Neural Network Architecture Search for Ball Detection Using a Distributed and Scalable Genetic Algorithm

Konrad Valentin Nölle*, Hendrik Sieck*, and Pascal Gleske

RobotING@TUHH e.V. (HULKs), Hamburg University of Technology (TUHH)

Abstract. The increasing application of artificial neural networks (ANN) in various domains of robotics with limited processing capabilities demands highly optimized ANN architectures. Efficient architecture search requires horizontal and vertical scaling of ANN evaluation. In this paper the HULKs present their approach and application of a scalable genetic algorithm based on distributed task execution which can be found at https://github.com/HULKs/ditef. The framework is able to scale dynamically and operate on heterogeneous hardware. Usage of the distributed genetic algorithm to evolve ANNs for the context of RoboCup soccer competitions is shown as a case study.

1 Introduction

Determining the ball position accurately using NAO robots is one of the most important problems in RoboCup Standard Platform League (SPL) games. As shown previously by Budden et al. [2], Teimouri et al. [16], and Felbinger et al. [6], convolutional neural networks (CNN), a variant of artificial neural networks (ANN), are well suited for the task of detecting SPL balls. Since training times for this use case are rather short[1], optimization algorithms can be used to find optimal architecture and hyperparameters. In particular, Elsken et al. [4] and Ren et al. [14] show that genetic algorithms can be used to optimize a CNN's architecture and hyperparameters. The execution time of genetic algorithms for neural architecture search is dominated by the time it takes to train and evaluate different neural networks. Since their evaluation is independent, the algorithm can be sped up significantly by parallelization.

1.1 Requirements

To evolve ANNs with a scalable genetic algorithm, search frameworks are required to scale horizontally and vertically i.e. efficiently utilize resources depending on particular computing capabilities and number of available computers. For

* Equal contribution

1 Small CNNs take several minutes to train, compared to days or weeks for state-of-the-art neural networks.

example at HULKs, computing resources are heterogeneous and shared with other experiments. Efficient utilization of this hardware needs dynamic coordination and scalability. Memory-intensive tasks must be prevented from running on low-memory hardware and evaluation must be scaled on demand depending on other experiments. Neural network libraries, such as Tensorflow [1] and PyTorch [12], are commonly used to train and evaluate neural networks. These libraries represent complex and volatile dependencies due to their active development. To increase flexibility, evaluation code must be separated from algorithm code in the distributed system.

1.2 Related Work

"Distributed Evolutionary Algorithms in Python" (DEAP) [7] is an evolutionary computation framework that implements state-of-the-art genetic algorithms and corresponding tools. DEAP uses "Scalable Concurrent Operations in Python" (SCOOP) [8] for distributed evaluation. Representing neural network architectures requires complex data structures. However, DEAP operates on list based data structures which require complicated conversion procedures to encode this information. For the use case of this paper it is necessary to dynamically add and remove worker instances, but SCOOP requires a static list of hosts at startup. Furthermore, in SCOOP, both the algorithm and the evaluation code must be defined in a single Python file or imported from it which prevents dependency separation. DEAP waits for all pending CNN candidates to be evaluated before advancing. This can lead to idle workers when evaluating the last pending CNNs.

Ray Tune [9] is an optimization framework providing distributed hyperparameter tuning. It supports many state-of-the-art optimization algorithms and is built upon the Ray Core [15] distributed task execution framework. Similar to SCOOP, Ray Core does not support separating algorithm and evaluation code. Since Ray Tune does not include a genetic algorithm, it needs to be implemented to fit the use case of this paper. Although it has a graphical user interface for analyzing suggestions, Ray Tune lacks introspection of the algorithm which hinders debugging when implementing new algorithms.

"Genetic Algorithm Framework in pyThon" (GAFT) [18] is a framework for genetic algorithms that can be parallelized by using the Message Passing Interface (MPI). As with the previous frameworks, GAFT does not support separating algorithm and evaluation code. Like in DEAP, the genetic algorithm waits for evaluation of all pending CNNs before advancing which reduces parallelization.

In conclusion, none of the presented frameworks meet all of the requirements. Some genetic algorithm implementations lack parallelism and therefore are not scalable. Furthermore, distributed task execution frameworks sometimes do not support constrained execution for heterogeneous hardware. Most genetic libraries tightly couple the algorithm with evaluation which violates the code separation requirement.

1.3 Contribution

To overcome the limitations of existing frameworks, this paper presents a distributed, dynamically scalable genetic algorithm capable of evolving neural network hyperparameters. The implementation consists of a generic distributed task execution framework with code separation and a restructured genetic algorithm for better resource utilization.

2 Prerequisites

As an example, the distributed and dynamically scalable genetic algorithm can be used to evolve neural networks. In the RoboCup SPL the NAO robot is used for playing soccer in a competition. An important task for the robot is to see the ball during the competition using its two cameras (Figure 1). According to Zhao et al. [17] and Erhan et al. [5], object detection in spatially related data such as visual imagery is a problem that can be solved with CNNs. They are often used either on the whole image or smaller samples. The HULKs operate on 32×32 pixel grayscale samples from the 640×480 pixel original images. Typically, ANNs are organized in layers, for example fully-connected layers or convolutional layers and their extensions. Activation functions make deep neural networks non-linear to allow approximation of any function if the neural network has sufficient complexity as shown by e.g. Cybenko [3]. Optimizers enable fitting ANNs to training data. The set of layers and activation functions of neural networks is called its architecture. Layers, activation functions, and optimizers have configuration parameters which are called hyperparameters. The architecture and hyperparameters have to be tuned for a specific application.

Since the NAO robot has limited processing capabilities[2] it can only feasibly execute neural networks that are small and fast. For detecting balls at the NAO camera's frame rate, images must be evaluated 60 times per second. Hence, state-of-the-art CNNs for visual object detection, like YOLO [13] or RetinaNet [10], are too complex for the available processing power. The ball detection at HULKs is implemented by filtering several hundred 32×32 pixel samples for a potential ball in each frame (Figure 1, right). This results in 10^3 to 10^4 inferences per second which requires sub-millisecond evaluation time per sample.

Genetic algorithms as described by Mitchell [11] can be used to generate efficient CNNs and find optimal neural network architectures and hyperparameters for the ball detection as mentioned by Budden et al. [2], Teimouri et al. [16], Felbinger et al. [6], Elsken et al. [4], and Ren et al. [14]. Genetic algorithms are optimization algorithms which simulate natural genetic processes. A genetic algorithm contains one or more populations with multiple individuals. These individuals are represented with a genome and can evaluate to a fitness metric. A genetic algorithm is a solution suggestion algorithm where operations on the populations and individuals generate new candidate solutions. New genomes are

[2] CPU: Intel® Atom® Processor E3845, GPU: Intel® HD Graphics for Intel® Atom® Processor Z3700

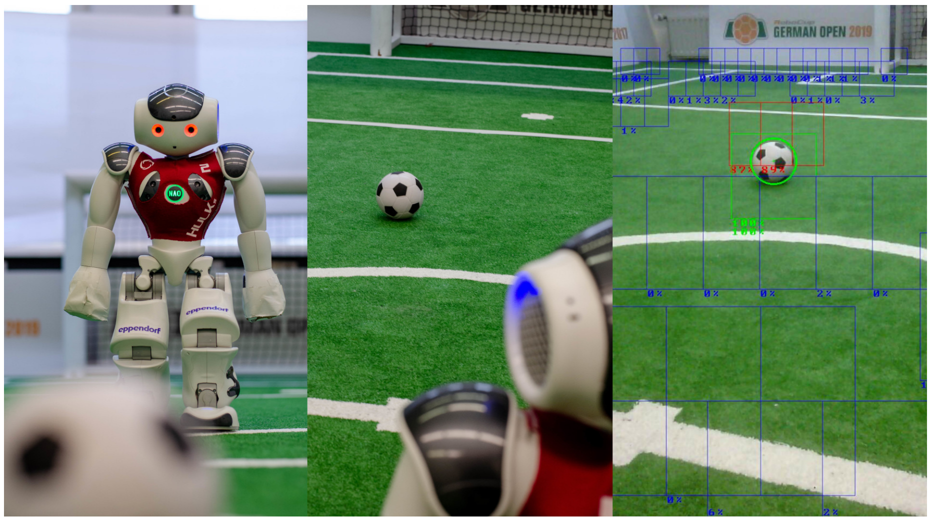


Figure 1: NAO robot in a ball detection situation. On the left, NAO robot with its two cameras in the head. In the middle, a ball from the robots perspective. On the right, an image from the robot's camera augmented with debug information about the ball detection.

created by randomly generating, mutating, or recombining other individual's genes and genomes. Through these operations, genetic algorithms continuously evolve their populations. Moreover, they improve the average fitness over time by preferring fitter individuals.

Genetic algorithms require repeated evaluation of candidate solutions, for example, training and evaluating neural networks. Since this step is quite slow, it is a good target for optimization. Parallelization can significantly speed up the genetic process by evaluating multiple candidates at the same time. To allow utilization of more than one computer, communication and discovery must be coordinated in a network. Distributed task execution frameworks exist to solve this common problem. In such frameworks, evaluations of genetic individuals can be modelled as tasks. These frameworks usually consist of task producers, task workers, and optionally a task router. A genetic algorithm generates new tasks as a task producer. Evaluations of candidate solutions are performed by task workers. The task router coordinates between the other components, queueing pending tasks, and delivering results back to task producers.

3 Implementation

This section is split into the implementation of a distributed task execution framework and a genetic algorithm based on it.

3.1 Distributed Task Execution Framework

The task execution framework (Figure 2) follows the general architecture of task execution frameworks outlined previously in Section 2: Task producers submit new abstract computation tasks and receive their results. Task workers receive tasks and generate results by executing them. A task router acts as central server to which task producers and task workers can be dynamically connected and disconnected while the system is running.

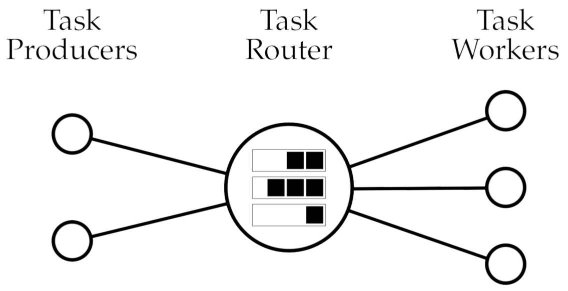


Figure 2: Distributed task execution framework consisting of task producers, task router, and task workers. The task router in the center contains a schematic representation of the multi-queue.

When a task producer wants to execute a task, it sends a representation of the task to the task router. Task representations consist of a task type and payload. The task router receives tasks and appends them to the queue corresponding to the task type. When a task worker is idle, it requests a new task of one of its supported types from the task router. The task router tries to deliver a task from one of the requested queues to the task worker. Once a task is available, it is dequeued and marked as reserved to prevent another execution. A task worker receiving a task executes it generating the execution result. The executed code is selected via the received task type. The result is returned to the task router where it is forwarded to the task producer which queued the task.

The implementation of task producer and task router is written in Python. Concurrent programming paradigms of the Python `asyncio` library are used for efficient resource usage in input/output intense and networking sections.

In order to achieve the required code separation, task producers and task workers are implemented in independent Python modules and executed as separate programs. The executed code for each task is implemented in a Python module that is imported by the task worker and selected via the received task type.

Communication occurs over the Hypertext Transfer Protocol (HTTP) where the task router provides the server and both task producer and task worker act as clients. HTTP allows implementing specialized task workers in a different language than Python by only implementing the HTTP interface between task router and task worker. The application programming interface client (API client) used in the task producer supports interleaving of HTTP requests which allows concurrent task execution.

The central task router implements a multi-queue that stores all requested tasks of different types until they are assigned to an idle task worker. The task router probabilistically selects a task from the multi-queue to ensure fairness during task assignment. Once assigned, the tasks are dequeued and stored in a separate pool. After a timeout caused by missing task worker heartbeats, tasks in this pool are returned to the tail of the multi-queue. The use of the multi-queue allows for heterogeneous task workers to coexist in the network and be connected to the same task router because they can request different task types. In practice, task workers may be instantiated dynamically by the user according to the available computing resources. This allows careful scaling to prevent resource exhaustion during algorithm execution without interrupting other components of the system.

All components are designed to be resilient against network disconnects and program crashes. This is achieved by attempting to reconnect and sending regular heartbeats with corresponding timeouts. Exceeded timeouts cause the components to retry previously requested operations which results in strong fairness. If the probability for connection failures and process crashes is below one, an execution of a task will eventually succeed, i.e. the result is received at the task producer. This liveness property holds because the task producer and task router retry the execution.

Each task worker executes tasks sequentially but can support multiple task types which are requested from the task router. However, even when requesting multiple task types, the task router always returns only a single task matching one of the requested task types.

As an example in Listing 1.1 the summation of two numbers is implemented with a task producer and the result is printed to the standard output of the program. First, an API client is constructed with the address to the task router. After that, a new task of type `summation` with the payload of two numbers is created and queued. After execution the result is returned and stored in the `result` variable which is printed to the standard output.

```python
from ditef_router.api_client import ApiClient

async with ApiClient(url='http://task-router:8080') as client:
    result = await client.run(
        task_type='summation',
        payload=[42, 1337],
    )
    print(result)  # 1379
```

Listing 1.1: `producer.py` implements a simple task producer.

The implementation of the task worker module in Listing 1.2 for the task producer only consists of a `run()` function which is loaded by its module name and executed for every task. It receives the payload of a task and returns its result which in this example is the sum of the payload.

```python
def run(payload):
    return sum(payload)
```

Listing 1.2: `summation.py` implements a task worker module for summation.

This simple summation example can be expanded to evolving neural networks using a genetic algorithm.

3.2 Genetic Algorithm

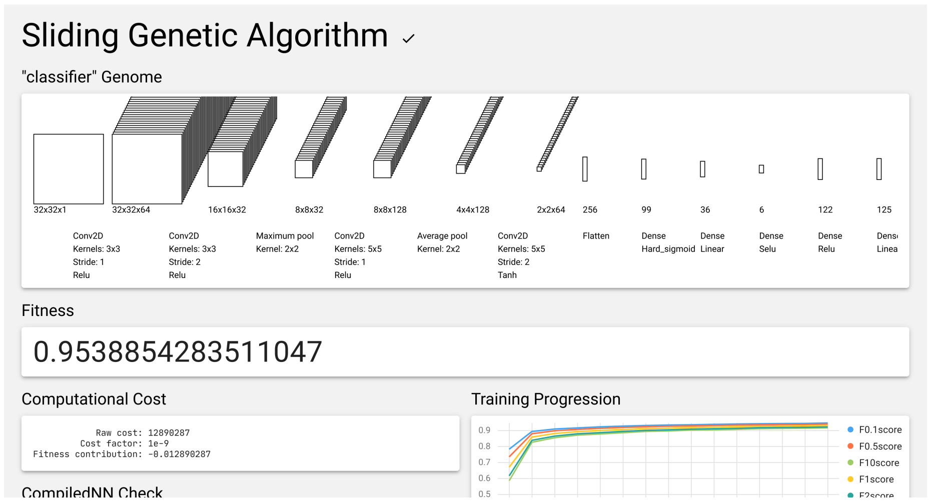
The genetic algorithm is implemented as a task producer in the aforementioned distributed task execution framework. The evaluation of an individual is represented by a task. When the algorithm generates a new individual with its genetic operations, the genome acts as the task payload. The task worker calculates the neural network metric and sends the result back to the algorithm where it is used for determining the fitness metric.

Genetic algorithms e.g. by Felbinger et al. [6] that evolve entire populations at a time can lead to idle task workers towards the end of each generation. To avoid idle time, this paper presents a sliding window approach where single individuals are added or removed from a persistent population over time. The number of unevaluated individuals is configurable and realized by running more than one loop per population.

To implement this sliding window approach, the main loop of the algorithm is split into three steps. First, a configurable minimum population size is ensured by spawning randomly generated individuals. Next, one of four spawning strategies is chosen at random and used to create a new individual. Lastly, if the population size exceeds an upper threshold, already evaluated individuals with the lowest fitness value are culled successively. Keeping the best individuals across iterations is also called elitism. To prevent less fit individuals from being removed immediately, the population is also culled probabilistically below the maximum size. These steps are repeated indefinitely.

New individuals can be created in four different ways. When generating an individual by *migration* between populations, a random member is selected and cloned from a random population. Another strategy is to *clone* an existing individual of the population where the whole genome is copied and random mutations are applied. The third method is to generate a *random* individual. However, the generation of a random genome may be non-trivial and depends on the individual type. Therefore, individual types provide their own random generators. The last strategy to generate a new individual is by *cross-over* where two parent individuals are selected from the population at random. The new genome is then constructed from genes of both parents before random mutations are applied.

The genetic algorithm also serves a web interface which allows interaction and debugging of the genetic algorithm. It can be used to change configuration parameters, add and remove populations, as well as monitoring. An individual's configuration and visual representation of the neural network architecture is shown in the individual view (Figure 3). It also shows parents, children, and the spawning strategy of the selected individual. Population views (Figure 4) contain a list of its members, statistics about the individuals, and graphical plots showing the history of the fitness metric and population size.


Sliding Genetic Algorithm ✓

"classifier" Genome

Fitness

0.9538854283511047

Computational Cost

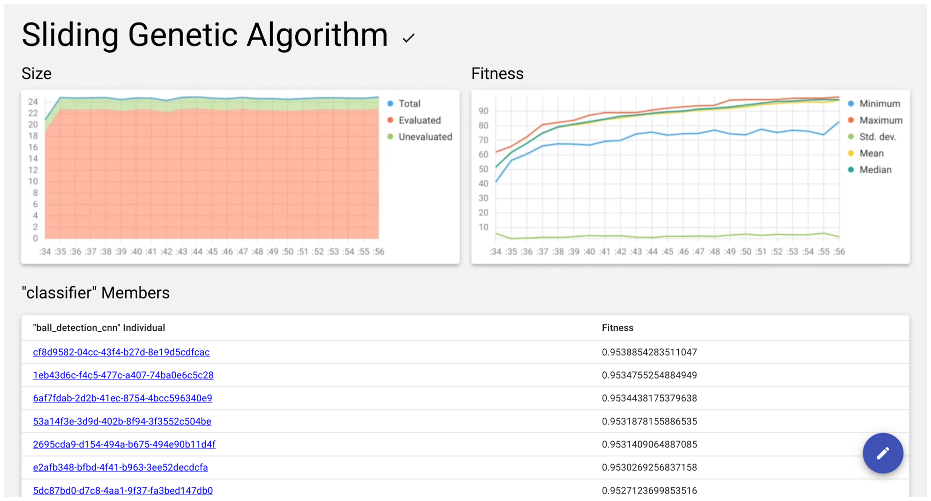
```
         Raw cost: 12890287
      Cost factor: 1e-9
Fitness contribution: -0.012890287
```

CompiledNN Check

Training Progression

Figure 3: Individual view of the web interface for interaction and debugging consisting of a genome visualization, fitness score, and other training information.

Individuals of genetic algorithms must be specified in the problem domain, in particular the genome and fitness representation. The genome in this implementation represents the network architecture and hyperparameters:



Sliding Genetic Algorithm ✓

"classifier" Members

"ball_detection_cnn" Individual	Fitness
cf8d9582-04cc-43f4-b27d-8e19d5cdfcac	0.9538854283511047
1eb43d6c-f4c5-477c-a407-74ba0e6c5c28	0.9534755254884949
6af7fdab-2d2b-41ec-8754-4bcc596340e9	0.9534438175379638
53a14f3e-3d9d-402b-8f94-3f3552c504be	0.9531878155886535
2695cda9-d154-494a-b675-494e90b11d4f	0.9531409064887085
e2afb348-bfbd-4f41-b963-3ee52decdcfa	0.9530269256837158
5dc87bd0-d7c8-4aa1-9f37-fa3bed147db0	0.9527123699853516

Figure 4: Population view of the web interface for interaction and debugging.

Convolutional layers Each layer: type, kernel size, activation function, pooling type, pooling size, batch normalization, drop out rate, stride
Dense layers Each layer: size, activation function, batch normalization, drop out rate
Optimizer Type
Learning rate Initial learning rate and factor per epoch
Training epochs Number of training epochs

As mentioned before, genetic or random operations must be implemented by the individual type. This is necessary because adding or removing layers that in turn have their own set of parameters is hard to model with generic mutation methods. Generating random genomes may result in incompatible layer types and sizes. The fitness of neural network individuals corresponds to a neural network performance metric on a test dataset.

4 Conclusion and Outlook

This paper presents a scalable genetic algorithm based on distributed task execution. The distributed task execution framework allows scaling task execution horizontally and vertically by dynamically adding or removing task workers. Within the genetic algorithm, or task producer, the iteration loop may be executed multiple times in parallel to increase concurrency of the individual evaluation. The task workers request tasks from the task router of a subset of supported types, allowing them to select which task types they want to execute. This feature can be used to distribute work selectively across the task workers by configuring which types they request from the task router. The system is robust against network failures, by simply resuming once the connection is re-established. Code separation in the presented distributed task execution framework allows running tasks with vastly different dependencies without requiring all nodes of the network to fulfill these dependencies. For example, only the neural network task worker module has a neural network library dependency.

Abstractions introduced by the implementation of the task producer allows swapping algorithms and individuals without overhead. This is achieved by introducing a common interface between algorithms and individuals where individuals provide methods for executing genetic operations which are called by the algorithm.

In the future, the presented genetic algorithm could be applied to other problem domains such as robot motion, game state estimation, and robot role assignment. Different solution suggestion algorithms instead of a genetic algorithm may be implemented, such as tree search or particle swarm optimizers. Task workers can be executed in any environment, evaluating vastly different tasks. Therefore, a possible future direction is to run task workers directly on the NAO robot, for example by evaluating walking parameters on the hardware.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
2. Budden, D., Fenn, S., Walker, J., Mendes, A.: A novel approach to ball detection for humanoid robot soccer. In: Thielscher, M., Zhang, D. (eds.) AI 2012: Advances in Artificial Intelligence. pp. 827–838. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
3. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems **2**(4), 303–314 (1989)
4. Elsken, T., Hendrik Metzen, J., Hutter, F.: Neural Architecture Search: A Survey. arXiv e-prints arXiv:1808.05377 (Aug 2018)
5. Erhan, D., Szegedy, C., Toshev, A., Anguelov, D.: Scalable object detection using deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (June 2014)
6. Felbinger, G.C., Göttsch, P., Loth, P., Peters, L., Wege, F.: Designing convolutional neural networks using a genetic approach for ball detection. In: Holz, D., Genter, K., Saad, M., von Stryk, O. (eds.) RoboCup 2018: Robot World Cup XXII. pp. 150–161. Springer International Publishing, Cham (2019)
7. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**, 2171–2175 (jul 2012)

8. Hold-Geoffroy, Y., Gagnon, O., Parizeau, M.: Once you scoop, no need to fork. In: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment. p. 60. ACM (2014)
9. Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I.: Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118 (2018)
10. Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollár, P.: Focal loss for dense object detection. In: Proceedings of the IEEE international conference on computer vision. pp. 2980–2988 (2017)
11. Mitchell, M.: An introduction to genetic algorithms. MIT press (1998)
12. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), https://arxiv.org/abs/1912.01703
13. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 779–788 (2016)
14. Ren, P., Xiao, Y., Chang, X., Huang, P.Y., Li, Z., Chen, X., Wang, X.: A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions. arXiv e-prints arXiv:2006.02903 (Jun 2020)
15. Team, R.: Ray 1.0 architecture (2020), https://docs.google.com/document/d/1lAy0Owi-vPz2jEqBSaHNQcy2IBSDEHyXNOQZlGuj93c/preview
16. Teimouri, M., Delavaran, M.H., Rezaei, M.: A real-time ball detection approach using convolutional neural networks. In: Chalup, S., Niemueller, T., Suthakorn, J., Williams, M.A. (eds.) RoboCup 2019: Robot World Cup XXIII. pp. 323–336. Springer International Publishing, Cham (2019)
17. Zhao, Z., Zheng, P., Xu, S., Wu, X.: Object detection with deep learning: A review. CoRR **abs/1807.05511** (2018), http://arxiv.org/abs/1807.05511
18. Zhengjiang, S.: Gaft - a genetic algorithm framework in python (2018), https://github.com/PytLab/gaft