms;
    foreach epoch ∈ Epochs do
        // Encode Population genomes
        encoded ⟵ EncodePopulationGenome(Population) ;
        // Send encoded Population genomes for distributed evaluation
        results ⟵ EvaluatePopulationFitnessRemote(encoded) ;
        foreach organism ∈ Population do
            organism.Fitness ⟵
              results.GetOrganismFitness(organism.ID) ;

        if results.HasBeenSolved() then
            return results.SuccessfullSolversList()
        else
            2. Run NEAT algorithm over evaluated Population to produce
              new Population for the next epoch of evolution
```

Algorithm 2: Distributed fitness evaluation of each organism in the population

```
Input: A Population of organisms at the epoch end.
Result: Results with fitness values of each organism in the population.

FutureResults ⟵ List() ;
foreach genome ∈ Population do
    // Schedule genome fitness evaluation by ray cluster
    future ⟵ GenomeFitnessRemote(genome) ;
    // Add future with result to the list
    FutureResults ⟵ FutureResults ∪ future ;

// Wait for futures to be completed by remote workers of Ray
   cluster
Results ⟵ Ray.get(FutureResults) ;
```

Algorithm 3: Fitness evaluation of organism's genome by remote worker

```
Input: A Genome of organism to be evaluated.
Result: Fitness value of organism after evaluation.
Result: Solved flag to indicate if objective was solved by provided organism.

solverAnn ⟵ CreateANN(Genome) ;
// Run simulation environment against ANN created
// from genome of particular Organism
Fitness, Solved ⟵ RolloutEnvironmentSimulation(solverAnn) ;
```

Ultimately, *Ray Task Scheduler* collects data from all remote workers and return it back to the *NEAT executor* in one batch. Using the received fitness values of each organism in the population, the *NEAT executor* creates a new population of organisms for the next epoch of evolution, or terminates evolution if solution has already been found.

6 Conclusions

Despite all the powerful features inherent in evolutionary algorithms, they are still not widely used due to slow training speed. We demonstrate how this obstacle can be avoided by using modern cluster-computing framework that allows us to move the most time-consuming step of evolutionary training to an almost limitless cloud infrastructure. This article presents the general architecture of such system based on Ray framework, which was specifically designed to handle scheduling of millions of tasks per second. Next, we are going to implement proposed architecture using an implementation of the NEAT algorithm written in the GO language [16].

Using proposed design, it will be possible to develop industrial micro-service systems with trained NEAT models without any additional model tuning. This will significantly reduce the maintenance time of ML operations cycle. In addition, such approach allows implementation of self supervised learning systems that continuously train the deployed models.

References

[1] Jihong Yan Abdelrahman Elgendy and Mingyang Zhang. “A Parallel distributed genetic algorithm using Apache Spark for flexible scheduling of multitasks in a cloud manufacturing environment”. In: *International Journal of Computer Integrated Manufacturing* 0.0 (2023), pp. 1–16. DOI: 10.1080/0951192X.2023.2228277. eprint: https://doi.org/10.1080/0951192X.2023.2228277. URL: https://doi.org/10.1080/0951192X.2023.2228277.

[2] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[3] Edgar Buchanan et al. “Bootstrapping Artificial Evolution to Design Robots for Autonomous Fabrication”. In: *Robotics* 9.4 (2020). ISSN: 2218-6581. DOI: 10.3390/robotics9040106. URL: https://www.mdpi.com/2218-6581/9/4/106.

[4] Travis Desell. “Large Scale Evolution of Convolutional Neural Networks Using Volunteer Computing”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO ’17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 127–128. ISBN: 9781450349390. DOI: 10.1145/3067695.3076002. URL: https://doi.org/10.1145/3067695.3076002.

[5] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[6] Norman Jouppi et al. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (2018), pp. 10–19. DOI: 10.1109/MM.2018.032271057.

[7] Joshua Karns and Travis Desell. "Improving the Scalability of Distributed Neuroevolution Using Modular Congruence Class Generated Innovation Numbers". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '21. Lille, France: Association for Computing Machinery, 2021, pp. 1299–1307. ISBN: 9781450383516. DOI: 10.1145/3449726.3463202. URL: https://doi.org/10.1145/3449726.3463202.

[8] Christopher G Langton. *Artificial life : the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987 in Los Alamos, New Mexico*. 1989.

[9] Joel Lehman and Kenneth O. Stanley. "Abandoning Objectives: Evolution through the Search for Novelty Alone". In: *Evol. Comput.* 19.2 (June 2011), pp. 189–223. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00025. URL: https://doi.org/10.1162/EVCO_a_00025.

[10] Joel Lehman and Kenneth O. Stanley. "Novelty Search and the Problem with Objectives". In: *Genetic Programming Theory and Practice IX*. Ed. by Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore. New York, NY: Springer New York, 2011, pp. 37–56. ISBN: 978-1-4614-1770-5. DOI: 10.1007/978-1-4614-1770-5_3. URL: https://doi.org/10.1007/978-1-4614-1770-5_3.

[11] Xiangrui Meng et al. "MLlib: Machine Learning in Apache Spark". In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 1235–1241. ISSN: 1532-4435.

[12] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. 2018. arXiv: 1712.05889 `[cs.DC]`. URL: https://arxiv.org/abs/1712.05889.

[13] Iaroslav Omelianenko. "Artificial Swarm Intelligence and Cooperative Robotic Systems". In: *Preprints* (Jan. 2019). DOI: 10.20944/preprints201901.0282.v1. URL: https://doi.org/10.20944/preprints201901.0282.v1.

[14] Iaroslav Omelianenko. "Autonomous Artificial Intelligent Agents". In: *Machine Learning and the City*. John Wiley & Sons, Ltd, 2022. Chap. 12, pp. 263–285. ISBN: 9781119815075. DOI: https://doi.org/10.1002/9781119815075.ch21. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119815075.ch21. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119815075.ch21.

[15] Iaroslav Omelianenko. *Hands-On Neuroevolution with Python: Build high-performing artificial neural network architectures using neuroevolution-based algorithms*. Birmingham, UK: Packt Publishing Ltd, 2019. ISBN: 9781838824914.

[16] Iaroslav Omelianenko. *The GoLang implementation of NeuroEvolution of Augmented Topologies (NEAT) algorithm*. Version v4.0.1. July 2023. DOI: 10.5281/zenodo.8178789. URL: https://doi.org/10.5281/zenodo.8178789.

[17] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Process-*

ing Systems. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[18] Sebastian Risi and Kenneth O. Stanley. "An Enhanced Hypercube-Based Encoding for Evolving the Placement, Density, and Connectivity of Neurons". In: *Artificial Life* 18.4 (2012), pp. 331–363. DOI: 10.1162/ARTL_a_00071.

[19] Jay Shah and Dushyant Dubaria. "Building Modern Clouds: Using Docker, Kubernetes and Google Cloud Platform". In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479.

[20] Nils T Siebel and Gerald Sommer. "Evolutionary reinforcement learning of artificial neural networks". In: *Int. J. Hybrid Intell. Syst.* 4.3 (Oct. 2007), pp. 171–183. DOI: 10.3233/HIS-2007-4304.

[21] Kenneth O Stanley. "Compositional pattern producing networks: A novel abstraction of development". In: *Genetic Programming and Evolvable Machines* 8.2 (June 2007), pp. 131–162. DOI: 10.1007/s10710-007-9028-8.

[22] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks". In: *Artificial Life* 15.2 (2009), pp. 185–212. DOI: 10.1162/artl.2009.15.2.15202.

[23] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: 10.1162/106365602320169811.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. Cambridge, MA, USA: MIT Press, 1998.

[25] Yujin Tang, Yingtao Tian, and David Ha. "EvoJAX: Hardware-Accelerated Neuroevolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '22. Boston, Massachusetts: Association for Computing Machinery, 2022, pp. 308–311. ISBN: 9781450392686. DOI: 10.1145/3520304.3528770. URL: https://doi.org/10.1145/3520304.3528770.

[26] Oskar Taubert et al. "Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations". In: *High Performance Computing*. Ed. by Abhinav Bhatele et al. Cham: Springer Nature Switzerland, 2023, pp. 106–124. ISBN: 978-3-031-32041-5.

[27] Nihat Engin Toklu et al. *EvoTorch: Scalable Evolutionary Computation in Python*. 2023. arXiv: 2302.12600 [cs.NE]. URL: https://arxiv.org/abs/2302.12600.

[28] Ruoshi Wen et al. "Neuroevolution of augmenting topologies based musculor-skeletal arm neurocontroller". In: *2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. 2017, pp. 1–6. DOI: 10.1109/I2MTC.2017.7969727.