Enabling Continuous Learning through Neural Network Evolution in Hardware

Ananda Samajdar
Georgia Institute of Technology
Atlanta, GA
anandsamajdar@gatech.edu

Kartikay Garg
Georgia Institute of Technology
Atlanta, GA
kgarg40@gatech.edu

Tushar Krishna
Georgia Institute of Technology
Atlanta, GA
tushar@ece.gatech.edu

Abstract

In the past few years we have witnessed monumental advancements in artificial intelligence (AI), thanks to the invention of deep learning techniques. However, we are still far away from implementing adaptive general purpose intelligent systems. Supervised learning techniques in the present depend upon large amounts of structured training data and have high computational cost. Moreover coming up with a model requires a lot of manual effort, with experts tweaking hyper-parameters over several iterations of trial and error.

Reinforcement Learning (RL) and Evolutionary Algorithm (EA) based methods circumvent this problem by continuously interacting with the environment and updating the models based on obtained rewards. This closed loop approach provides adaptability and is a promising solution for general purpose AI. However, deploying these algorithms on ubiquitous autonomous agents (robots/drones) demands extremely high energy-efficiency for the following reasons: (i) tight power and energy budgets, (ii) continuous interaction with the environment, (iii) intermittent or no connectivity to the cloud to run heavy-weight processing. This in turn drives the choice of algorithm and the platform of implementation.

We demonstrate that EA offer a tremendous opportunity for parallelism and HW-SW co-design. One can not only parallelize across multiple individuals of a population, but also across *genes* (NN nodes and connections) within an individual, enabling orders of magnitude speedup and energy reduction for performing evolution and thereby learning. We propose a novel hardware accelerator called EVE (EVolutionary Engine) that is optimized for running the evolutionary learning steps (crossovers and mutations). We implement EvE in 15nm Nangate FreePDK and observe that EvE achieves over **five to six orders of magnitude in energy efficiency** over desktop and embedded CPUs and GPUs.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

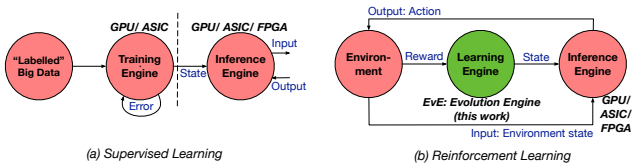


Figure 1. Conceptual view of where EvE fits within learning.

ACM Reference Format:
Ananda Samajdar, Kartikay Garg, and Tushar Krishna. 2018. Enabling Continuous Learning through Neural Network Evolution in Hardware. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Ever since modern computers were invented, the dream of creating an intelligent entity has captivated humanity. We are fortunate to live in an era when, thanks to deep learning, computer programs have paralleled, or in some cases even surpassed human level accuracy in tasks like visual perception or speech synthesis. These advancements are monumental and have transcended all past expectations. So much so, that popular celebrities express concerns over creation of nefarious omnipotent AI. However in reality, despite being equipped with powerful algorithms and computers, we are still far away from realizing general purpose AI.

The problem lies in the fact that the development of supervised learning based solution is mostly open loop (Fig. 1(a)). A typical deep learning model is created by hand-tuning the topology by a team of experts over multiple iterations, often by trial and error. The said topology is then trained over gargantuan amounts of labeled data, often in order of petabytes, over weeks at a time on high end computing clusters. The trained model hence obtained, although remarkable for the task for which its trained, is highly rigid and cannot adapt to any other task. This limits applicability of supervised learning for problems for which structured labeled data is not available, or the nature of the problem is not static.

Reinforcement Learning (RL) is one possible solution for realizing general purpose AI. Algorithms based on RL work by interacting with the environment by a set of actions taken with respect to the given state of the environment, as shown in Fig. 1(b). At the heart of these algorithm is a policy function, which determines which action to be taken. Each interaction with environment generates a reward value, which is an measure of effectiveness of the action for the given problem. The algorithm uses reward values obtained in each

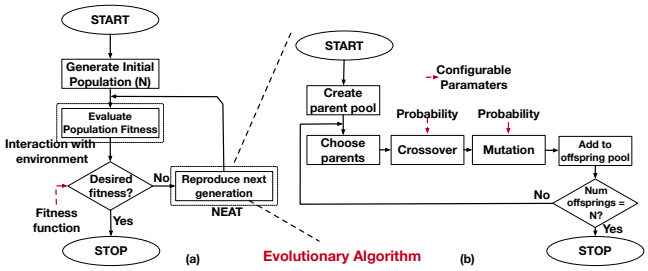


Figure 2. (a) Flow chart depicting the steps in any Evolutionary Algorithm (EA) (b) Expanded view to show the reproduction steps in NEAT

iteration to update the policy function until it converges upon the optimal policy.

There have been limited but extremely promising demonstrations of RL [1, 2] - the most famous being Google DeepMind's supercomputer autonomously learning how to play AlphaGo and beating the human champion [1]. However, moving RL algorithms from supercomputers to robots/drones at the edge introduces tremendous architecture and system-level challenges from a compute, memory and energy-efficiency point-of-view. Continuous interaction with an environment through an embedded energy-constrained device with little/no connectivity to a backend cloud dictates the the choice of algorithm, platform, and implementation. This work attempts to take a first-key step towards this direction.

We focus on Evolutionary algorithms (EA) which are a class of algorithms similar to traditional RL, but computationally more scalable [38]. EAs are blackbox optimization methods and do not have the means to capture the nature of the environment via a policy function or any similar embedding. Instead EAs start with a population of simple agents or individuals which interact with the environment and obtain a reward value, similar to RL. The reward values are translated into fitness scores for each individual. The fitness score is then used to select a set of individuals who will be responsible for generating next generation of individuals using genetic algorithms (GA). We will discuss EAs in detail later in Section 2.3.

We perform a detailed computation and memory profiling of an EA called NEAT [44] and identify several opportunities for parallelism *(Gene level parallelism - GLP)* and data reuse*(Genome level reuse - GLR)*. We use the results of the analysis to drive the design of a hardware accelerator called *EvE* for extracting maximum energy and compute efficiency. Fig. 1(b) demonstrates where EvE would fit within an autonomous learning engine. EvE provides over five-orders of magnitude higher efficiency over general-purpose desktop and mobile CPUs and GPUs.

2 Background

2.1 Supervised Learning

Supervised learning is arguably the most widely used learning method used in the present. The basic idea is, that the output of the model is computed for a given set of input and is compared against an existing label or the ground truth to generate an error value. Based on the error obtained the parameters of the model is perturbed such that the error is reduced. This is done iteratively till convergence is achieved.

Among the various training methods available today, Backpropagation (BP) [36] is the most popular and is exclusively used for training deep networks. However, BP has the following limitations as the learning/training engine for general purpose AI:

- Dependence on large structured & labeled datasets to perform efficiently [20, 37]
- Effectiveness is heavily tied to the NN topology, as we witnessed with deeper convolution topologies [25, 27] that led to the birth of Deep Learning.
- Extreme compute and memory requirements. It often takes weeks to train a deep network on a compute cluster consisting of several high end GPUs.

2.2 Reinforcement Learning (RL)

Reinforcement learning is used when the structure of the underlying policy function is not known. For instance, suppose we have a a robot learning how to walk. The system has a finite set of outputs (say which leg to move when and in what direction), and the aim is to learn the right policy function so that the robot moves closer to its target destination. Starting with some random initialization, the agent performs a set of actions, and receives an reward from the environment for each of them, which is a metric for success or failure for the given goal. The goal of the RL algorithm is to update its policy such that future reward could be maximized. At a fundamental level, RL algorithms learn the representation of the environment (which can be modeled as a NN). RL algorithms perturb the actions, and perform backpropagation to compute the update to the parameters. RL algorithms can learn in environments with scarce datasets and without any assumption on the underlying NN topology, but the reliance on BP still makes them computationally very expensive.

2.3 Evolutionary Algorithms (EA)

EA get their name from biological evolution, since at an abstract level they be seen as sampling a population of individuals and allowing the successful individuals to determine the constitution of future generations. The algorithm starts with a pool of agents or individuals, each one of which independently tries to perform some action on the environment to solve the problem. Each individual agent is then assigned a fitness value, depending upon the effectiveness of the action taken by them. Similar to biological systems, each agent is represented by a list of parameters called a *genome*. Each of these parameters, called *genes*, encode a particular characteristic of the individual. After the fitness calculation is done for all, next generation of individuals are created by crossing over and mutating the genomes of the parents. This step is called reproduction and only a few individuals, with

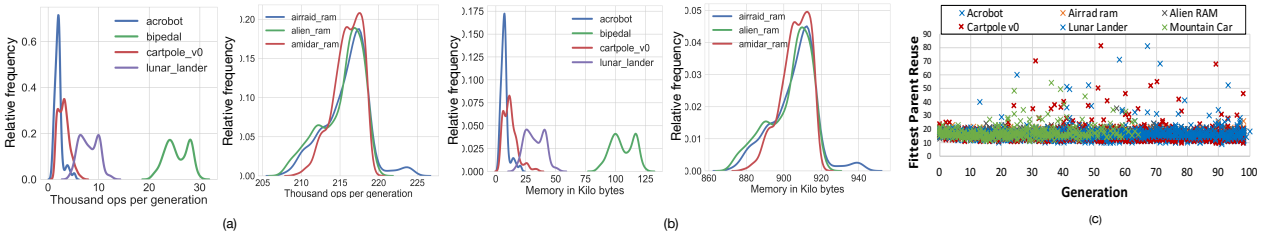


Figure 3. (a) Computation (i.e., Crossover and Mutations) Ops and (b) Memory Footprint of applications from OpenAI Gym in every generation. A distribution is plotted across all generations till convergence and 100 separate runs of each application. (c) Plot depicting the fitness of the fittest parent in a generations for 100 separate runs across workloads.

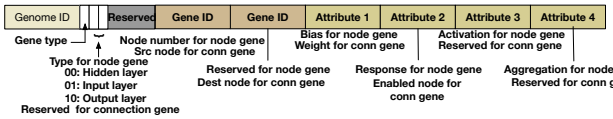


Figure 4. Data structure used for Genes in EvE

highest fitness values are chosen to act as parents in-order to ensure that only the fittest genes are passed into the next generation. These steps are repeated multiple times until some completion criteria is met. Fig. 2(a) illustrates the flow.

Mathematically EA can be viewed as a class of black-box stochastic optimization techniques. The reason they are "black-box" is because they do not make any assumptions about the structure of the underlying function being optimized, they can only evaluate it (like a lookup function). This leads to the fundamental difference between RL and EA. Both try to optimize the expected reward, but RL perturbs the action space and uses backpropagation (which is computation and memory heavy) to compute parameter updates, while EA perturbs the parameters (i.e., nodes and connections) directly.

3 Computational Behavior of EAs

3.1 The NEAT Algorithm

TWEANNS are a class of evolutionary algorithms which evolve both the topology and weights for given NN simultaneously. *Neuro-Evolution for Augmented Topologies* (NEAT) is one of the algorithms in this class developed by Stanley et al [44]. Fig. 2(b) depicts the steps and flow of the NEAT algorithm. Without loss of generality, we use NEAT for all our studies in this work.

Population. The population in NEAT is the set of NN topologies in every generation that each run in the environment to collect a fitness score.

Genes. The basic building block in NEAT is a *gene*, which can represent either a NN node (i.e., neuron), or a connection (i.e., synapse). An overview of a gene is shown in Fig. 4. Each node gene can uniquely be described by an id, the nature of activation (e.g., ReLU) and the bias associated with it. Each connection can be described by its starting and end nodes, and its hyper-parameters (such as weight, enable). A *genome*, which the collection of these genes, uniquely describes one NN in the population.

Initialization. NEAT starts with a initial population of very simple topologies comprising only the input and the output layer. It evolves into more complex and sophisticated topologies using the *mutation* and *crossover* functions.

Mutation. Akin to biological operation, mutation is the operation in which a child gene is generated by tweaking the parameters of the parent gene. For instance, a connection gene can be mutated by modifying the weight parameter of the parent gene. Mutations can also involve addition or deletion of genes, with a certain probability.

Crossover. Crossover is the name of the operation in which a child gene for the next generation is created by cherry picking attributes from two parent genes.

Speciation and Fitness Sharing. EAs in essence work by pitting the individuals against each other in a given population and competitively selecting the fittest. However, this policy can lead premature pruning of new individuals with useful topological features. NEAT counteracts this by using speciation and fitness sharing. *Speciation* works by grouping a few individuals within the population with a particular niche. Within a species, the fitness of the younger individuals is artificially increased *(fitness sharing)* so that they are not obliterated when pitted against older, fitter individuals, thus ensuring that the new innovations are protected for some generations and given enough time to optimize.

3.2 Compute and Memory Trends

Open AI gym [10] is a popular platform for testing reinforcement learning algorithms. The platform has multiple environments for simulating a variety of tasks like robotic control, locomotion, gameplay and so on. We selected seven environments - *acrobot, bipedal, cartpole_v0, lunar_lander, airraid_ram, alien_ram, amidar_ram* - and evolved one agent for each environment using NEAT. In all the experiments we used a population size of 150, and started with a bare minimum topology of input and output layer. The only thing changed between the experiments is the fitness function, which in most cases is the reward or an affine transformation of the reward value obtained from the environment.

Compute Trends. Fig. 3(a) show the distribution of the number of compute (crossover and mutation) operations within a generation. The distribution is plotted across all

generations till the application converged and across 100 runs of each application.

Opportunity for Parallelism. Mutations and crossovers of genes of a genome can occur in parallel, demonstrating the raw parallelism provided by EAs. We observe that the mutations and crossovers are in thousands in one class of applications, and are in the range of hundred thousands in another class. We term this as ***gene level parallelism (GLP)*** in this work. Moreover, the amount of GLP scales as the size of each NN (i.e., genome grows).

Memory Trends. Fig. 3(b) shows the distribution in memory footprint of the population in each generation for all the workloads. For EAs, the memory footprint is simply the size (i.e., number of genes) of all genomes (NNs) in the population in that generation. Within OpenAI gym, even for the large workloads corresponding to the Atari game environments, the memory footprint of the entire population is shy of 1MB.

Opportunity for Data Reuse. Given that only a handful of parents contribute to creation of next generation, one parent genome is reused multiple times during crossover. We refer to this property as ***genome level reuse(GLR)***. This behavior can be leveraged to provide energy efficiency and ease memory bandwidth. Fig. 3(c) demonstrates that in most cases one parent is on average used around 20 children, while in some cases like Cartpole and Lunar lander, this GLR number is around 80 for a population size of 150.

3.3 Opportunity for Spatial Acceleration

The compute and memory behaviors lead us to the following insights for our accelerator design:

- Since the compute operations fall into two classes, crossovers and mutations, a custom implementation of these compute units in hardware can create low footprint yet high performance power efficient solutions.
- The GLP offered by EAs suggest that it is possible to evolve NNs with reasonable accuracy in a short period of time, if enough compute elements are available.
- A spatial accelerator can be developed by packing a large number of these compute units together, which essentially provides cheap compute in large numbers within a tight power envelope.
- Given the reasonable memory foot print (less than 1MB) and GLR opportunity, it is evident that a sufficiently sized on chip memory can help remove/reduce off-chip accesses significantly, saving both energy and bandwidth.
- If we can reduce the energy consumption of the compute elements and store all genomes locally (on-chip or in memory), complex behaviors can be evolved even in mobile autonomous agents.

4 EvE Microarchitecture

With the motivations discussed in the previous sections, we designed a hardware accelerator ensuring maximum utilization of GLP, which we describe next.

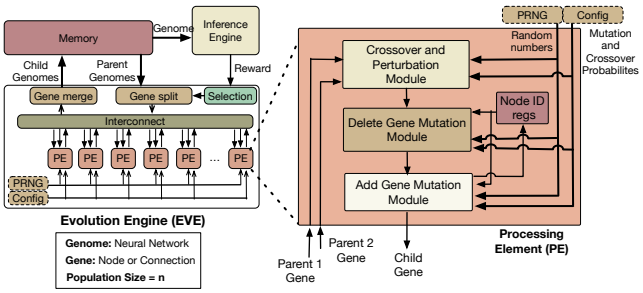


Figure 5. Schematic of the architecture of EvE and interaction with memory and inference engine. Expanded schematic shows internals of PE in EvE.

Gene Encoding Fig. 4 shows the structure for a gene we use in our design. NEAT uses two types of genes to construct a genome, a node gene which describe vertices and the connection gene which describe the edges in the neural network graph. We use 64 bits to capture both types of genes.

4.1 Processing Element (PE)

Fig. 5 shows the schematic of our PE. It has a four-stage pipeline. These stages are shown in Fig. 6. Perturbation, Delete Gene and Add Gene are three kinds of mutations that our design supports.

Crossover Engine. The crossover engine receives two genes, one from each parent genome. As described in Section 3.1, crossover requires picking different attributes from the parent genome to construct the child genome. The random number from the PRNG is compared against 0.5 and used to select one of the parents for each of the attributes. We provide the ability to program in a bias, depending on which of the two parents contributes more attributes (i.e., is preffered) to the child. This logic is replicated for each of the 4 attributes.

Perturbation Engine. A perturbation probability is used to generate a set of mutated values for each of the attributes in the child gene that was generated by the crossover engine.

Delete Gene Engine. There are two types of genes in a given genome, node and connection, and implementing gene deletion for each of them differs. A gene is deleted depending on the deletion probability, and compared against a PRNG. However, gene deletion is slightly more complicated since a node gene that has been deleted could leave an already created connection gene floating. Gene deletion for node and connection genes is handled in the following way. For a node gene, two things are checked, the deletion probability and number of nodes deleted in the genome. If the number of nodes deleted is more than a certain threshold, the gene is not deleted. This is required in some EAs, such as NEAT, to ensure a minimum number of nodes in the NN. In case the logic decides to delete the gene, its node ID is saved in the deleted nodes list, and a counter representing number of nodes deleted is incremented by one. For connection genes, the source and destination nodes are checked against the list

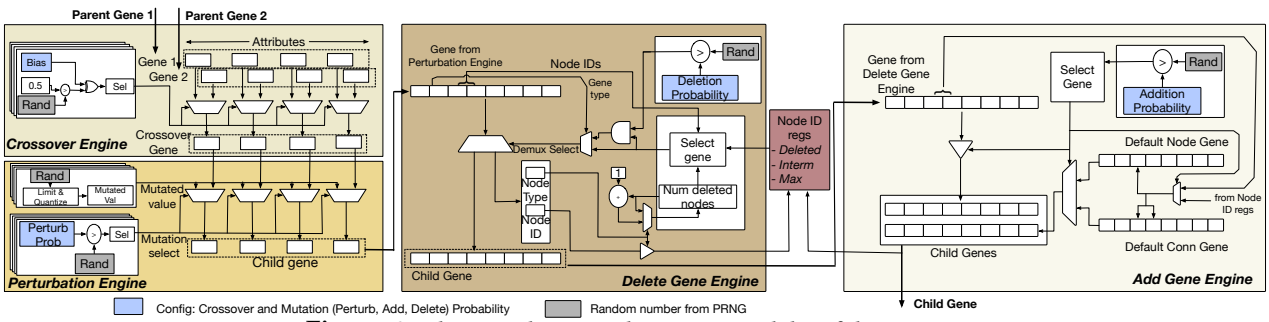


Figure 6. Schematic depicting the various modules of the Eve PE.

of deleted nodes to figure out if the connection is floating or not. If it is found to be floating, it is deleted as well.

Add Gene Engine. This is the fourth and final stage of the PE pipeline. As in the case of the previous stage, depending upon the type of the gene, the implementation varies. To add a new node gene, the logic inserts a new gene with default attributes and a node ID greater than any other node present in the network. Additionally two new connection genes are generated and the incoming connection gene is dropped. The addition of a new connection gene is carried out in two cycles. When a new connection gene arrives, the selection logic compares a random number with the addition probability. If the random number is higher, then the source of the incoming gene is stored. When the next connection gene arrives, the logic reads the destination for that gene, appends the stored source value and default attributes, and creates a new connection gene. This mechanism ensures that any new connection gene that is added by this stage always has valid source and destinations.

4.2 Gene Movement

Next, we describe all blocks that manage gene movement.

Gene Selector. The selection logic interprets the reward values obtained from inference engine into corresponding fitness scores. Once all the fitness values are available, the logic readjusts fitnesses to implement fitness sharing in NEAT. The readjusted fitnesses are then compared to a threshold to identify parents for the next generation. This list is hence forwarded to the *Gene Split* block.

Gene Split. The Gene Split block orchestrates the movement of genes from the Genome Buffer to the PEs inside EvE. In the crossover stage, the keys (i.e., node id) for both the parent genes need to be the same. However both the parents need not have the same set of genes or there might be a misalignment between the genes with the same key among the participating parents. The gene split block therefore sits between the PEs and the Genome Buffer to ensure that the alignment is maintained and proper gene pairs are sent to the PEs every cycle.

Gene Merge. Once a child gene is generated, it is written back to the Gene Memory as part of the larger genome it is part of. This is handled by the Gene Merge block.

Pseudo Random Number Generators (PRNG). The PRNG feeds a 8-bit random numbers every cycle to all the PEs, as shown in Fig. 5. We use the XOR-WOW algorithm to implement our PRNG (also used within NVIDIA GPUs).

Interconnection network The interconnection network manages the distribution of parent genes from the Gene Split to the PEs and collection of child genes at the Gene Merge. We explored two design options. Our base design uses separate high-bandwidth buses, one for the distribution and one for the collection [14]. However, recall that the NEAT algorithm offers opportunity for reuse of parent genomes across multiple children, as we showed in Section 3.2. Thus we also consider a tree-based network with multicast support [29] and evaluate the savings in SRAM reads in Section 5.

5 Evaluation

5.1 Implementation

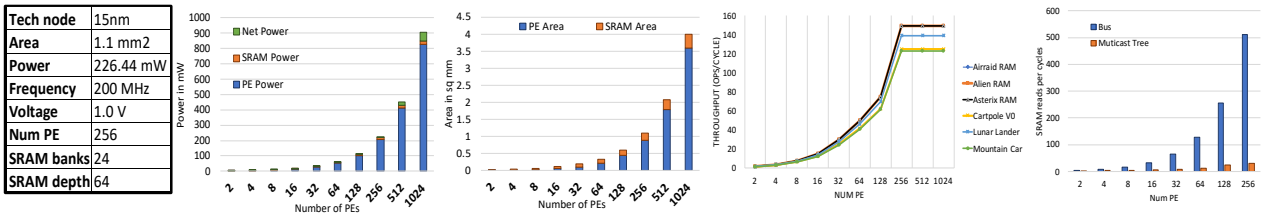
We implemented the EvE microarchitecture we discussed in Section 4 in RTL and synthesized the design using Nangate 15nm PDK at 200MHz target. We list all the parameters of EvE for this design point in a table in Fig. 7(a). The post synthesis results reported 3509.01 μm^2 in area footprint for each PE, consuming 0.808 mW of power. We also synthesized a 64x64 SRAM block using TSMC 28nm memory compilers and applied *Dennard's scaling* [16] to estimate power and area numbers at 15nm. We choose a power budget of 250 mW at a frequency of 200MHz, motivated by recently published DNN accelerators chips [14, 17, 32, 41]. Fig. 7(b) shows that we reach this power limit with 256 PEs. Fig. 7(c) shows that a design with 256 EvE PEs demands about 1.1mm^2 in area.

5.2 Evaluation Methodology

We modify the code in NEAT python library [5] to generate a trace of reproduction operations for the various workloads presented in Section 3.2. Each line on the trace captures the generation, the child gene and genome id, the type of operation - mutation or crossover, and the parameters changed or added or deleted by the operations. These traces serve as proxy for our workloads when we evaluate our accelerator and GPU implementations.

CPU evaluations. We measure the completion time and average power consumption by running traces on 6th gen Intel i7 desktop CPU and ARM Cortex A57 on nVidia Jetson

Tech node	15nm
Area	1.1 mm2
Power	226.44 mW
Frequency	200 MHz
Voltage	1.0 V
Num PE	256
SRAM banks	24
SRAM depth	64



(a) (b) (c) (d) (e)

Figure 7. (a) Parameters for EvE implementation (b) Variation of power consumption of EvE with number of PEs. (c) Variation in area footprint of EvE with number of PEs. (d) Throughput variation with number of PEs in various Open AI workloads. (e) SRAM reads per cycle with a bus vs multicast tree (that leverages genome-level reuse) interconnect inside EvE.

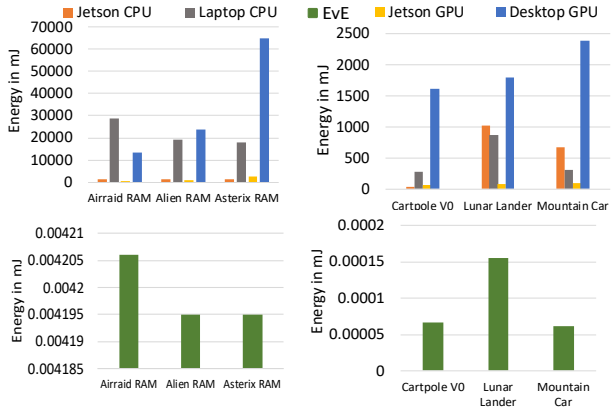


Figure 8. Charts showing energy consumption for reproduction by CPU, GPU and EvE for different workloads. The top row is for CPU and GPU; the bottom row is for EvE

TX2 board. To measure the power on desktop CPUs we use Intel's power gadget tool [3]. While on the Jetson board we use the onboard instrumentation amplifier INA3221.

GPU evaluations. We wrote CUDA program, to read activities from our traces in each generation and launch them in parallel. To ensure that the correctness of the operations are maintained, we apply some constraints in ordering, for example crossovers precede mutation in time. We get power and completion time by running this program with traces from various operations, on nVidia GTX 1080 desktop GPU and Pascal GPU on Jetson TX2.

EvE evaluations. The traces along with the parameters obtained by our analysis in Section 5.1 are used to estimate the energy consumption for our chosen design point for EvE.

5.3 Performance and Energy efficiency of EvE

Fig. 7(d) we observe that throughput of EvE is proportional to number of PEs. The plot tapers off when the number of PEs is greater than number of genomes. When a multicast tree is used over a bus to exploit GLR, we get an order of $\frac{n}{log(n)}$ savings in memory reads (see Fig. 7(e)). In our case this translates to 94% less SRAM reads.

In Fig. 8 we plot the end-to-end energy consumption for running the evolution code of NEAT on EVE, compared to the Desktop and Jetson (mobile) CPUs, and Laptop and Jetson (mobile) GPU described in Section 5.2. Among the conventional methods Jetson GPU is the most energy efficient. When compared to Jetson GPU, EvE consumes 10^5 times less energy in *Atari environments* (eg. for Airraid RAM Jetson GPU takes up 521mJ while EvE takes 40μJ). For smaller workloads, the savings are more pronounced and are in order of 10^6 (eg. for MountainCar 98mJ on Jetson GPU vs 6μJ on EvE).

6 Related Work

Neuroevolution. Researchers have been looking into neuro-evolution algorithms for decades [18, 40, 45]. NEAT [44] and its variants [42, 43] have been studied and modified for increasing accuracy [15, 21, 23]. Recent research has demonstrated the promise of applying neuro-evolution to generate *deep* neural networks [9, 22, 31, 34, 46, 47]. One prior work proposed a hardware implementation of NEAT [30] but focuses on running the inference across the population in parallel. This is the first work, to the best of our knowledge, that has attempted to parallelize the evolutionary algorithm itself across genes and built a hardware accelerator for it.

Online Learning. Traditional RL methods have also gained traction in the last year with Google announcing AutoML [6, 8, 49]. In-situ learning from the environment has also been approached from the direction of spiking neural nets (SNN) [26, 35, 39]. Recently intel released a SNN based online learning chip Loihi [4]. IBM's TrueNorth is also a SNN chip. SNNs have however not managed to demonstrate accuracy across complex learning tasks.

DNN Acceleration. Hardware acceleration of DNNs is an active research topic with a lot of architecture choices [7, 11–13, 19, 24, 28, 33, 48] and silicon implementations [14, 17, 32, 41]. However, all of these focus on inference, and can readily be used in conjunction with EvE, as Fig. 5 shows.

7 Conclusion

In this work we identify an opportunity to expand the reach of intelligent systems towards general purpose AI by automating topology generation of ANNs. Our experiments with evolutionary algorithms show the opportunity to come up with extremely power efficient hardware solution. We implement a hardware accelerator for automated topology and weight generation of ANNs. Our evaluations show ***five to six orders of magnitude in energy efficiency improvements*** over commodity CPUs and GPUs running the same algorithm, indicating an optimistic path forward.

References

[1] *Alphago*, https://deepmind.com/research/alphago, 2017.

[2] *Atari open ai environments*, https://gym.openai.com/envs/#atari, 2017.

[3] *Intel power gadget*, https://software.intel.com/en-us/articles/intel-power-gadget-20, 2017.

[4] *Intels new self-learning chip promises to accelerate artificial intelligence*, https://newsroom.intel.com/editorials/intels-new-self-learning-chip-promises-accelerate-artificial-intelligence/, 2017.

[5] *Neat python*, https://github.com/CodeReclaimers/neat-python, 2017.

[6] *Using machine learning to explore neural network architecture*, https://research.googleblog.com/2017/05/using-machine-learning-to-explore.html, 2017.

[7] Jorge Albericio et al., *Cnvlutin: Ineffectual-neuron-free deep neural network computing*, ISCA, 2016, pp. 1–13.

[8] Bowen Baker et al., *Designing neural network architectures using reinforcement learning*, arXiv preprint arXiv:1611.02167 (2016).

[9] Justin Bayer et al., *Evolving memory cell structures for sequence learning*, Artificial Neural Networks–ICANN 2009 (2009), 755–764.

[10] Greg Brockman et al., *Openai gym*, arXiv preprint arXiv:1606.01540 (2016).

[11] Tianshi Chen et al., *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning*, ASPLOS, 2014, pp. 269–284.

[12] Yu-Hsin Chen et al., *Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks*, ISCA, 2016, pp. 367–379.

[13] Yunji Chen et al., *Dadiannao: A machine-learning supercomputer*, Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2014, pp. 609–622.

[14] Chen, Yu-Hsin and others, *Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks*, IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers, 2016, pp. 262–263.

[15] David B D'Ambrosio and Kenneth O Stanley, *Generative encoding for multiagent learning*, Proceedings of the 10th annual conference on Genetic and evolutionary computation, ACM, 2008, pp. 819–826.

[16] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc, *Design of ion-implanted mosfet's with very small physical dimensions*, IEEE Journal of Solid-State Circuits **9** (1974), no. 5, 256–268.

[17] Giuseppe Desoli et al., *14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems*, Solid-State Circuits Conference (ISSCC), 2017 IEEE International, IEEE, 2017, pp. 238–239.

[18] Shifei Ding et al., *Using genetic algorithms to optimize artificial neural networks*, Journal of Convergence Information Technology, Citeseer, 2010.

[19] Zidong Du et al., *Shidiannao: Shifting vision processing closer to the sensor*, ACM SIGARCH Computer Architecture News, vol. 43, ACM, 2015, pp. 92–104.

[20] Mark Everingham et al., *The pascal visual object classes (voc) challenge*, International journal of computer vision **88** (2010), no. 2, 303–338.

[21] Chrisantha Fernando et al., *Convolution by evolution: Differentiable pattern producing networks*, Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, ACM, 2016, pp. 109–116.

[22] Keyan Ghazi-Zahedi, *Nmode—neuro-module evolution*, arXiv preprint arXiv:1701.05121 (2017).

[23] David Ha et al., *Hypernetworks*, arXiv preprint arXiv:1609.09106 (2016).

[24] Song Han et al., *Eie: efficient inference engine on compressed deep neural network*, ISCA, 2016.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep Residual Learning for Image Recognition*.

[26] Nikola Kasabov et al., *Dynamic evolving spiking neural networks for online spatio-and spectro-temporal pattern recognition*, Neural Networks **41** (2013), 188–201.

[27] Alex Krizhevsky et al., *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 25 (NIPS2012) (2012), 1–9.

[28] Jaeha Kung et al., *Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware*, ISLPED, 2016, pp. 168–173.

[29] Hyoukjun Kwon et al., *Rethinking nocs for spatial neural network accelerators*, Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip, ACM, 2017, p. 19.

[30] Daniel Larkin et al., *Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices*, Neural Information Processing, Springer, 2006, pp. 1178–1188.

[31] Risto Miikkulainen et al., *Evolving deep neural networks*, arXiv preprint arXiv:1703.00548 (2017).

[32] Bert Moons et al., *14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi*, Solid-State Circuits Conference (ISSCC), 2017 IEEE International, IEEE, 2017, pp. 246–247.

[33] Angshuman Parashar et al., *Scnn: An accelerator for compressed-sparse convolutional neural networks*, Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 27–40.

[34] Esteban Real et al., *Large-scale evolution of image classifiers*, arXiv preprint arXiv:1703.01041 (2017).

[35] Daniel Roggen et al., *Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot*, Evolvable hardware, 2003. proceedings. nasa/dod conference on, IEEE, 2003, pp. 189–198.

[36] David E Rumelhart et al., *Learning representations by back-propagating errors*, Cognitive modeling **5**, no. 3, 1.

[37] Olga Russakovsky et al., *Imagenet large scale visual recognition challenge*, International Journal of Computer Vision **115** (2015), no. 3, 211–252.

[38] Tim Salimans et al., *Evolution strategies as a scalable alternative to reinforcement learning*, arXiv preprint arXiv:1703.03864 (2017).

[39] Catherine D Schuman et al., *An evolutionary optimization framework for neural networks and neuromorphic architectures*, Neural Networks (IJCNN), 2016 International Joint Conference on, IEEE, 2016, pp. 145–154.

[40] Gene I Sher, *Dxnn platform: the shedding of biological inefficiencies*, arXiv preprint arXiv:1011.6022 (2010).

[41] Jaehyeong Sim et al., *14.6 a 1.42 tops/w deep convolutional neural network recognition processor for intelligent ioe systems*, Solid-State Circuits Conference (ISSCC), 2016 IEEE International, IEEE, 2016, pp. 264–265.

[42] Kenneth O Stanley, *Compositional pattern producing networks: A novel abstraction of development*, Genetic programming and evolvable machines **8** (2007), no. 2, 131–162.

[43] Kenneth O Stanley et al., *A hypercube-based indirect encoding for evolving large-scale neural networks*.

[44] Kenneth O Stanley and Risto Miikkulainen, *Efficient reinforcement learning through evolving neural network topologies*, Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, Morgan Kaufmann Publishers Inc., 2002, pp. 569–577.

[45] Christopher M Taylor, *Selecting neural network topologies: A hybrid approach combining genetic algorithms and neural networks*, Master of Science, University of Kansas (1997).

[46] Phillip Verbancsics and Josh Harguess, *Generative neuroevolution for deep learning*, arXiv preprint arXiv:1312.5355 (2013).

[47] Lingxi Xie and Alan Yuille, *Genetic cnn*, arXiv preprint arXiv:1703.01513 (2017).

[48] Chen Zhang et al., *Optimizing fpga-based accelerator design for deep convolutional neural networks*, FPGA, 2015, pp. 161–170.

[49] Barret Zoph and Quoc V Le, *Neural architecture search with reinforcement learning*, arXiv preprint arXiv:1611.01578 (2016).