A GPU based distributed algorithm for HyperNEAT

Emad Hosseini [*] Ali Kamandi

October 11, 2019

1 Introduction

The advancements in the field AI in the past few decades has led to it being one of the tools of our everyday life. Yet there are some great challenges in developing and training AI systems. There are some tasks that today's AI systems are particularly more capable of doing. That includes image classification [1], natural language processing [2], motor control [3] and many other fields that previously seemed impossible for machines to do and were considered specific to humans and animals.

But what made these traits possible is the advancements in computers and hardware that made faster processors and larger memories and with the help of more data, scientists could build working AI software. Yet training each model that is capable of a narrow field of tasks takes thousands of hours of CPU time in huge clusters and on sometimes petabytes of data [4]

Having more data and enough time to process that data is not something to come by easily and in many cases like autonomous navigation in unknown environments or critical decision making for self driving vehicles lack either the data or the time to train conventional neural network models. There are also other challenges including vanishing and exploding gradients [5], optimum network structure and other known issues of back propagation that are known to scientists.

One group of models that tackles these challenges are neuroevolution models. That is defined by Gomez and Miikkulainen in 1999 as "systems that evolve neural networks using genetic algorithms" [6] and genetic algorithm has some features that makes it ideal for such tasks as training a neural network. These features include no assumptions about the search space and its derivatives, high capability of parallel processing and incremental complexity.

Therefor many research is done in actual models that implement neuroevolution including early works that we mentioned before ([6]), probabilistic models like the one used by Mao et al. [7], NEAT [8] and many others.

Full utilization of any GA means using its distributable capacities and that is the target of this paper.

In this paper we will introduce a computation method for a specific type of NE model i.e. NEAT using general purpose computers. In the next part we see what has already been done in this field and after that in section 3 our computational method is presented. Some benchmarks are done and results are shown in section 4 with conclusions that follows.

2 Background

Neuroevolution of augmented topologies (NEAT) is one of the most successful models of neuroevolution introduced in 2002 by Stanley and Miikkulainen [8] in this model there is an encoding of the neural network in a genotype that considers an innovation number for each newly formed weight. Also different complexity networks are kept separate using a mechanism of speciation to allow each of them excel in their own rival group and avoid new and unfit individuals get consumed by the older and more mature ones.

A population of such genomes and their respective

[*]University of Tehran, Department of Algorithms and computation.

phenotypes is then created. Using genetic algorithm this population is then directs towards better networks that work better to solve the problem at hand.

NEAT is really successful in finding minimal networks for many tasks that are simple enough but when the requirement of the task is more than that, the search space gets too big and NEAT is inefficient in finding the best networks. This was addressed in another model called HyperNEAT [9] that uses underlying symmetries in tasks through a substrate that is essentially a raw initial network.

This substrate is then filled with the connections that are themselves products of another smaller network trained through NEAT algorithm. For example an object detection task would consider the rotation of the target object as a symmetry therefore having a circular substrate leads to automatic consideration of the required symmetry in the task.

The nature of GA involves many simple individuals controlled by an environment that produces the survival of the fittest mechanism for a certain goal. This seems an ideal task for a distributed system. And some researches have already exploited this feature.

for example Such et al. compared using a parallel GPU based and distributed CPU based neuroevolution against other methods of training the network like Q-learning (DQN) and policy gradients (A3C). [10]

Also the fact that GA can utilize GPU and run more efficient on a distributed systems is not new and many existing research in this field is gathered by Cheng and Gen in their recent review of the field. [11]

Using the same methods for getting better results in the HyperNEAT is the target of this paper. There are two main steps in distributing the task of any GA based algorithm the first and easy part is distributing the individuals (which is very important in the case of HyperNEAT as explained in section 3) and the next step is to distribute the control unit that is often called the environment. This part involves each of the separate individuals of the population in the task of finding the fittest and crossover the parents to create child genome replacing them with the less fit individuals.

3 Distributed HyperNEAT

There are a few words that are used in this paper that we should clarify their meaning before we delve into the algorithm. Every neural network problem has vector of real numbers as it's *input* and another vector is generated as *output*. In HyperNEAT there is an empty network that consists only of nodes with no connections which is called *substrate*. The nodes in the substrate have coordinates and it is defined by the expert considering the symmetries and other features of the problem space as defined by Stanley et al. [9]

Generally in the literature the genetic algorithm of NEAT is applied on small neural networks with the task of creating best connections for the substrate but in this paper, for the reason which we will discuss later, an *individual* of the genetic algorithm consists of its own copy of the substrate plus, a NEAT genome.

Each individual in this model is capable of generating output based on an input by creating the genotype of NEAT network filling its substrate and running the input vector through the newly generated ANN.

A collection of individuals create a *population* that runs for many *generation*s. Each generation is done by calculating the performance measure for each individual and replacing the unfit ones with the new offsprings of the fitter ones.

*Offspring*s of a generation is the result of *crossover* and *mutation* of existing individuals, the details of which will be addressed later on.

With this terminology we can write the normal HyperNEAT algorithm as in algorithm 1.

We don't go into details of some part of the algorithm here because it's outside the scope of this paper. But one part that interests us is shown in algorithm 2

In algorithm 2 w_{ij} is the weight between node i and j, $Substrate_{Input}$ is the nodes in the input layer of substrate and likewise, $Substrate_{Output}$ and $Substrate_{Node}$ are respectively output layer nodes and nodes with connection to specified node.

We call this organization of HyperNEAT algorithm as "CPU version" from now on but what we actually

Algorithm 1 HyperNEAT Algorithm

```
1:  procedure Train HyperNEAT($Inputs$,$Outputs$)
2:      $Individuals \leftarrow CreateInitialPopulation()$
3:      for $GenerationCounts$ do
4:          for $Individual \in Individuals$ do
5:              $TotalError \leftarrow 0$
6:              for $i \leftarrow 1, |Inputs|$ do
7:                  $Actual \leftarrow GetOutput(Individual, Inputs[i])$
8:                  $Error \leftarrow Actual - Outputs[i]$
9:                  $TotalError \leftarrow TotalError + Error^2$
10:             end for
11:             $Error_{Individual} \leftarrow TotalError$
12:         end for
13:         $EvictHighErrors()$
14:         $CreateNewChildren()$
15:     end for
16:     return $MinErrorIndividual$
17: end procedure
```

mean by that is that this runs serially as opposed to "GPU version" which will distribute the tasks.

The process in algorithm 1 consists of three parts. The first part is creation of the initial population which is of $\mathcal{O}(n)$ where n is size of the population. Because creating each individual takes constant time for each of the NEAT genomes as each one contains exactly $2d$ randomized weights where d is the number of dimensions specifying coordinated of a node in the substrate. For example a 2D grid substrate will have a 2D coordinate position for each node and the substrate would have 4 input nodes taking in 2 nodes and giving back the weight between them.

The third part of the algorithm 1 is the eviction and recreation of next generation. The eviction percentage of the total population is another parameter of the algorithm. Creating a next generation individual from existing ones is not deterministic and involves two stochastic steps of cross-over and mutation. We will not go through this analysis in this paper and generally the third part is $\mathcal{O}(geE_c)$ where g is the number of generations, e is number of evictions and E_c is the expected value of time requirement of the creation of a new individual with cross-over and mutation operations.

The second part of algorithm 1 is the actual training generations which we will focus on. The time complexity of this part of the algorithm 1 is heavily dependant on the time complexity of algorithm 2 which is not deterministic due to the stochastic nature of the NEAT algorithm.

Algorithm 2 Calculate output for each individual

```
1:  procedure GetOutput($Individual$,$Input$)
2:      $NEAT = CreatePhenotype(Genome_{Individual})$
3:      for $Node_i \in Substrate$ do
4:          for $Node_j \neq Node_i \in Substrate$ do
5:              $w_{ij} = GetOutput_{NEAT}(Node_i, Node_j)$
6:              if $w_{ij} < Threshold$ then
7:                  $w_{ij} = 0$
8:              end if
9:          end for
10:     end for
11:     for $Node \in Substrate_{Input}$ do
12:         $Value_{Node} \leftarrow Input_i$
13:     end for
14:     for $Node \in Substrate_{Output}$ do
15:         $GetValueRecursive(w, Node)$
16:     end for
17:     return $Substrate_{Output}$
18: end procedure
19: procedure GetValueRecursive($w$,$Node$)
20:     if $Value_{Node} \neq null$ then
21:         return $Value_{Node}$
22:     end if
23:     $sum \leftarrow 0$
24:     for $ConnectedNode \in Substrate_{Node}$ do
25:         $value \leftarrow GetValueRecursive(ConnectedNode)$
26:         $sum \leftarrow sum + value \times w_{ij}$
27:     end for
28:     $Value_{Node} \leftarrow ApplyActivation(sum)$
29:     return $Value_{Node}$
30: end procedure
```

Let E_I be the unknown expected time complexity of getting the output of a single individual for a single input. Then $\mathcal{O}(gnTE_I)$ will be the time complexity of the second part of algorithm 1 where T denotes the number of training data size.

The total time complexity of the algorithm 1 is shown in equation 1.

$$\mathcal{O}(n + gnTE_I + geE_c) \tag{1}$$

3.1 Thread per Individual

As it is clear in algorithm 1 in previous section, training phase of the HyperNEAT consists of a 3 layer main loop. These loops can be addressed by parallelism to reduce time complexity of the algorithm.

We will first combine the inner two loops in a collection of single operations. Each of these operations will calculate the actual error of a single input for a single individual. This simple process is shown in algorithm 3.

Algorithm 3 Kernel of the parallel mode

```
    $Actual \leftarrow GetOutput(Individual, Inputs[i])$
    $Error \leftarrow Actual - Outputs[i]$
    $TotalError \leftarrow TotalError + Error^2$
```

There are many memory operations done in single machine (i.e. CPU) mode of the algorithm that we ignored for simplicity but, for the sake of completeness we will include the amount of added memory operations to the multi-machine (i.e. GPU) mode. Since the internal operations between the two modes are the same read and write access to memory is the same in both modes. The only added memory operations are the necessary data transfer required for data integrity and reading the results, between the machines in the multi-machine mode.

Another thing worth mentioning here is that there will be a controller (leader) machine in multi-machine mode. In our particular architecture of using a CPU and many processors of the GPU as the cluster machines the CPU will play the role of the leader.

Getting back to algorithm 3, execution requires $n.T$ single operations in general there will be m machines where $m < n$ and since $T > 1 \Rightarrow m < n.T$. So we split the required error value calculation of each generation into chunks of size shown in equation 2.

$$\left\lfloor \frac{n.T}{m} \right\rfloor \tag{2}$$

This way the nT in the middle part of equation 1 will be replaced by the value in equation 2. But the data must be transferred to cluster machines. The required data for algorithm 3 is the individual and the input. The input is fixed in size and it's size is negligible asymptotically compared to the individual. The individual size is in direct order of the time required for it to get a single output and therefore $\mathcal{O}(Individual) = E_I$ which is unknown but by placing these values in equation 1 we will get the time complexity of the algorithm as in equation 3.

$$\mathcal{O}(n+g\frac{nT}{m}E_I+nE_I+geE_c) = \mathcal{O}(n+nE_I(\frac{gT}{m}+1)+geE_c) \tag{3}$$

This is the time reduction for fixed amount of stochastic training that NEAT offers and simply means more training in the same time. But one more implication of this statement is that with this method you can have more individuals in the population which by the mechanism of speciation causes more young species. The speciation mechanism is the counterpart of exploration as oppose to inter-species exploitation of GA algorithm and a good ballance in the two can be achieved in this algorithm where you can have more individuals. [12]

In equation 3 if we increase the number of population, n, and the number of machines, m, since E_I will remain constant, we will see a factor of $1/m$ decrease in time complexity of the algorithm.

3.2 Distribution Challenges

The original NEAT and HyperNEAT algorithms were not designed with distributed environment in mind. Although, as mentioned in previous section, genetic algorithms have a natural tendency to be used in distributed architecture, some challenges arose in the implementation of this particular algorithm for GPU.

First one is the innovation number. Innovation number is a mechanism originally included in NEAT algorithm to be incorporated with each network connection and provide the ability to distinguish similar and different connections even when the nodes and weights of them have mutated over time. [8]

Assigning a number for this innovation number is an easy task in single process architecture but, not in distributed architecture. We should notice that the intentions behind innovation number only requires a unique id and the order of that is not important for the algorithm. Therefore we have assigned innovation numbers not based on a simple increasing number but as a combination of process id and a number generated in each process.

One more thing that plagued our particular implementation is the matter of using device bandwidth in moving data from CPU to GPU. This might not effect the more broad multi-machine usage of this algorithm but, in single machine CPU-GPU implementation that we use moving data to and from GPU has a overhead cost of opening and closing the stream.

To handle this issue we split the operation in two parts first we move all required data to GPU memory and then access it in different threads. This does not have an effect on our complexity analysis but extremely improves performance in wall clock time of algorithm execution due to removal of I/O waits and utilizing device bandwidth.

4 Comparison of Results

In this study we have used nVidia CUDA®library which is well known and widespread in the GP-GPU community. The program is written in C++ in a way that the same code runs on the CPU as in algorithm 1 as well as GPU. The benchmarks will therefore not include implementation differences of the two platforms.

Selected task is also a very simple function approximation task, since the performance of both algorithms will be the same in terms of accuracy and the goal is speed improvement.

The data-set consists of 10,000 randomly generated 2D inputs in the form of $[x, y]$ where $x, y \in \mathbb{R}$ and $0 < x, y < 100$. 80% of this data is used for training and the rest used for test phase. The expected output is z as shown in equation 4. In which ϵ is white noise introduced to make the problem more challenging.

$$z = x^2 + y^2 + \epsilon \quad \epsilon \sim N(0, 1) \tag{4}$$

The value of m is defined by the system and there is little control over it. The test results shown here are generated using a *nVidia GeForce GTX 960m* with 640 CUDA cores and 4GB of memory. Also host computer (which runs the CPU tests) has an *Intell Core i7 6700HQ* processor.

Number of individuals, n, and generations, g however are configuration parameters and can be changed easily. T and e are dictated by the data-set and we will not change them in our analysis for better comparisons.

The chart shown in figure 1 shows how much parallel work has been done. This is the result of 1 generation of $n = 100$ individuals ran in around 6.4 seconds[1]. The same configuration runs around 7.7 seconds in CPU.

If we keep increasing the value of n and track the runtime of the application for one generation, we expect the time to increase with each new individual added to the population but there will be two constant factors. One is the amount CPU works needed for each generation i.e. for eviction, initial creation of individuals, reading training data from file and etc. and the second is the time required to transfer data to and from GPU.

The chart in figure 2 shows the result that we expected. In it, the amount of time required for a single training generation of different sizes of population is shown for both CPU and GPU versions of the algorithm.

At first the GPU transfer time is causing the total time of GPU to be higher than CPU but as the population size increases, this time dissipates and parallel execution of individuals runs around 2 times faster at $n = 500$. The CPU work overhead is constant for both scenarios. Later we will give an idea about how we can distribute that work between different processors as well.

The effect of the transfer overhead can be seen in figure 3. In that the average time of running a single individual is shown for both GPU and CPU models.

[1]The overhead of the profiler tool is included in these measurements.

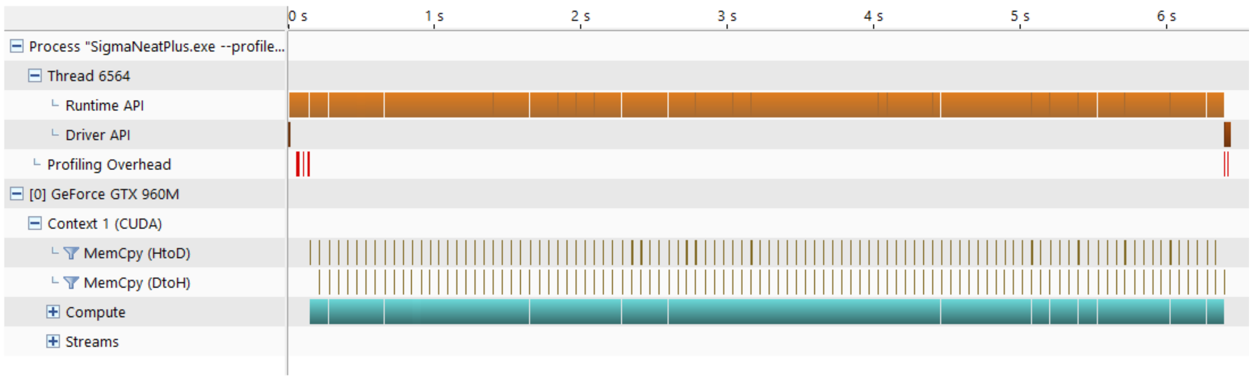

Figure 1: Profiler output of parallel algorithm

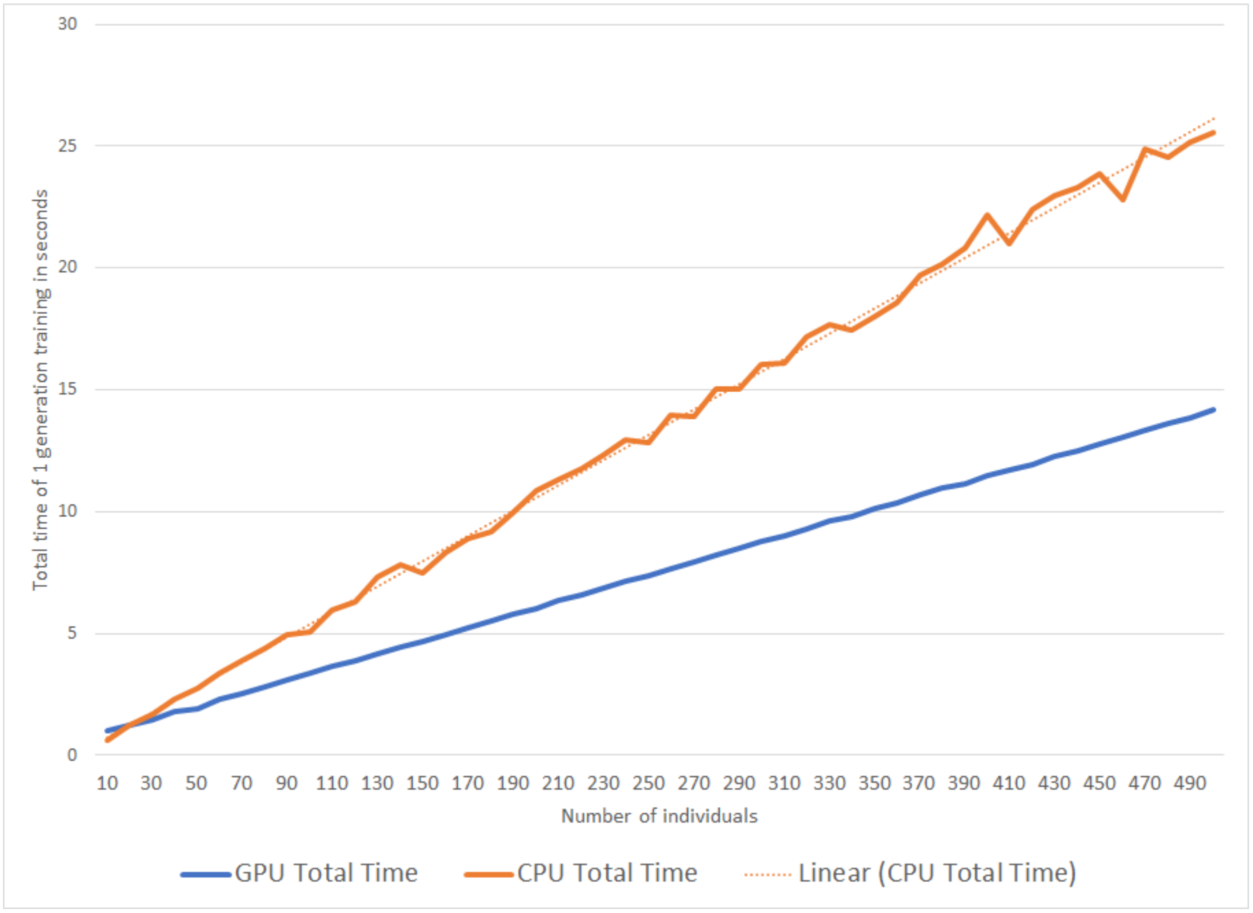

Figure 2: Comparison of total training time of 1 generation of n individuals

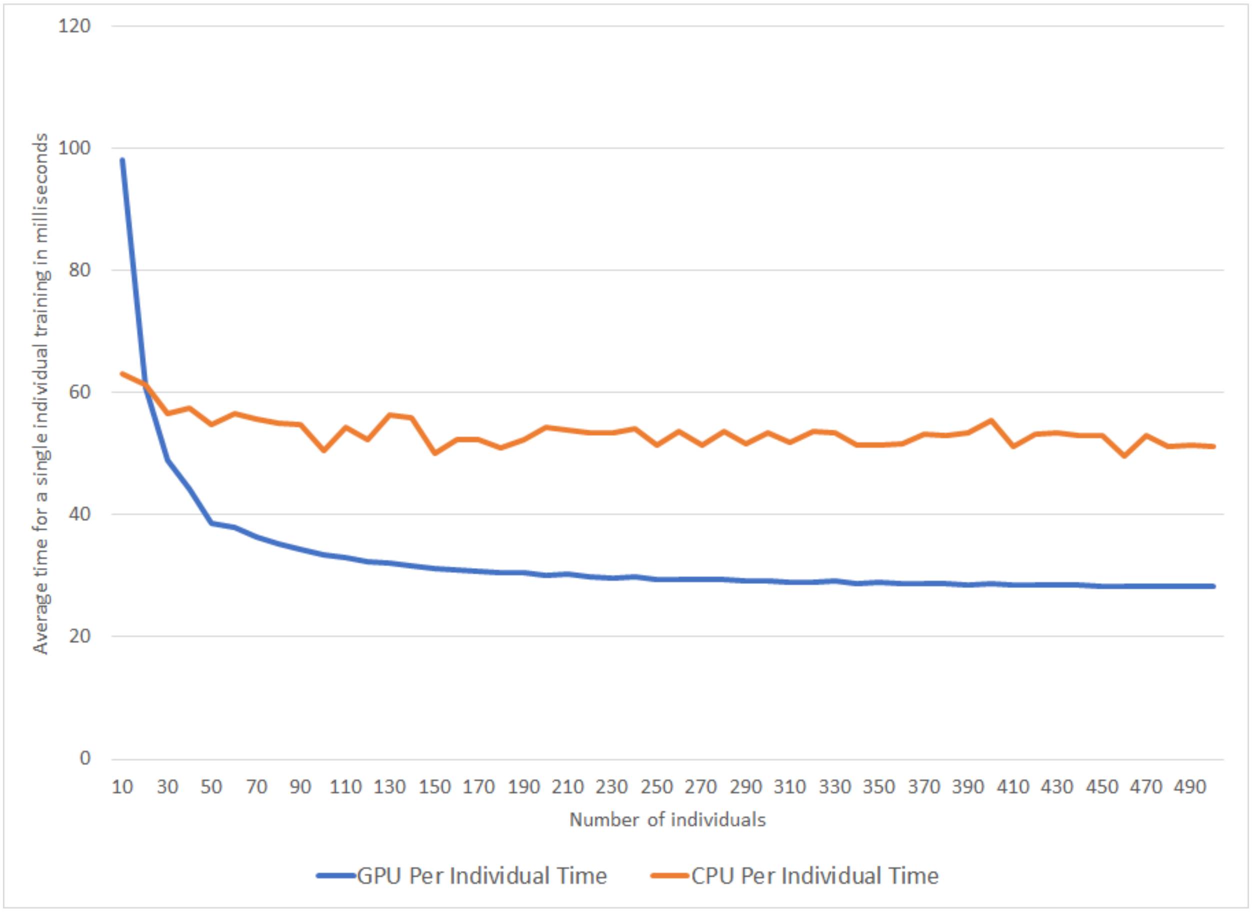

Figure 3: Comparison of average training time of 1 individual for 1 generation

At first in lower population sizes GPU takes around double the time of CPU for calculating the output of each individual then, a sharp drop happens and the situation changes. This drop virtually does not exist in CPU version.

One observation that you might have is the difference between the smoothness of the charts. The smoothness of results for GPU in both figures is duo to the fact that GPU is a collection of processors and that averages the result of each. The performance of each processor may have fluctuations like we see in CPU charts. These results are the average of 3 runs per each scenario.

The results in figure 2 doesn't show the actual difference between the two approaches. Because it ran for only 1 generation and an important part of the algorithms, i.e. the eviction of old and creating new individuals, is left out. If we run the algorithm for longer we will see that, since most of the execution time is used for the second part of algorithm 1 the same result will hold.

For example running the algorithm for 100 generations of 100 individuals for 3 times is done in average 285.530s in GPU and 511.398s in CPU which shows about 56% increase in performance. And the gap will only widen as n and g increase.

5 Conclusion

In this paper we have introduced a parallel mechanism to run an existing neuroevolution algorithm. We have overcome the challenges and ran the algorithm on GPU using nVidia CUDA technology. But as we kept this mindset throughout the paper this algorithm can run on any distributed processor system and gain an even greater performance gain.

The actual goal here was to give a mechanism for scaling the neuroevolution approach for training neural networks. Since most of the research in ANNs is done using back propagation algorithm which scales well but, limits the search space only to smooth and derivable activation functions.

The proposed algorithm showed scalability of the algorithm to almost double the performance of the conventional implementations.

6 Future Studies

The distributed algorithm described in section 3 has the benefit of breaking the main loop. While in theory we should be able to distribute each individual for itself.

This requires a communication model between the individuals that doesn't have a huge impact on the total working time of each of them. Also as mentioned before there are many tasks involved in the population control that needs to be addressed by a central node. This node can certainly use the leader election algorithms like [13] and [14].

This higher level of distribution can be considered as future goals of such development.

References

[1] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.

[2] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[3] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[4] Ananda Samajdar, Parth Mannan, Kartikay Garg, and Tushar Krishna. Genesys: Enabling continuous learning through neural network evolution in hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 855–866. IEEE, 2018.

[5] Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *Advances in Neural Information Processing Systems*, pages 582–591, 2018.

[6] Faustino J Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361, 1999.

[7] Ke Zhi Mao, K-C Tan, and Wee Ser. Probabilistic neural-network structure determination for pattern classification. *IEEE Transactions on neural networks*, 11(4):1009–1016, 2000.

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

[9] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

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

[11] John Runwei Cheng and Mitsuo Gen. Accelerating genetic algorithms with gpu computing: A selective overview. *Computers & Industrial Engineering*, 128:514–525, 2019.

[12] David Beasley, David R Bull, and Ralph Robert Martin. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, 15(2):56–69, 1993.

[13] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

[14] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 990–999. Society for Industrial and Applied Mathematics, 2006.