A GPU based distributed algorithm for HyperNEAT

Emad Hosseini *   Ali Kamandi

October 5, 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 [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 phenotypes is then created. Using genetic algorithm

---
*University of Tehran, Department of Algorithms and computation.

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 big enough that 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 HYPER-NEAT($Inputs$,$Outputs$)
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.

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

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

This way

3.2 Environment Distribution

In this version we lose coordination but gain on not data copy

REMEMBER Random instead of innovation number

4 Comparison of Results

5 Conclusion

GPU is analogous to many machines

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.