E3: A HW/SW Co-design Neuroevolution Platform for Autonomous Learning in Edge Device

Sheng-Chun Kao
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
felix@gatech.edu

Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
tushar@ece.gatech.edu

***Abstract*—The true potential of AI can be realized once we move beyond supervised training using labelled datasets on the cloud to autonomous learning on edge devices. While techniques like Reinforcement Learning are promising for their autonomous learning ability, they exhibit high compute and memory requirements due to gradient computations, making them prohibitive for edge deployment. In this paper, we propose E3, a HW/SW co-designed edge learning system on a FPGA. E3 uses a gradient-free approach called neuro-evolution (NE) to evolve the neural network (NN) topology and weights dynamically. The NNs evolved using NE are highly irregular, and a population of such NNs need to be evaluated quickly in order for the NE algorithm to make progress. To address this, we develop INAX, a specialized accelerator inside E3 for efficient irregular network computation. INAX leverages multiple avenues of parallelism both within and across the evolved NNs. E3 shows averaged 30× speedup than CPU-based solution across a suite of OpenAI environments.**

I. Introduction

Deep Neural Networks (DNNs) have surpassed human performance in many tasks such as image recognition and speech processing. The common scenario of AI deployment today is to train hand-tuned DNN models on the cloud on GPU clusters, and deploy the trained model for inference on the edge over accelerators [6], [9], [1]. These inference accelerators enable low latency and high energy-efficiency deployments hardware datapaths customized for running DNNs.

The focus of this work is on an emerging deployment scenario that we call autonomous learning. This encapsulates two use-cases. The first, that we call ***model-tuning***, is one where a trained model deployed on an edge device (say a self-driving car or drone) needs to be tuned further for the inputs it encounters. This use-case is common in robotics applications, since the environment is full of variance (say a robot trained to walk on grass but now encounters sand); it becomes inefficient to collect all the training data beforehand for a perfect model. Moreover, transmitting new data samples to the cloud everytime, invoking a training loop, and beaming the updated model back is inefficient because of the transmission bandwidth and the delayed reaction to the environment, which can be catastrophic for tasks with real-time requirements. A better strategy is to have an adequate model that trained on generic environment and continuously train it on the target environment. The second, that we call ***model-replacement*** is the use-case of continuous learning where an autonomous agent is deployed and given new tasks for which no trained model exists (say a robot/drone deployed in remote regions). Here, relying on constant connectivity to a backend cloud for training is infeasible or insecure. In summary, autonomous learning on the edge is needed because the tasks on the edge are often full of diversities and with unknown complexity; in contrast to the cloud learning [16], [2], where the learning data are often well organized (i.e., labeled) and the task is well defined.

This naturally raises two questions: (i) what algorithm do we use for autonomous learning, and (ii) what edge system do we build for running this algorithm. On the algorithmic end, federated learning (FL) and reinforcement learning (RL) have gained popularity for these use-cases. In FL, each edge device learns locally starting from a generic model and periodically transmits the update to the cloud. However, FL requires a pre-defined task and a pre-defined DNN model structure - this can serve the *model-tuning* scenario, but not the *model-replacement* scenario. RL [7], [26] offers promise for the *model-replacement* scenario where the target model architecture is unknown. It has shown successful demonstrations for strategy games [40], robotics [17], and machine translation [45]. However, RL approaches [37], [28], [13], [21], [44], [18], [46] still rely on human-picked and tuned DNN architectures as the underlying model, based on the practitioners' understanding of the tasks, making it challenging for autonomous learning. In addition to the algorithmic challenges, both FL and RL rely on gradient computations and updates via backpropogation for the model update, which is known to be highly compute and memory heavy, taking up to hundreds of GPU hours [52] on cloud servers, and thus infeasible for running directly on a low-power edge device.

In this work, we leverage a technique that has shown promise on both the algorithmic and system fronts called Neuro-evolution (NE). NE relies on gradient-free approaches like genetic algorithms (GA) or evolutionary strategies (ES) to evolve the model in response to stimuli from the environment, instead of backpropogating error gradients. NE approaches have demonstrated competitive convergence and accuracy as RL [35], [43]. It can thus work for both model-tuning and model-replacement use-cases. Moreover, prior work [36], [24] has shown the promise of NE for edge learning due to its low memory footprint and opportunity for compute parallelization.

One challenge with gradient-free methods like NE, however, is that they require huge amounts of evaluation computations[1] as a replacement of backpropogation, demanding a super efficient inference accelerator. However, self-evolved network topologies from NEs are often irregular, not structured like hand-designed CNNs and Multi Level Perceptrons (MLPs), which makes off-the-shelf inference accelerators unsuitable. In this paper we demonstrate a closed-loop FPGA prototype of a NE-based on-device autonomous learning system. Our key novelty is an inference accelerator (running on the FPGA) for irregular NNs, interacting with the environment and an evolution engine (both running on the CPU).

The primary contributions of this work are as follows:

- We profile a NE algorithm called NEAT [42] to identify opportunities for HW-SW co-design. We also contrast it against two popular RL algorithms in terms of convergence, runtime and network complexity.
- We propose a new irregular NN accelerator called INAX with high adaptability to different irregular NN topologies. We investigate parallelization opportunities when running irregular NNs within INAX, and develop heuristics for enhancing utilization.
- We integrate INAX into a light-weight on-device autonomous learning platform, E3 (Eval-Evol-Engine), enabling efficient human-off-the-loop edge training.
- We prototype E3 on a Xilinx ZCU104 board, and observe averaged 30× speedup over a SW-only solution across a suite of OpenAI environments.

II. Background and Motivation

In this section, we provide background on relevant learning techniques, and qualitatively contrast them in Table I.

A. Deep Learning (DL) Techniques

Supervised Learning. Supervised learning are widely used in regression, image classification, object detection, and speech recognition. The learning scheme required a pre-collected dataset with label (e.g., ImageNet [8]), a pre-defined DNN structure (number of layers and per-layer shape), and a set of training hyperparameters (optimization algorithm, learning rate). When training, we execute inference on the batched input data and evaluate the error with their corresponding labels. Conventionally, we use gradient descent to update the weights in DNNs and train until the error converges.

Un-supervised Learning. Un-supervised learning requires a pre-collected dataset (but without label), a DNN or ML model, and hyperparameters. Clustering is one common unsupervised learning algorithm, where K-mean [3], expected maximization [4] and other ML algorithms [32], [51] are used. One significant challenge for these algorithms is that their performance are heavily dependent on hyperparameters such as number of cluster or number of optimizers, which often require manual-tuning to adapt to different tasks.

[1]Evaluation is akin to inference phase of NN, where forward pass through the NN is performed to determine a reward for the task at hand.

TABLE I
COMPARISONS OF ALGORITHMS.

	DL	RL	EA	NEAT
Continuous learning	Not supported (conventionally)	Supported	Supported	Supported
Data	Labeled/unlabeled	Unlabeled	Unlabeled	Unlabeled
Network structures	Manual	Manual	Manual	**Automatic**
Memory overhead	High	High	**Light**	**Light**
Compute	Heavy (BP)	Heavy (BP)	**Low** No-BP (Heavy influence)	**Low** No-BP (Heavy influence)
Task flexibility	Limited to tasks that the data can be pre-collected	**Flexible** to tasks but a pre-defined network structure is needed for each task	**Flexible** to tasks but a pre-defined network structure is needed for each task	**Flexible** to tasks and **flexible** autonomously-tuned network structure

Semi-supervised Learning. Semi-supervised learning, such as Generative Adversarial Networks (GANs) [15], [27], use a small amount of labelled data and boost the model performance with unlabelled data.

Conventionally, for supervised, autoenconder in unsupervised, and semi-supervised learning, we use backpropogation (BP) to update the weight of the DNN models. BP propagates the gradient of the error from output layer to input layers. To accomplish the calculation in the backward path, we need to store the intermediate values (weight, activations) along the forward path, which makes BP extremely memory and compute intensive.

B. Continuous Learning Techniques

Deep Reinforcement Learning (RL/DRL). RL is often used in a continuous learning scenario, where the agent in RL interacts with the environment and updates its policy to maximize the expected reward/fitness feedback from the environment. The policy is often abstracted with a DNN, whose weights are updated according to the fitness. DRLs such as DQN [31], A2C [28], PPO2 [37] and others[13], [21], [18], [46], [44], have found great success in robotic controls [17], playing AI game [30], or playing chess at human levels[40]. However, DRLs are gradient-based methods and require BP for training the DNN, leading to high computation complexity and high memory requirement. Moreover, in many DRLs a large replay buffer, which stores the experiences along the episodes, are often required. This intensifies the memory requirement.

Evolutionary Algorithms (EA). EAs work in the same setting as RL, where the agent interacts with the environment continuously and update its solution to maximize the fitness. Recent research from OpenAI [35] and Uber AI lab [43] have shown EA techniques demonstrate high scalability for parallelization, and hence can lead to shorter training time than DRLs at similar accuracies. In general, EA is gradient-free without BP, and thus has lower algorithm complexity and memory requirement than DRL. However, the trade-off for gradient-free is more instances of inference to evaluate the performance, i.e., it is a training scheme relieving the memory overhead and computation of BP, but posing more pressure on the number of computation in inference.

TABLE II
TERMINOLOGY IN NEAT.

Term	Description
Gene	Basic building block of NN component (node/connections). Node gene: node bias value, node activation. Connection gene: Linkage between nodes, weight of the linkage.
Genome (Individual)	A sequence of genes that describe a complete NN. Each NN is treated as an individual that is evaluated independently.
Elite	A set of individuals that has the highest evaluated fitness.
Population (Generation)	An entire set of individual forms a population (one generation). The populations evolve with time by selecting the elites and mutating/crossovering intra populations.

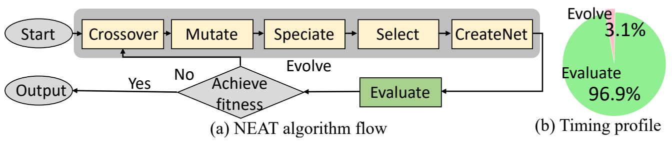


Fig. 1. (a) The algorithm flow and (b) timing profile of NEAT.

C. NEAT

In this work, we consider an branch of EA techniques called Neuro-Evolution of Augmented Topology (NEAT) [42] that uses GAs internally to evolve a NN in response to the reward. While some ES and GA methods [35], [43] require a human-defined network topology and only evolve weights (like DRL), NEAT [42] evolves and optimizes both the topology and weights. Hence, it can effectively adapt to tasks with different complexities and potentially find smaller NNs with the same performance (through inherent pruning [14]), making it promising for edge deployment. Table I summarizes different characteristics of DL, RL, EA and NEAT. We perform a quantitative comparison of RL and NEAT in Sec. III.

Terminology in NEAT. We list NEAT-specific terminology we will use in the paper in Table II.

Initial state of NEAT. The algorithm starts with the basic two layer network: input layer and output layer, where the number of nodes is decided by the states (inputs) and actions (outputs) respectively. The network evolves with time, adapting to the task.

Algorithmic Flow of NEAT. The algorithm flow of NEAT is described in Fig. 1(a), and its major functions are listed in Table III. At the start, we have a population of genomes. "CreateNet" decodes those genomes to NNs. "Evaluate" runs inference and calculates fitness value of each NN. Next, "Evolve", using GA operators like "Mutate", "Crossover", and "speciate", to perturbs the population and reproduces the next generation of population. More specifically, "evolve" selects some well-performed individuals as elite parents; "mutate" reproduces children by tweaking genes of elite parents; "crossover" reproduces children by randomly mixing genes of two elite parents. Then in the pool of the current population and its reproduced children, "speciate" divides them into species by their similarity of topology. This is done to ensure diverse evolved traits survive through generations, even if their genomes do not perform well initially. Finally, "evolve" selects the new population out of each species to enter the next generation. These "evaluate" and "evolve" loops go on until the fitness value is achieved.

TABLE III
THE MAJOR FUNCTIONS IN NEAT.

Phase	Function	Descriptions
Evaluate	Evaluate	Inference the NN and compute its fitness through Env.
Evolve	CreateNet	Decode the genes to nodes and connections, solve the dependency among nodes, and formulate them into NN topology.
	Mutate	Perturb a parent's gene to reproduce individuals.
	Crossover	Blend two parents' gene to reproduce individuals.
	Speciate	Classify individual by their topology similarity so that they only compete within group. It protects the young individuals from elimination before well-evolved.
Interactive Environment (Env)		Applications that require the autonomous agent to make a sequence of decisions on the tasks interactively. The environment responds with a **reward** or **fitness**, that NEAT uses to improve its decisions via evolution.

TABLE IV
ANALYSIS OF OVERHEAD IN ALGORITHMS*.

	RL (A2C)	EA (ES/GA)	NEAT
Op. Forward	33K	33K	0.1K
Op. Backward	32K	0	0
Local Memory	268K (B)	132K (B)	0.4K (B)

*Estimates for the performed operations (Op.) and required memory when experimenting across a suite of OpenAI environments.

III. ALGORITHMIC PROFILING FOR RL AND NEAT

In order to identify HW acceleration opportunity, we profile the algorithmic performance (i.e., convergence), runtime, and underlying neural network complexity (number of nodes and connections) of NEAT [25] and contrast it against two popular RL algorithms, A2C (Advantage Actor Critic) [28] and PPO2 (Proximal Policy Optimization) [37], with open-source implementation [19]. Table IV contrasts the compute and memory overhead of NEAT vs A2C and traditional EA methods.

A. Profiling of RLs

We compose the RL algorithms with two configurations for the underlying actor and critic networks: *Small* with two layers of MLPs with 64 nodes each and *Large* with three layers of MLPs with 256 nodes each, and run across 6 different tasks (environments) in Open AI gym [5]. For each of the tasks, we set a required fitness value and the algorithm stops when the fitness is achieved, or when it converges.

Algorithmic performance. Since different tasks have different required fitness, we normalized the achieved fitness to the range of [0, 1] in Fig. 2(a-c). When the algorithm achieves 1.0, it means it finishes the task. We observe that PPO2-small completes more tasks than A2C-small. Increasing the size can potentially increase the learning capacity, however requiring more training time to converge. PPO2-large achieves more tasks over PPO2-small with the cost of larger runtime.

Runtime. The RL runtime can be roughly divided into two part Forward (or called predict) and Training (including the backpropagation and the rule updates of RL algorithms). Fig. 3 shows the time profiling of the two RLs with small and large networks, they show that the Training part accounts more percentage of the runtime, which is around 60%.

Network complexity. Finally, we show the number of nodes and connections of the small and large network that we use in RLs in Table V.

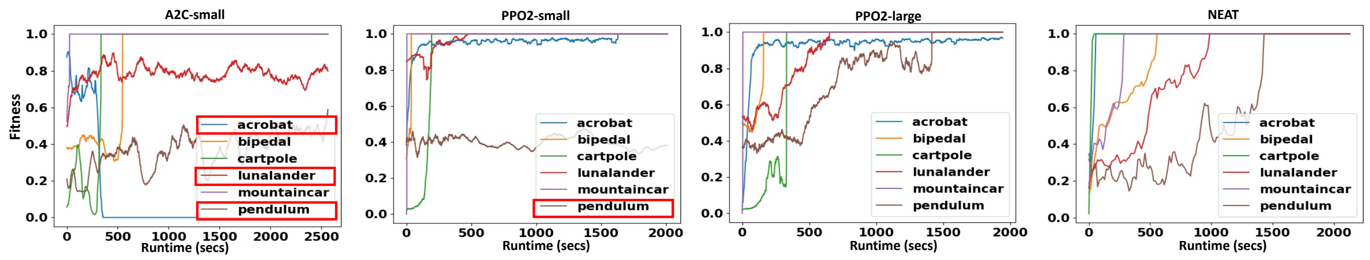


Fig. 2. The achieved fitness trace across runtime of (a) A2C-small, (b) PPO2-small, (c) PPO2-large and (c) NEAT. The red box are the tasks that did not reach or converge to near required fitness values.

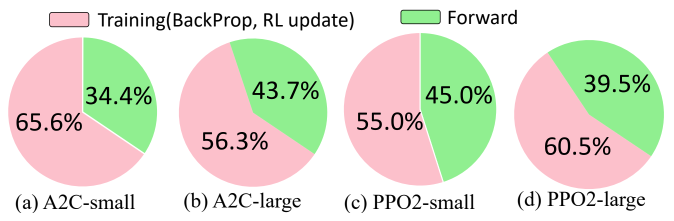


Fig. 3. The time profiling of two RL algorithms will small and large network: (a) A2C-small, (b) A2C-large, (c) PPO2-small, and (d) PPO2-large.

TABLE V
THE COMPARISONS OF NETWORK COMPLEXITY (NUMBER OF NODES AND CONNECTIONS) OF SMALL, LARGE NETWORKS USED IN RLS, AND EVOLVED NETWORKS IN NEAT .

# of nodes/connects		Acrobat	Bipedal	Cartpole	Lander	Mount.Car	Pendulum
Small	Nodes	137	156	133	140	133	132
	Connections	4,672	5,888	4,416	4,864	4,416	4,352
Large	Nodes	5,443	6,660	5,185	5,636	5,187	5,121
	Connections	1,327,872	1,639,424	1,261,824	1,377,280	1,262,336	1,245,440
NEAT	Ave. nodes	10.8	32.2	6.3	13.6	5.7	5.2
	Ave. connections	16.4	79.8	3.6	17.0	5.0	19.2

B. Profiling of NEAT

Algorithmic performance. In Fig. 2(d), we show that NEAT can achieve the required fitness under the set runtime constraint for all the environments we tested against.

Runtime. As shown in Table III, NEAT primarily comprises two parts, "evolve" and "evaluate" , which are analogous to Training and Forward in the RLs. From Fig. 1(b), we can see that NEAT has a drastically different timing profile from the RLs. The "evolve" (the counterpart of Training in RLs, which takes 60% of runtime)) only takes 3% of the runtime in NEAT; instead, "evaluate" occupies the majority of the time. NEAT has a light "evolve", and to compensate for it, includes a much heavier "evaluate". This is because "evaluate" is called for *all* networks in the population; meanwhile "evolve" is only called once when all the entire population are evaluated. This observation motivates this work. By accelerating the "evaluate" part in HW, we could potentially have both "evolve" and "evaluate" light and fast. On the other hand, accelerating Forward of RLs offers less margin for performance gain as it would be limited by Training which is expensive to accelerate.

Network complexity. From Table V, we observe that NEAT evolves fairly small networks and still achieves compatible performance with RL (as Fig. 2 demonstrates), compared to the MLP policy networks we use in RL experiments. This is because "evolve" inherently incorporates a pruning process. However, if we initiate RLs directly with a small network with similar complexity as the ones in NEAT, the RLs were unable to converge, consistent with the lottery ticket hypothesis [12][2].

Summary. *NEAT can find and train neural networks with lower complexity but compatible performance comparing to RLs*, making it a promising learning algorithm for edge devices. NEAT trades of the expensive Training step in RLs with more "evaluate" (i.e., inference) steps which can become the performance bottleneck. To this end, we propose to accelerate NEAT in a HW/SW co-design scheme. Our runtime profiling results drive our partition of workload: we keep "evolve" on the CPU, and offload the "evaluate" to a FPGA accelerator. A key challenge for the accelerator is that the NNs evolved by NEAT are highly irregular, as we discuss next.

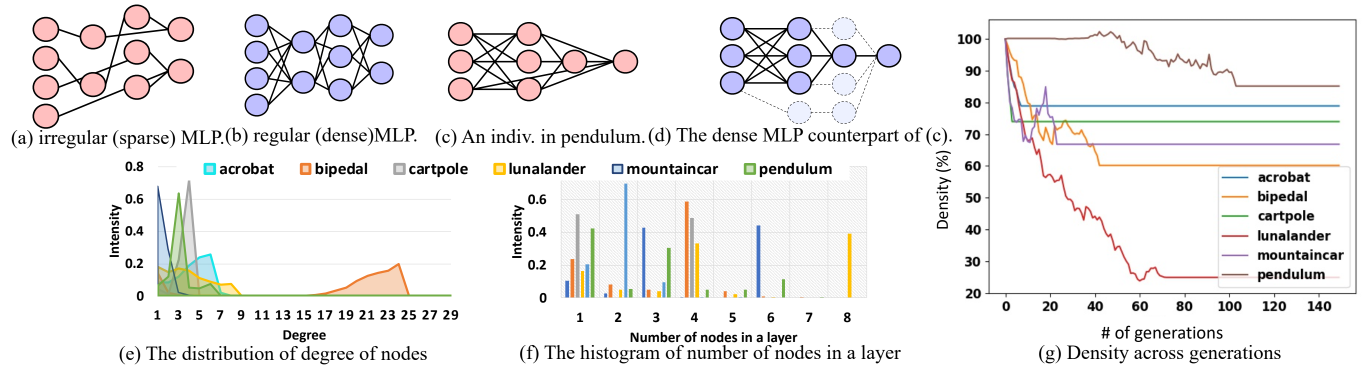
IV. E3-INAX

We propose a HW/SW co-designed platform called Eval-Evol-Engine (E3) with a novel accelerator called INAX for irregular NNs, enabling on-device learning with low latency and high energy efficiency.

A. Challenge: Irregular and Sparse NNs.

Though "evaluate" is fundamentally NN inference, accelerating it in HW is not trivial because of a fundamental challenge: unlike modern hand-designed NNs that have a regular layer-by-layer connectivity, as shown in Fig. 4(b), the networks generated via mutation and crossover can have arbitrary sparse connectivity between neurons, resulting in irregular NNs. In other words, connections can span across layers, as shown in Fig. 4(a)(c). The irregular link across layer incurs dummy node padding in standard DNN accelerators, as shown in Fig. 4(d), causing their inefficiency. Fig. 4(g) shows the trace of density change along generation in a suite of OpenAI env. The trace indicates the dynamic change in sparsity of the NNs that any accelerator would need to handle. Fig. 4(e) and (f) show the distribution of the degree of the node and the number of nodes in each layer across all generations in the evolved NNs, which emphasizes the irregularity of NNs. These make the evolved NNs different from NNs being designed today with pruning which, though sparse, still retain regular layer to layer connectivity. Thus off-the-shelf NN accelerators like systolic arrays [20] that optimize for dense matrix-matrix multiplications, and sparse accelerators [49], [33], [34] that still try to keep model weights "stationary" to exploit reuse cannot directly work. In this work, we design an accelerator for

[2]Pruning from large network to small network is always better than training a small network.



(a) irregular (sparse) MLP. (b) regular (dense)MLP. (c) An indiv. in pendulum. (d) The dense MLP counterpart of (c).

(e) The distribution of degree of nodes (f) The histogram of number of nodes in a layer (g) Density across generations

Fig. 4. The example topology of (a) irregular (sparse) MLP and (b) regular (dense) MLP. (c) An example of individual of pendulum at the 50th generation, where density exceed 100% and (d) its dense MLP counterpart †.(e) The distribution of degree of nodes and (f) the statistics histogram of number of node in a layer of INAX across a suite of OpenAI Env. (g) The density * of the population across generations of NEAT in a suite OpenAI environments. The lines stop when the solutions converge.

†The transparent nodes are dummy nodes needed for (d) to owns equivalent calculations as in (c) when executing in regular accelerator (e.g. Systolic array).

* Density: (# of conns in evolved NN)/(# of conns in dense MLP counterpart). E.g. in (a), Density=(9/18).

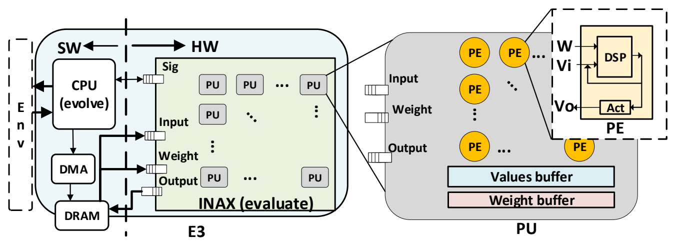


Fig. 5. The architecture of E3.

irregular NNs and get performance by exploiting parallelism within and across genomes, as we discuss later in this section.

B. Eval-Evol-Engine (E3)

1) Architecture: The HW/SW division of E3 is shown in Fig. 5. E3 is partitioned into SW and HW part, and we run SW on the CPU as a master and HW on the FPGA as a slave. The "evolve" portion sits in the SW, while INAX runs the "evaluate" portion in the HW division. The data are passed between them by Direct-Memory-Access (DMA). DMA, as a slave of CPU, moves data in and out between INAX and DRAM via input, weight, and output channels. CPU and INAX pass the sync signal of "start" and "done" via sig channel. "Env" is an OpenAI environment program running on the CPU, which is the benchmark our system is interacting with.

2) Dataflow: E3 implements the NEAT algorithmic flow, shown earlier in Fig. 1(a). In the "evaluate" phase, CPU sends a "start" signal via sig channel. CPU passes two groups of data to INAX through DMA, which are the NN configurations of an populations via weight channel and the input data gathered from "env" via input channel. After inferences are completed, INAX passes the result back to CPU via output channel and send a "done" signal via sig channel. Then, CPU uses the results to interact with "env", gathers the "env" feedback, sends them to INAX as the next input values, and finally sends a "start" signal. INAX then inferences again. This loop iterates until the "env" terminate and fitness values are calculated. Afterward, E3 switches to "evolve" phase and CPU takes over the job to mutate, crossover, speciate, and create NNs. INAX and CPU cooperate to proceed "evaluate" and "evolve" multiple times until the desired fitness value is achieved.

C. Irregular Network Accelerator (INAX)

1) Architecture: As shown in Fig. 5, INAX has three in channels: for weight, input and signal and one out channel. In INAX, there are two major building blocks - Processing Unit (PU) and Processing Element (PE). INAX is composed of a cluster of distributed PUs working independently. A PU itself is composed of a cluster of distributed PEs. PUs and PEs are designed for different levels of parallelism - across individual NNs in a population, and across nodes within a NN, respectively, which we will discuss in Sec. V. INAX has a central controller of the PUs array, which syncs up the HW status with CPU via sig channel.

2) Dataflow: INAX accounts for "evaluate", which we break down to two phases: set-up phase and compute phase. In the set-up phase, INAX receives a batch of individuals (NN configurations) from weight channel. The batch size of individuals is controlled to match the number of PUs in the accelerator. INAX's controller dispatches the individuals to their assigned PUs. According to the individuals' NN configuration, each PU configures an NN topology ready for inference inside itself. Afterward, INAX enters compute phase. It receives input values from the input channel and scatters them to their responsible PUs. Receiving the input values, each PU utilizes its own NN to run inference and outputs the result values. INAX gathers the result values and delivers to the CPU. Then one iteration of the procedure completes.

It is worth noting that each PU reuses the same NN to execute inference on a series of input values, which is typical in RL environments till the "env" terminates (e.g., game ends).

D. Processing Unit (PU) in INAX

1) Architecture: Each PU is responsible for running the full "evaluate" for an individual of the population. A PU comprises of a cluster of PEs (Sec. IV-E) for parallel computing of different nodes within the NN, a weight buffer and a value buffer.

Weight buffer. The weight buffer restores the NN configuration, including the description of network topology, the bias value and activation function of each node, and the weight values of each connection. In the common case of

MLP, weights are not reused, and hence it is less helpful to keep them in local memory. However, the weight buffer is a specialized HW design for accelerator in the system interacting with RL environments, where NNs are reused throughout the "env". It gives us the reuse opportunity of the weight values. Hence, we keep weights locally.

Value buffer. The value buffer restores all the intermediate activations, which is a special requirement for irregular NN, because the intermediate activations could be used by all the subsequent layers.

2) Dataflow: In the set-up phase, PU decodes the data received from weight channel into NN configurations (including topology, activation functions, and bias) and weights. Both of them are kept in the weight buffer. Then, PU configures a network topology based on the decoded NN configuration. In the evaluate phase, PU parallelizes the computations across nodes of the NN.

E. Processing Element (PE) in INAX

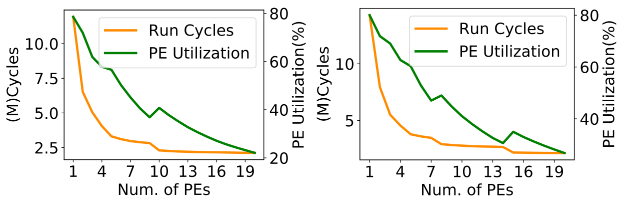
1) Architecture: Each PE is composed of a DSP and an activation function unit. DSP is responsible for MAC on the weight and inputs, and addition on a bias value. The components are executed in a pipeline manner.

2) Dataflow: We design an output-stationary dataflow [6], where each PE is responsible for computing the output of an entire node. The partial sum of MAC result is accumulated locally, over time. The PE sends the accumulated summed value to an internal activation function unit. The result output is then stored locally in the value buffer.

Our choice of dataflow is based on the following analysis. Evolution generates irregular feed-forward MLP NNs, which formulates the topology like Fig. 4(c), where each node can have connections with nodes to any nodes before it. We follow the definition in [6], which categorized the dataflow into three main types: input stationary (IS), weight stationary (WS), and output stationary (OS). The stationary implies which data we keep locally and exploit their reuse opportunity. To begin with, since an MLP has no weight reuse opportunity, WS is not effective. IS requires a partial sum adder and a local buffer for each egress nodes. However, when the network is irregular, the worst case of number of egress nodes is the number of total nodes in NN. This dataflow is HW-unfriendly, since HW needs to meet the worst case, leading to resources over-provisioning at most of the life time. In contrast, we find OS is more HW-friendly in terms of resource requirement on the problem of irregular NN. It accumulates the partial sum locally, and hence prevents over-provisioning. It is important to note though that the time taken to compute each output can be variable at each PE, depending on the node size. We discuss this next.

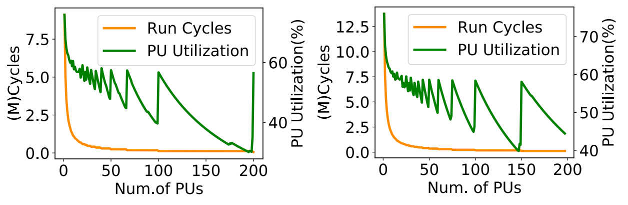
V. Parallelism Strategies in E3

INAX supports two levels of parallelism. Nodes within an evolved NN can be run in parallel across PEs, while individual NNs in a population can be run in parallel across PUs. We discuss the design time problem of choosing HW configurations (number of PUs and PEs) that can achieve higher HW efficiency (utilization rate) next However, the dynamic



(a) Num. of output nodes=10 (b) Num. of output nodes=15

Fig. 6. Parallelism across PEs.



(a) Num. of Individuals=200 (b) Num. of Individuals=300

Fig. 7. Parallelism across PUs.

characteristics in Fig. 4(e) and the fluctuations in Fig. 4(g) also infers the difficulty to find a heuristic for HW configurations that provide good parallelism across generations. In this study, we are also providing two practical heuristics through analysis.

Definition of HW efficiency (Utilization rate). Since parallelism comes at the cost of more resource provisioning, we evaluate the efficiency of each parallelism strategy on resource (r) by its utilization rate $(U(r))$:

$$U(r) = T_{active}(r)/T_{total}(r), \tag{1}$$

where $T_{total}(r)$ is the total running time of all provided resources (r), and $T_{active}(r)$ is the total running time of the active (functioning) resources.

A. Parallelism Across PE

Inside one PU, we can put in multiple PEs to increase the parallelism. More PEs bring low latency, but a higher chance of lower utilization rate. When designing PEs parallelism, to prevent over-provisioning, the number of PEs should not exceed the number of independent nodes, which can be inferred by the number of nodes in each layer. This implies that a good PE assignments relies on prior-knowledge of the deployed NNs, which is challenging in our use case because of three issues: 1) dynamic network topology, 2) PEs alignment, and 3) synchronization.

1) Dynamic network topology: NN topologies could have high variance both within and across populations.

2) PEs alignment: Since INAX owns an output-stationary dataflow, assuming we have n PEs and m nodes in certain layer, we need $\lceil m/n \rceil$ iterations to complete computation of this layer. It implies the nonalignment (non-divisible) between NN nodes (m) and PEs (n) leads to under-utilized iteration.

3) Synchronization: For the output stationary dataflow, the execution time of a PE depends on the number of ingress connections. However, an irregular NN will cause variable number of ingress connections, and hence variable execution time of PEs across a PU. However, to preserve the correctness of a feed-forward NN calculation, we need to keep the PEs

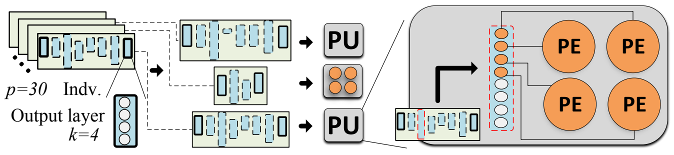


Fig. 8. Example of parallelism across both PU and PE, with 3 PUs in INAX and 4 PEs per PU. Therefore, 3 individuals are parallel computed by 3 PUs. In each PU, 4 nodes of the sitting individual are parallel computed by 4 PEs.

synchronized, before the next layer computation begins. This causes some PEs to remain idle, waiting for others.

Proposed Heuristic. Fig. 4 (f) shows the statistics of layer with different number of nodes, which infers the parallelism opportunity across PE. The variety of number of nodes in a layer across time increase the difficulty to compose a strategy for PEs configuration. We propose a heuristic with analysis, next. Even though the NN topologies in E3 are dynamic, the shape of their input and output layers are constant throughout the generations. They conform with the size of input data and the size of output space (action space) respectively. Therefore it gives us opportunity to optimize the PEs assignment based on those two layers. We chose to size the PU based on the number of nodes in the output layer because of the chosen OS dataflow.

Assuming the output layer has k nodes, to preserve good utilization rate, 1) the heuristic is to assign k PEs to each PU cluster. Moreover, 2) if we are more resource-restricted, we can also use $\lceil k/2 \rceil$, $\lceil k/3 \rceil$, and so on, number of PEs, increasing the number of iterations.

Evaluation. We show the runtime and PE utilization across different number of PE in a cluster in Fig. 6.[3] It is obvious that the runtime decreases with the increase of number of PEs, since we are providing more computing power. In most case, PE utilization rate, $U(PE)$, decreases with the increase of PEs, which is expected because of the thee above mentioned issue. However, we can notice some local peak points on the $U(PE)$ curve, which has better $U(PE)$ locally. Fig. 6(a) has 10 output nodes and it has a local peak at the parallelism number of 10. Likewise for 15 output nodes in Fig. 6(b). Moreover, we can notice some smaller local peak at the parallelism points of $\lceil k/2 \rceil$, $\lceil k/3 \rceil$, and so on, where k is 10 and 15 respectively.

B. Parallelization Across PU

While a cluster of PEs exploit parallelism of independent nodes, a cluster of PUs exploits parallelism of individuals. When designing number of parallelism of PUs, it is without a doubt that if we put more PUs in HW, the runtime will decrease. However, with the chance of lower PU utilization rate. To keep good PU utilization rate, $U(PU)$, there are two challenges: 1) synchronization issue from variance of 1) NNs and 2) "env".

1) Synchronization issue from variance of NNs: Since each PU deals with different individuals with different NN topology, their inference latency is different. When synchronising, it could lead to some PUs, dealing with complex network, lagging the finishing time of the whole cluster, while other PUs are waiting.

2) Synchronization issue from variance of "env": Different individuals react differently to the "env", and hence their lifetime in the "env" is different. For example, some bad performance individuals can fail, terminate early, and stay idle while the other populations are still running.

Proposed Heuristic. There are still heuristic we could follow for better PU utilization. The number of populations, p, inside E3 is an algorithm parameter, predefined before the start of NE. Hence, we could optimize upon it. 1) We can assign p PUs to exploit the maximum PUs parallelism. However, 2) if we are more resource-restricted, we can use a fraction as well. Since population size in NEs is generally large, we may not be able to provide full PU parallelism due to resource constraints.

Evaluation. From the experiment of runtime and PU utilization across different number of PU,[3] as shown in Fig. 7, we could find local peak points on the $U(PU)$ curve. As shown in Fig. 7(a), where we have 200 populations, the peak points are at 200, 100, 67, 50, and so on, which correspond to PUs number of $\lceil p/2 \rceil$, $\lceil p/3 \rceil$, $\lceil p/4 \rceil$, $\lceil p/5 \rceil$, and so on. This validates our heuristic. To illustrate this, we take $p/2$ (100 PUs) case as an example. It finishes in two iterations. In contrast, if we choose to use 99 PUs, it requires three iterations, while 98% of PUs are idle in the last iteration.

Parallelism Across PU and PE We could also parallelize across PU and PE together, by following the heuristics for PUs and PEs simultaneously. An illustrative example is shown in Fig. 8. Since PE is the basic build block of INAX, we use $U(PE)$ as our criterion. We do not plot quantitative results in the interest of space, but we observed the response surface of $U(PE)$ following the expected behaviors we described above.

VI. Evaluation

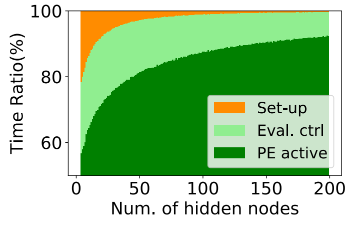
A. Setup

We ran multiple environments provided in OpenAI [5], using a mix of control benchmarks and Atari games[4]. We experimented and compared the performance of three settings: 1) SW-based NE (E3-CPU), where all the functions ran in the CPU, 2) E3-INAX, where "evolve" ran on the CPU, and "evaluate" ran on INAX, and 3) a GPU implementation (E3-GPU), where "evolve" ran on the CPU and "evaluate" ran on the GPU. For all three settings, "env" is another program on the CPU. NEAT algorithm is generally not efficient on GPUs [36], because of small batch size and dynamic topology. However, we implemented E3-GPU as a reference comparison.

Configuration: The SW program was run on the desktop with 8th generation intel i7 CPU (@2.3GHz) and nVidia GTX 1080 GPU. The INAX HW was implemented and ran in Xilinx ZCU104 board with Zynq UltraScale+ XCZU7EV device (16nm). We measured the power of CPU by Intel power gadget, of GPU by nvidia-smi utility, and of FPGA by Vivado post-routing power analysis.

[3]If not specified, the default parameters are: num individuals:200, num inputs:8, num outputs:4, num hidden nodes: 30, sparsity rate:0.2, num PU:1, num PE:1

[4]Env1:cartpole, Env2:Acrobot, Env3:moutain car, Env4:bipedal, Env5:lunar lander, Env6:pendulum

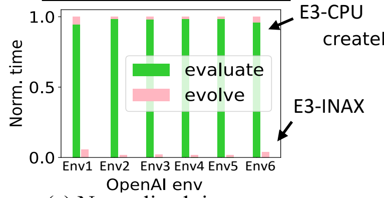


(a) Runtime analysis

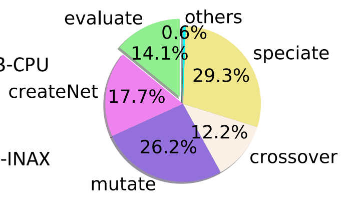
	E3-CPU	E3-GPU	E3-INAX
Env1	**0.3 (s)**	**11.7**	**0.02**
Env2	**43.3**	**1,472**	**0.7**
Env3	**115.4**	**2,971**	**2.4**
Env4	**164.9**	**3,877**	**2.8**
Env5	**220.1**	**3,242**	**3.4**
Env6	**527.0**	**9,749**	**20.9**

(b) Experiment runtime results

	E3-CPU	E3-GPU	E3-INAX
Ave.	**1**	**18.0**	**0.03**



(c) Normalized time



(d) E3 timing profile

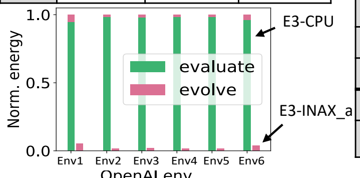
Fig. 9. (a) The runtime analysis of INAX. The experiment results across a suite of OpenAI env: (b) runtime comparisons of three platforms, (c) their normalized runtime and their function time breakdown (E3-GPU is too large to be displayed in this figure). (d) E3 timing profile.

TABLE VI
COMPARISONS WITH CONTINUOUS LEARNING ACCELERATORS.

	FA3C [9]	PPO-FPGA [29]	CLAN [27]	GeneSys [36]	E3
algorithm	(RL) A3C (*Need BP*)	(RL) PPO (*Need BP*)	(EA) NEAT (*No BP*)	(EA) NEAT (*No BP*)	(EA) NEAT (*No BP*)
Internal NN	Regular Fixed NN (Manual)	Regular Fixed NN (Manual)	Irregular Evolved NN (**Autonomous**)	Irregular Evolved NN (**Autonomous**)	Irregular Evolved NN (**Autonomous**)
Platform	**FPGA**	**FPGA**	**CPU**	**ASIC**	**FPGA**
Scheme	**HW/SW codesign** Inference: FPGA BP: FPGA Env: CPU	**HW/SW codesign** Inference: FPGA BP: FPGA Env: CPU	**Distributed learning** Inference: CPU Evolve: CPU Env: CPU	**HW acceleration** Inference: ASIC Evolve: ASIC Env: CPU	**HW/SW codesign *** Inference: FPGA Evolve: CPU Env: CPU
Memory overhead	Large DRAM Large on-chip SRAM	Large DRAM Large on-chip SRAM	**Small** DRAM **Small** on-chip SRAM	**Small** DRAM **Small** on-chip SRAM	**Small** DRAM **Small** on-chip SRAM
Inference accelerator	**Standard systolic array** • Designed data layout for BP. • Designed data layout for systolic array interface. • Efficient for **regular** NN.	**PE array** • Designed data layout for BP. • Efficient for **regular** NN.	**CPUs** • Distribute the computations of both training and evolutions to a cluster of Raspberry Pis	**Standard systolic array** • Need special *input-data-alignment phase* for standard systolic array interface. • Inefficient for irregular NN.	**INAX** (dynamic accelerator) • Efficient for both **regular** and **irregular** NN. • Layer-level parallelism (across PE) • Population-level parallelism (across PU)

* Identify Inference dominates 97% runtime while Evolves takes only 3%. After HW/SW codesign opt., Inference achieves 30X speedup. The runtime of Inference is balanced to the same scale as Evolve.

	E3-CPU	E3-GPU	E3-INAX_a
Ave.	**1**	**71.6**	**0.03**



(a) Normalized energy

Config (PU, PE)	E3-INAX_a (50, 4)*	E3-INAX_b (64, 16)
DSP	400 (24%)	1,024 (60%)
LUT	6,173 (3%)	13,935 (6%)
FF	8,964 (2%)	20,358 (5%)
Freq.	100MHz	100MHz
Power	0.852 W	1.216 W

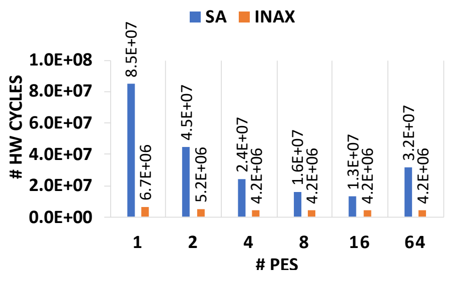
(b) Resource utilization

Fig. 10. (a) The normalized energy expense of three platforms and the breakdown of energy consumption in their functions (E3-GPU is too large to be displayed in the figure). (b) The resource utilization in FPGA of two set of configs

**PE number slightly change according to the pre-defined* output nodes *of different Env. The number of output node (the PE number we use): cartpole:3, acrobar:3, mountain car:3, bipedal:4, lunar lander:4, pendulum:1.*

B. Runtime analysis in INAX

We first evaluate the resource utilization efficiency of INAX by its runtime breakdown of two HW phases: set-up and compute phase, where the compute phase is detailed into time of "PE active" and "evaluate control". The "evaluate control" includes the total PE under-utilization time and miscellaneous control time, including pipeline overhead and reading and writing of intermediate values. We show the normalized runtime of INAX across different number of hidden nodes, which implies the size of the network in Fig. 9(a)[3]. After normalization, the ratio of the PE active time is the $U(PE)$ that we defined earlier in Sec. V, which also reflects the control overhead



(a) Required HW cycles

# PE	Speedup ($\frac{SA}{INAX}$)
1	**12.6**
2	**8.6**
4	**5.6**
8	**3.7**
16	**3.0**
64	**7.5**

(b) Speedup

Fig. 11. (a) The averaged number of required HW cycles (in a suite of OpenAI env: Env1-Env7) with different accelerator structures: Systolic Array (SA) and INAX, across various number of PEs inside accelerators, and (b) the speedups.

(set-up, evaluate control) when offloading to HW. The higher the computation intensity (number of hidden nodes), the more control overhead is hided, and the higher the $U(PE)$.

C. Acceleration Benefit

The chosen gradient-free algorithm, NEAT, has the edge-device friendly characteristics of low memory requirement as discussed in Table I and Table IV, with the trade-off of huge amount of evaluation computations. It leads to a runtime bottleneck as described in Fig. 1(b), which makes accelerating the evaluation a critical need. and we discuss it next. In algorithm level, we set the number of population=200, mutation and crossover rate=0.5, and start with no hidden nodes. The INAX configuration is following the PE and PU parallelism heuristics in Sec. V, where we picked PE=*output nodes* and PU=50.

Fig. 9(b) shows the runtime of E3-CPU (the baseline), E3-GPU (GPU acceleration), and E3-INAX (HW acceleration). The irregularity and sparsity makes E3-GPU inefficient and ends up requiring longer runtime than the baseline. However, E3-INAX, housing an irregular network accelerator, can provide averaged **30× speedup** over the baseline. Fig. 9(c) shows how the main bottleneck function, "evaluate", in the baseline system is reduced in the E3-INAX system. After speedup, the time for "evaluate" in E3 decreases to the same scale of the time for other sub-function inside "evolve". E3 has a more balanced time distribution among each function, as shown in Fig. 9(d), as a contrast comparison with the original timing profile on E3-CPU in Fig. 1(b).

D. Energy Benefit

In the experiments, we evaluate the total energy expense to complete the tasks. The results are summarized in Fig. 10(a). E3-GPU consumes 71x more energy than E3-CPU, because of its longer runtime and higher power. However, E3-INAX can **reduce the energy consumption by 97%** comparing with the baseline (E3-CPU).

E. FPGA utilization

The averaged FPGA resource utilization in the experiments are shown as config $E3\_a$ in Fig. 10(b). We could also introduce more resources for lower latency but higher chance of under-utilization and higher energy such as $E3\_b$ in Fig. 10(b).

F. Effectiveness of INAX over GeneSys

We show the effectiveness of the dynamic characteristics of INAX by contrasting with Systolic Array (SA) accelerator structure used in GeneSys [36]. Since SA can also exploit PU level of parallelism, we implement a PU-parallelized SA for fair comparisons. The underlying SA is a 1-D systolic array since we are executing MLP-type calculation. Across the experiments, we use the same PU=50 as in Sec. VI-C. We also experiment on different number of PEs in the underlying accelerator. Fig. 11(a) shows averaged number of required HW cycles (in a suite of OpenAI Env) of these two accelerators with different number of PEs. SA's performance is inferior to INAX because of two reasons. **First**, the inherent sparsity of NN incurs zero filling in SA, which decreases efficiency. **Second**, the irregular link across layer incurs dummy node padding as shown in Fig. 4(d), where SA needs to fetch the node in previous layer and align them with nodes in the current layer, leading to more execution time. In Fig. 11(a), we can also observe the effectiveness of the proposed heuristic in Sec. V-A, where we choose PEs according to number of output nodes for higher PE utilization rate. Over-providing number of PEs (8, 16, 64) will only result in more idle PEs but not runtime benefit. On the other hand, SA, as a regular accelerator, requires more PEs because of the dummy nodes padding effect as mentioned above in the second point. SA has the best performance at 16 PEs; however, it is still 3x slower than INAX. Due to the flexible structure of INAX (Sec. IV-C), it provides 3× to 12.6× speedup compared to SA.

VII. Related-works

DNN Inference Accelerators. Many inference accelerator design focus on analyzing and optimizing the dataflow of ML workload. The dataflow comprises the computation order,parallelization-strategy, and tiling strategy employed by the accelerator. There are specialized HW design in the area of ASIC, such as Eyeriss[6], ShiDianNao [9], NVDLA [1], FlexFlow[22], and C-Brain[41]. Also, many FPGA-based accelerator, featuring its flexibility, were proposed, such as Caffeine [48], Cnp [10], and others [23], [11], [47], [39], [50]. However, they are targeting the conventional structured regular NN layer, especially convolution layer, but cannot deal with irregular NN. Many recent DNN accelerators also tackle sparse inference efficiently, such as SCNN [33] and Cambricon-X [49]. However, they still operate on layer-by-layer sparse DNNs, and not for irregular NNs generated by NEAT as shown in Fig. 4(a)(c). Irregular NNs also have activation sparsity, which we did not investigate in this study and is ripe for future work.

DNN Training Accelerators. Many modern DNN models are trained on GPUs. However, in the RL tasks, GPUs are generally not computation-efficient and energy-efficient[43], [7], which is not friendly for edge device[36]. Cloud TPUs[20] are specialized for training, but require structured NNs for efficient computing, which is challenging in irregular NN case. However, in the RL tasks, they are generally not as efficient as in other DNN tasks such as supervised learning, because of the frequent interaction with RL env and small batch size[43], [7]. Moreover, GPUs and cloud TPU are not suitable for edge device due to the high power consumption. Furthermore, both of them are optimized for accelerating structured regular NN, who becomes inefficient in irregular case.

Continuous Learning Accelerators. Table VI summarizes contrast between these continuous learning accelators. FA3C [7] builds an FPGA-based platform for an DRL algorithm, A3C [29]. PPO-FPGA [26] accelerates an DRL algorithm, PPO [38], with HW/SW co-design manner on FPGA. However, the BP step costs more buffer and high demand of resources owing to the need of high complexity calculation, which could become bottleneck when the NN scales up. CLAN [24] utilizes Neuroevolution as underlying algorithm for a distributed learning system on edge CPU, which is not the focus of this paper. GeneSys [36] is an ASIC Neuroevolution accelerator. However, GeneSys [36] used standard accelerator for inference ("evaluate"), where irregular NN are executed inefficiently. However, in this paper, we identify "evaluate" is actually the bottleneck of NE-based algorithm, hence needing a specialized irregular network accelerator for low latency, and this work fulfills the need.

VIII. Conclusion

For continuous edge-learning, we identify a gradient-free neuro-evolution algorithm, NEAT, as a good candidate for its low memory overhead and no backpropogation, however, with the trade-off of huge amount of evaluation. Unfortunately, the NNs to be evaluated are often irregular, which makes off-the-shelf inference accelerators not suitable. In this work, we designed a specialized irregular NN accelerator, INAX, integrated into an autonomous continuous edge-learning FPGA prototype, E3, that uses HW/SW co-design to accelerate the algorithm and balance workload between CPU and INAX. E3 achieves 30× speedup and 97% energy reduction across a suite of OpenAI benchmarks. This work provides a promising path toward the envisioning of ubiquitous edge learning.

Acknowledgement

This work was supported in part by NSF Award 1909900 and a Google Faculty Award. We thank Ananda Samajdar for the preliminary discussions on this idea and motivating us to pursue this research direction.

References

[1] "Nvdla deep learning accelerator," http://nvdla.org, 2017.

[2] "Google speech to text," https://cloud.google.com/speech-to-text, 2020.

[3] A. Ahmad and L. Dey, "A k-mean clustering algorithm for mixed numeric and categorical data," *Data & Knowledge Engineering*, vol. 63, no. 2, pp. 503–527, 2007.

[4] T. L. Bailey and C. Elkan, "Unsupervised learning of multiple motifs in biopolymers using expectation maximization," *Machine learning*, vol. 21, no. 1-2, pp. 51–80, 1995.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[6] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[7] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 499–513.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[9] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[10] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 32–37.

[11] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[12] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[13] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[14] A. Gaier and D. Ha, "Weight agnostic neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 5364–5378.

[15] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, and D. Warde-Farley, "Generative adversarial nets in advances in neural information processing systems (nips)," 2014.

[16] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[17] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3389–3396.

[18] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[19] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[22] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[23] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[24] P. Mannan, A. Samajdar, and T. Krishna, "Clan: Continuous learning using asynchronous neuroevolution on commodity edge devices," *arXiv preprint arXiv:2008.11881*, 2020.

[25] A. McIntyre, M. Kallada, C. G. Miguel, and C. F. da Silva, "neat-python," https://github.com/CodeReclaimers/neat-python.

[26] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 19–27.

[27] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[32] A. Moore, "K-means and hierarchical clustering," 2001.

[33] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[34] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training."

[35] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[36] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 855–866.

[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[39] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[40] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[41] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[42] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, 2002, pp. 569–577.

[43] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[44] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[45] L. Wu, F. Tian, T. Qin, J. Lai, and T.-Y. Liu, "A study of reinforcement learning for neural machine translation," *arXiv preprint arXiv:1808.08866*, 2018.

[46] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Advances in neural information processing systems*, 2017, pp. 5279–5288.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[48] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[49] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[50] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[51] Y. Zhuang, Y. Rui, T. S. Huang, and S. Mehrotra, "Adaptive key frame extraction using unsupervised clustering," in *Proceedings 1998 International Conference on Image Processing. ICIP98 (Cat. No. 98CB36269)*, vol. 1. IEEE, 1998, pp. 866–870.

[52] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.