Evolutionary approach for neural network based agents applied on time series data in the Cloud

István Pintye
Institute for Computer Science and Control (ELKH SZTAKI),
Budapest, Hungary
istvan.pintye@sztaki.hu

József Kovács
Institute for Computer Science and Control (ELKH SZTAKI),
Budapest, Hungary
jozsef.kovacs@sztaki.hu

Róbert Lovas
Institute for Computer Science and Control (ELKH SZTAKI),
Budapest, Hungary
robert.lovas@sztaki.hu

2022 IEEE 16th International Symposium on Applied Computational Intelligence and Informatics (SACI) | 978-1-6654-8125-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/SACI55618.2022.9919475

***Abstract*—One of the alternatives and most popular heuristics for global optimisation and for training artificial neural networks is the application of evolutionary searching method. This paper investigates the internal details of the evolutionary searching mechanism from theoretical aspects and also from experimental parallel execution aspects utilising commercial (AWS) and community (ELKH) cloud resources. The paper introduces and evaluates a procedure where the resources assigned for the execution of calculations are scalable horizontally in order to increase significantly the speed of processing.**

***Index Terms*—neural network, evolutionary algorithm, time series, parallel-trained agent, decision making, optimization, distributed data driven training, evolution, cloud computing, virtual machines**

I. Introduction

These days artificial intelligence as a research topic and its application area reserves a significant amount of research capacity in the world due to its promise to make humans life easier, safer and happier. Artificial intelligence tends to appear in most of the areas in our life such as work, traveling, medical diagnosis and so on.

One of the most widespread way of creating such artificial intelligence is to apply neural networks that is able to adapt to any circumstances once it has been properly trained. As such, constructing and training neural networks are the key to providing the representation of artificial intelligence.

This research area has significant results in the recent period on various machine learning techniques. During the machine learning the main aim is to find the optimal weights for the connections among the neurons to make the network fit to a certain data set representing the inputs and outputs among which the mathematical relationship must be determined.

In this paper we focus on the evolutionary artificial neural networks which provides a solution for training an artificial neural network with a procedure based on genetic algorithms to find the relevant characteristics (such as weights) using evolutionary searching.

During the evolutionary search for neural networks, the fully connected neural network is fed by a given dataset together with the values for the weights. Based on the output of the neural network an agent makes a decision. Multiple executions are required in order to find the best weights for the optimal decision. To perform multiple executions, a set of different values for the weights as inputs are generated for which the same neural networks using the same input dataset will generate a set of outputs during the independent set of executions. We try to maximize in this task the fitness score of the Feed-Forward Fully Connected Neural Network driven agent.

The goal is to maximise the quality of this aggregated decision. The evolutionary search of these weight values performs the iterative calculation of these neural networks with a different set of weights where in each iteration the next set of weight values are determined based on the performance of the previous set. The keys to find the optimal weights are the logic of how the weights are evolved from iteration to iteration, how many iterations and how many set of weights can be processed.

There are several proposals on how to realise/perform the calculation of the next set of weights i.e. on how to evolve the weight parameters during the iterations in order to reach the best final results during the convergence. The speed of the entire search is significantly influenced by the quality/goodness of the evolutionary algorithm applied for calculating the next set of weight parameters. The execution of the neural network on the given dataset using a given set of weight parameters can also be a time-consuming task in certain cases. Finally, performing a full iteration based on the individual calculations of the neural network also consumes significant time during the processing since the number of calculations within an iteration can be a critical factor.

In this paper, our focus is on exploring the possibilities to make the entire evolutionary searching faster by applying different techniques to speed up the computation required for one iteration in which numerous execution must be performed to calculate the output of the neural networks based on its input data and the set of weight parameters.

Especially, the main direction within speeding up the calculation is the proper organisation of executing the different (set of) predictions performed by the neural networks including the usage of cloud resources and parallelization of executions.

With the appearance of cloud computing the access to a large number of distributed resources became easier than before. Clouds especially commercial ones provide a virtually infinite set of resources on a pay-as-you-go basis. These re-

sources can be instantiated to execute any type of calculations and they are ideal for performing parallel executions on the individual resources represented by virtual machines.

The parallelization of metaheuristics methods has received much attention to reduce the run time for solving large-scale problems [1]. In connection with this we would like to show a possible solution for parallelising the genetic algorithm on Cloud. In the next sections we are going to introduce our results on experimenting with cloud resource utilisation for the evolutionary search problem detailed above.

The paper is organised as follows. Section II details the theoretical background of the evolutionary search algorithm as well as the neural network construction used in this paper. Section III introduces the workflow containing the logical steps to perform the calculation. Section IV is about the implementation of the architecture running on cloud and then an evaluation of the results and experiences is described. Finally, Section VI closes the paper with conclusion and future work.

II. Theoretical background

Let us suppose there is a timeseries dataset containing millions of observations. Let us create input variables from this dataset where X_t denote the current observation and X_{t-1} denote its first lag. Construct a vector from that timeseries where each element of the vector is the n^{th} lag of X.

Finally, let the input variable be X and let us find a function $f(X, w)$ which outputs y. Based on the value of y, decision must be made in order to maximise the cumulative result.

Our goal is to find the best w which maximizes the gain (or reward) of the neural network with the given parameters. Since it is unknown which decision leads to the best result, it is not possible to find the best w in a supervised fashion.

Another challenge is that function f is also unknown. In order to come across this problem we use a 2 layered deep feed-forward neural network as a universal approximator. [2] [3] [4]

The neural network will act as function f to calculate the $f(X, w)$. In order to find the proper w, multiple calculation of function f is needed, where the input w is different for each calculation. The set of w forms the population.

To find the best w, the result of the multiple calculations of f is performed and the result is evaluated. For this purpose a fitness function is introduced which takes the y as input. The result of the fitness function is the fitness value representing the quality of the solution. The better the fitness value the better the solution. Selection of the best individuals based on their quality is applied to generate the next generation.

Usually the higher quality individual has higher probability of being selected to qualify itself to the next generation. But we keep only the best one, in other words there is no selection rate at all, only one survives. By just keeping the best individual, it is expected to get a better quality offspring than its parents. We drop the bad individuals from generating more bad individuals. By keeping only the best individual, there will be higher chances to just keep good properties of the individual and leave out bad ones.

The best individual then will be cloned n times and mutation also applied on them. Later we will discuss about the mutation mechanism which has a high impact on the evolution. There are several other methods that exist to select, mutate and encode the properties of the individuals as introduced in [5].

In the next few paragraphs, we further detail the architecture of the neural network. The input of the neural network is $x \in \mathbb{R}^{m \times d}$ where m represents all the cases, d is the number of input parameters (in our case $d = 30$). The weights and bias values of the hidden layers can be defined as follows: $W_h \in \mathbb{R}^{d \times u}, b_h \in \mathbb{R}^u$ where u is the number of neurons in the hidden layer, W_h is the weight matrix of a given layer and b_h is the bias of a given layer. The output of the neural network can be defined by $W_o \in \mathbb{R}^{u \times v}, b_o \in \mathbb{R}^v$, where v is the number of neurons of the output layer i.e. one in our case.

In a feed forward neural network of such neurons, the inputs x_i are the outputs of the neurons in the preceding layer. Furthermore, an Multi Layer Perceptrons is fully-connected, which means that each neuron receives a weighted sum of the outputs of all neurons in the previous layer. A hyperbolic tangent activation function is applied on the output layer, which forms its inputs into the [-1, +1] range. Multi Layer Perceptron, despite its quite complex mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\hat{x} \in \mathbb{R}^n \rightarrow \hat{y} \in \mathbb{R}^m$.

The initialization of the weight values for the neural network are done by drawing values from a standard normal distribution when X is normally distributed with mean $\mu = 0$ and standard deviation $\sigma = 1$:

$$X \sim \mathcal{N}(\mu, \sigma^2) \tag{1}$$

The output of the neural network is finally fed into a simple decision. The rule of decision is specified in the following way:

$$A(y) = \begin{cases} A = SellAction & \text{if } y < 0; \\ A = BuyAction & \text{if } y > 0. \end{cases} \tag{2}$$

This means that whenever y is positive the value of Agent opens a Buy position at the given price otherwise Agent closes the opened Buy position if exists at the given price. The agent's gain or return is the difference between the sell price and the buy price. The agent's overall fitness function is the sum of the total gains of the Agent. After each agent finishes with the task we immediately get the candidate for the parent of the next population. The selection here takes the highest fitness score among the Agents' fitness scores.

Once the candidate parent has been selected, we use the corresponding weight values and generate the new set of weight values for the next population with mutation. The next iteration starts again and we repeat the process until the maximum iteration number reaches a predefined value.

III. Concept of the solution

The method that we are going to demonstrate is applicable in every case where time series data or historical data is avail-

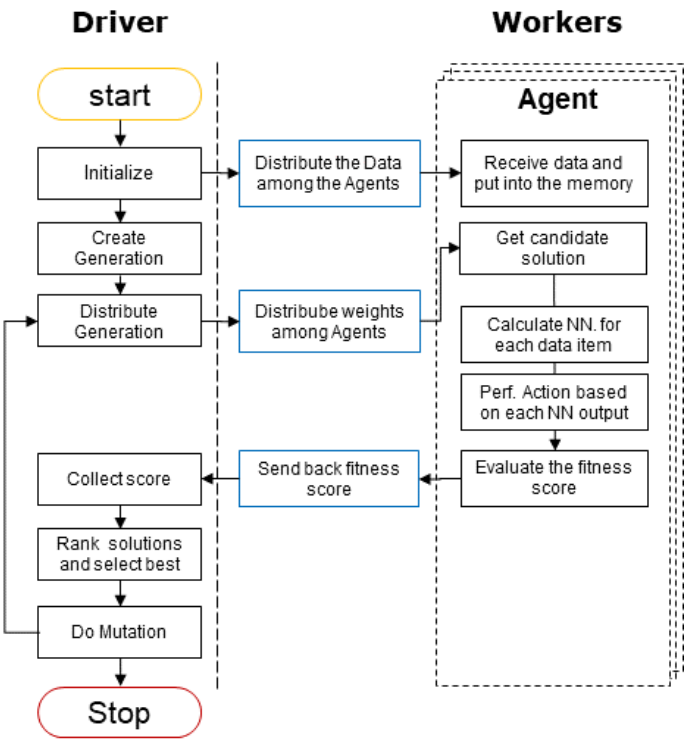

Fig. 1. Workflow of the evolutionary search computation

able and exists. It can come from sensors, measurements, log files or any other observation where time is the main domain. Because the agent which has to solve some problem is trained on this data and not trained in a simulation environment.

We evolve the weights of a given neural network while measuring the goodness of the current solution. The agents run their own simulation/computation on the same, replicated time series data. The replicated time series data are distributed among the agents and each agent holds an exact copy of the original data. Even though in our case this time series data is quite large, it does not necessarily mean that this can cause a computation overhead, because the data has to be distributed only once. In this concept we distinguish two different components: Driver and Distributed Agents (see Figure 1) where Agents are running on different Virtual Machines independently. This distinction is not just theoretical but driver program and Agents run on different physical devices which communicate each other via the network.

Driver component coordinates the distribution of weights and the collection of the results from the Agents. Moreover it performs the comparison, selection, mutation of the weight values. Meanwhile, the Agent or Worker component is responsible to perform, execute the current solution and measure the performance of the given solution.

This concept is based on the generation of multiple instances of weights to form a population and on the execution of the agents in a parallel way to speed up the evolutionary searching. For that purpose we define the resource, called Worker which is responsible for executing one instance of the Agents. By providing multiple Workers, the number of Agents can be scalable as the number of workers is increasing or decreasing.

There are similar distributed solutions have been outlined such as DEAP [6] and Apache Spark [7] [8], however we are going to follow a slightly different approach. In order to understand our concept and the operation of the aformentioned components, a detailed explanation of Figure 1 is given in the next few paragraphs.

In Figure 1, there are three logical columns. The left one shows the logical steps of the Driver component, the middle column details the iteractions, while the right column contains the logical steps of the Agents running on multiple workers.

When the entire evolutionary search starts, the Driver starts with initialisation where the workers are informed about the neural network as well as the data set. In this concept, the architecture of the neural network is not changed in contrast with other Neuro Evolution program like NEAT [9]. After initialisation the Driver creates the first generation of the weight values where each one is considered as a solution. The first generation is created with random weights. The distribute generation step performs the notification of each agent about a candidate solution to be used for applying the neural network with given weights on the data set. Here we continue with the logical steps of the agent component.

The Agent receives the candidate solution and then the calculation of the neural network begins. For this computation the input data which are constituted from the time series data are fed into the the neural network which calculates the corresponding output with the given weights. This step is calculated for every time points in the given time series data set. In our case this calculation might take a while from 10 seconds to 1 minute depending on the size of the data and the complexity of the given neural network. We could achieve a reasonable speed-up with the distribution of this computation task among the workers and executed them simultaneously. We could do this because these computations do not depend on each other. The next step is to perform the actions based on each output generated by the neural network calculation. The action here is the one defined in Section II by Equation 2. The result of this step is a value of [-1,1] for each data item considered as action values. Finally, the agent evaluates these actions using a fitness function that returns a numeric value representing how good the given solution is. The higher the fitness value the higher the quality the solution is. This value is considered as the score which is then reported back to the Driver component.

The driver is waiting for the reports from the Agents once the distribution has happened and the Agents are working in parallel. It continues its execution only when for each of the weight instances the score value has been reported by the Agents. The report contains the fitness score value which then is ranked in the next step. The driver ranks and selects the best candidate that will be considered as the parent of the next generation. The next generation of the candidate solution is created from the parent by mutation. The individual candidates of the new generation are distributed among the

agents/workers again.

When the fitness value reaches a predefined level the iteration is terminated and the best solution is always stored in the driver. Finally, the best solution is tested against unseen test data to validate the model.

Parallelism can help improve both computational time and the number of iterations for convergence. In the literature, different parallel models can be found [1] where one of the most popular is the master-slave model.

In evolutionary computation, there is a method that optimizes a problem in an iterative way to improve a candidate solution concerning a given measure of quality. Such methods are commonly known as metaheuristics as they make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, EA does not guarantee that an optimal solution is ever found. It is used for multidimensional real-valued functions but does not use the gradient of the problem being optimized, which means EA does not require the optimization problem to be differentiable, as is required by classic optimization methods such as gradient descent [10] [11].

IV. Architecture and Operation of the Implemented Algorithm

In order to realise the workflow shown in Figure 1 the program components have been implemented according to the architecture detailed in Figure 2. The driver is responsible for coordinating the computation, distributing the parameters to the agents to execute the simulation based on them, calculating the next set of parameters based on the results arriving from the agents/workers. On the right side (in Figure 2 there are virtual machines denoted as workers executing one agent component. The agent components receive the input parameters, perform the computation i.e. runs a simulation and returns with the fitness score.

In our examination we kept the architecture of the tested neural network fixed. We tried to find only the best weights of the given neural network for the given task. A multilayer perceptron was used with 2 hidden layers with 30 on the first and 15 neurons on the second hidden layer. The number of the inputs was 30. The weights of the neural net are held in a float64.numpy.ndarray. The entire neural net weights were stored in 4240 bytes when they are serialized into JSON to be sent to one of the agents.

The input data set that must be processed by the agents contains the aforementioned 5.6 million items is altogether 780 MB. Currently, this input data set is stored on each of the virtual machines to be used by the agents. In our current implementation, the input data is loaded into the memory to save time and to improve performance.

Because each worker uses the exact copy of the original time series data, the calculation time for each of the agents within an iteration is almost the same, only very small deviations may happen. This is important for the efficiency of parallelisms, as the next iteration may only start when each worker has finished their calculation. Regarding the number of agents,

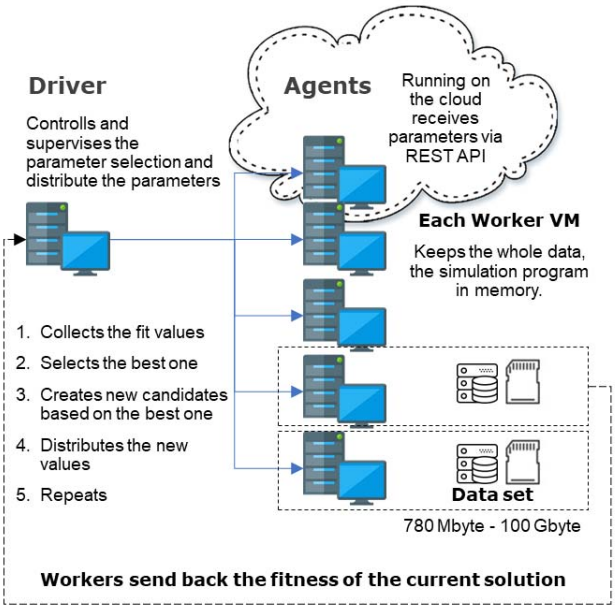

Fig. 2. Architecture and operation of the implementation

the maximum efficiency can be reached when the number of agents is equivalent to the size of the population the driver generates for an iteration. Following this rule, all the calculations will be performed in a parallel way without loss of time and delay.

Both, the driver and the agent components are implemented in Python using Flask for implementing the REST API. The driver is using the REST API of the agent to initiate the next round of calculation by sending the weights. The agent is using the REST API of the driver to report the results back. Since the execution may take an unpredictably long period, we considered this communication pattern better compared to blocked, synchronized remote invocation of the agent component to avoid the driver to be blocked for a certain period.

The agent components can be scaled by launching new workers i.e. virtual machines in the cloud. We do not feel the need for scaling dynamically during operation, we prefer a fixed number of workers that fits the population size as mentioned above. Scaling is better be performed before the entire evolutionary search starts. When adding a new agent component, a new virtual machine is spawned. The agents are deployed based on contextualisation of the virtual machines using cloud-init script to realise the steps of the deployment. The input data is stored on a separate volume which is mounted to each virtual machine executing an agent component under a predefined directory path.

V. Results and Experiences

In order to evaluate the efficiency of the Cloud based parallel implementation of the neural network weight selection algorithm (nnws), different experiments have been carried out.

Its behavior in terms of convergence and total execution time has been compared to the sequential implementation.

For the experimentation, the application was deployed with default settings in the AWS public cloud using virtual clusters formed by 10 and 20 nodes linked by the AWS standard network (Ethernet 1GB). For the nodes the t3a.large instance (2 vCPU, 7.75GB RAM) was selected. Each experiment was executed 5 times on every virtual cluster and the average and standard deviation of the execution time were measured and collected. Comparison of the sequential and parallel execution is detailed in Table V.

TABLE I
COMPARISON OF DIFFERENT EXECUTIONS

	VM	Pop size	Avr. exec. time min.	Fitness score
Sequential	1	10	$382 \pm (5.6)^{a}$	2.31
Parallel	10	10	$45 \pm (4.0)^{a}$	2.31
Sequential	1	20	$798 \pm (10.1)^{a}$	2.93
Parallel	20	20	$49 \pm (5.1)^{a}$	2.93

[a] 1 std. deviation is in the bracket.

We investigated two types of execution (seq, parallel) with two population size (10,20) as it can be seen on Table V. The number of generation was set to 5000. While the sequential setup takes twice as much time when the population size was doubled, the parallel solution takes only a little longer in the case of population size 20 compared to the execution for the population size 10. The small increase may come from the network communication overhead of the parallel solution. For the parallel cases, we set the number of workers equal to the number of the population size to maximize the parallelism.

Based on our measurements which is not part of Table V we have seen that the network communication overhead takes roughly 1/20 less time than the computation of the fitness function. But this ratio strongly depends on the capability of the network and the amount of the processed data that resides on the workers as well.

We also made a couple of experiments shown in Figure 3 with the mutation rate, the population size in the following order. First, we examined whether keeping the best solution in the new population leads to a better solution or not. The result showed that holding the best solution from the previous generation in the new generation produces a faster convergence in the fitness score. We also noticed that many of the candidate solutions lead to a worse fitness score which may be caused by the relatively high mutation rate, or the complexity of the neural network where small modifications of the weights can cause a huge difference in the result.

The effect of the population size was also examined as Figure 4 shows. On the Y axis the fitness score (left the best fitness score in the given experiment respectively, right all fitness score in the given experiment). Experiments are indicated by color. Negative fitness score can also occur. The result revealed that the bigger population size does not necessarily lead to a better fitness score. To make the results comparable when the size of the population increased two

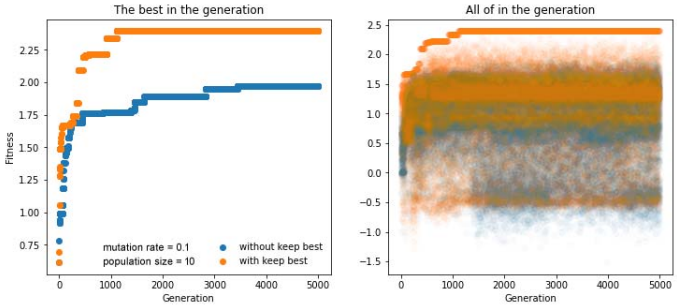

Fig. 3. Effect of keeping the best candidate for population generation

times bigger, the number of iterations has been decreased to its half. For example, when the population increased from 10 to 20, the number of iterations decreased from 2500 to 1250. It is important to note that 10 simultaneous, individual runs with 2500 generation took half of the time compared to the 5 individual runs with 5000 generation, however, it reached a better fitness score.

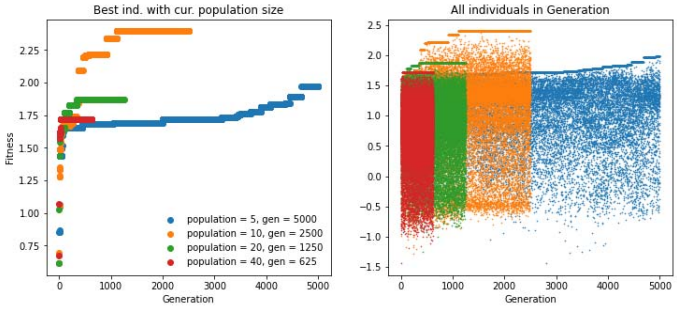

Fig. 4. Results for 4 variations of population size and generation size

We also made experiments where we varied the mutation rate while we kept the population size at the same level. The results are shown in Figure 5 This experiment was carried out only with a population size of 10. Detailed results show that With a higher mutation rate the convergence of the fitness score was faster but after a while, it was not able to find proper weights.

The subtle differences between the bad and the good weight modification may indicate that adaptive mutation rate is needed [12]. With a lower mutation rate the convergence was slower and could not reach a good solution, but its fitness score increases gradually from generation to generation.

For the reason of space, we can not show our results about the effect of the crossover [13], but we found that crossover between two parents does not lead to a better solution but makes the performance of the offspring even worse. This phenomenon can be caused by the ”permutation problem”, also known as ”the competing convention problem” as several other studies analysed this phenomenon. When two parents are permutations of the same network, then their offspring will have lower fitness. Since the same part of the network is copied in two different locations, the offspring is losing viable

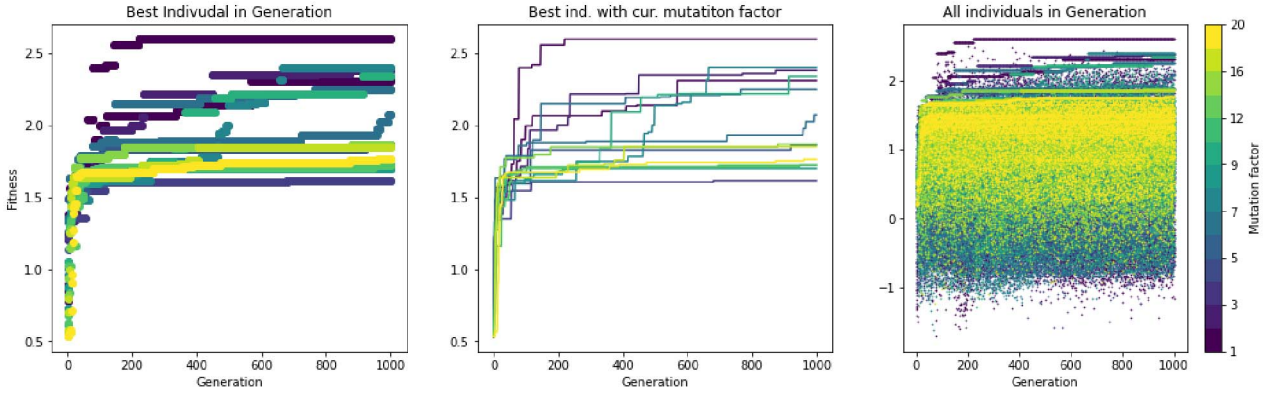

Fig. 5. Effect of the mutation rate for 1000 generations

genes for one of the two locations. Similarity-based crossover can come across this problem as introduced in [13].

VI. Conclusions and Future Direction

In this paper we presented a possible solution to implement EA on Cloud in parallel fashion using Python a REST API service to reduce the overall computation time.

On our particular task supervised learning is not possible because the proper labels or the target variable does not exist. In such case the weight of the neural network was found by evolutionary selection.

The implementation successfully introduced the utilisation of cloud resources efficiently to speed up the computation of the neural networks. The paper detailed the theoretical background as well as the high-level abstract workflow of the complete solution. Then the solution has been implemented and finally, the results and experiences have been summarised.

One of the future directions is to investigate the possibility to further parallelise the computation for one agent. Currently, the biggest performance is needed to run the agent to calculate the neural network for each of the data items of the dataset. We are going to investigate data-parallel execution for example.

The success of a neural network at a particular application is often determined by a series of choices made at the start of the research, including what type of network to use, what is the method and the architecture to train it. These choices - known as hyperparameters - are chosen through experience, random search or computationally intensive search processes. In our future work we also would like to implement the distributed hyperparameter search in order to improve the performance of the agent. Similar hyperparameter search techniques exist [14] such as Population Based Training.

VII. Acknowledgement

On behalf of the research presented in this paper we thank for the usage of ELKH Cloud (https://science-cloud.hu/) that significantly helped us achieving the results published in this paper. The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program.

The authors thankfully acknowledge the support of the Doctoral School of Applied Informatics and Applied Mathematics, Obuda University. The presented work was supported by the UNKP-21-5 New National Excellence Program of the Ministry For Innovation And Technology from the source of the National, Research, Development and Innovation Fund.

The work was supported in part by the Eötvös Loránd Research Network Secretariat (ELKH KÖ-37/2021).

References

[1] Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research 20(1) (2013) 1–48

[2] Franco Scarselli, Ah Chung Tsoi, Universal Approximation Using Feedforward Neural Networks: A Survey of Some Existing Methods, and Some New Results, Neural Networks, Volume 11, Issue 1, 1998, Pages 15-37.

[3] Hornik, Kurt; Tinchcombe, Maxwell; White, Halbert (1989). Multilayer Feedforward Networks are Universal Approximators (PDF). Neural Networks. Vol. 2. Pergamon Press. pp. 359–366.

[4] Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function Mathematics of Control, Signals, and Systems, 2(4), 303–314.

[5] Eiben, A. & Smith, Jim. (2003). Introduction To Evolutionary Computing.

[6] Fortin, Félix-Antoine & De Rainville, François-Michel & Gardner, M.A. & Parizeau, Marc & Gagné, Christian. (2012). DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research, Machine Learning Open Source Software. 13. 2171-2175.

[7] Maqbool, Fahad & Razzaq, Saad & Lehmann, Jens & Jabeen, Hajira. (2019). Scalable Distributed Genetic Algorithm Using Apache Spark (S-GA). 10.1007/978-3-030-26763-6_41.

[8] Teijeiro, Diego & Pardo, Xoán & González, Patricia & Banga, Julio & Doallo, Ramón. (2016). Implementing Parallel Differential Evolution on Spark. 9598. 75-90.

[9] Floreano, Dario, Peter Dürr, and Claudio Mattiussi. "Neuroevolution: from architectures to learning." Evolutionary intelligence 1.1 (2008): 47-62.

[10] Rocca, P.; Oliveri, G.; Massa, A. (2011). "Differential Evolution as Applied to Electromagnetics". IEEE Antennas and Propagation Magazine. 53 (1): 38–49.

[11] Storn, R.; Price, K. (1997). "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces". Journal of Global Optimization. 11 (4): 341–359.

[12] Marsili Libelli, S., Alba, P. Adaptive mutation in genetic algorithms. Soft Computing 4, 76–80 (2000).

[13] Mauro Dragoni, Antonia Azzini, Andrea G.B. Tettamanzi, SimBa: A novel similarity-based crossover for neuro-evolution, Neurocomputing, Volume 130, 2014, Pages 108-122

[14] Jaderberg, Max, et al. "Population based training of neural networks." arXiv preprint, arXiv:1711.09846 (2017)