8-2022

The Islands Project for Managing Populations in Genetic Training of Spiking Neural Networks

Chaohui Zheng
czheng4@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Chaohui Zheng entitled "The Islands Project for Managing Populations in Genetic Training of Spiking Neural Networks." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James S. Plank, Major Professor

We have read this thesis and recommend its acceptance:

Garrett S. Rose, Catherine D. Schuman

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

The Islands Project for Managing Populations in Genetic Training of Spiking Neural Networks

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Chaohui Zheng

August 2022

Abstract

The TENNLab software framework enables researchers to explore spiking neuroprocessors, neuromorphic applications and how they are trained. The centerpiece of training in TENNLab has been a genetic algorithm called Evolutionary Optimization For Neuromorphic System (EONS). EONS optimizes a single population of spiking neural networks, and heretofore, many methods to train with multiple populations have been ad hoc, typically consisting of shell scripts that execute multiple independent EONS jobs, whose results are combined and analyzed in another ad hoc fashion.

The Islands project seeks to manage and manipulate multiple EONS populations in a controlled way. With Islands, one may spawn off independent EONS populations, each of which is an "Island." One may define characteristics of a "stagnated" island, where further optimization is unlikely to improve the fitness of the population on the island. The Island software then allows one to create new islands by combining stagnated islands, or to migrate populations from one island to others, all in an attempt to increase diversity among the populations to improve their fitness.

This thesis describes the software structure of Islands, its interface, and the functionalities that it implements. We then perform a case study with three neuromorphic control applications that demonstrate the wide variety of features of Islands.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The TENNLab group at the University of Tennessee focuses on spiking neural networks (SNN's) as novel, brain-based computational platforms. SNN's are powerful computationally, allowing one to implement complex functionalities with low requirements on size, weight and power. Their difficulty lies in training, as standard backpropagation approaches are difficult to employ, and yield large SNN's whose size, weight and power go against the reasons why we want to explore them [29, 21]. Accordingly, the major training algorithm employed by TENNLab is a genetic algorithm called EONS (Evolutionary Optimization of Neuromorphic Systems) [22].

Each EONS job manages a single population of SNN's. These are typically initialized randomly, and then go through generations of selection and "reproduction." The algorithm optimizes both the structure and parameters of the networks. Typically, an EONS population *stagnates* after a period of time. This means that although EONS may continue to generate new populations, it is highly unlikely that the new populations perform any better than the current ones because EONS has discovered the local optima and gets stuck there. Therefore, one is motivated to employ multiple EONS populations.

The Islands project's goal is to manage multiple EONS populations. Each population is an "island," created by an EONS process. There is a manager process, a monitor process, and a collection of worker processes. The manager instructs the workers to perform EONS jobs, either from initial random populations, or by combining populations from other islands. Additionally, the manager maintains a pool of the best SNN's from the collection of islands, and allows "migrant" SNN's from this pool to join other EONS populations.

In this thesis, we detail the software structure of the Islands project and its interface. We then demonstrate its features with an optimization study involving three control applications. The Islands project is poised both to aid TENNLab researchers as they optimize SNN's for various applications and to be the focus of a research study on the benefits of the Islands approach.

Chapter 2

Related Work

There are many ways to parallelize genetic algorithms. One simple approach is to divide the task of the evaluation process across many CPU cores. This approach is also known as global/master-slave parallelization [2, 7, 5]. The evaluation of one individual is independent of another. Therefore, this approach is usually implemented with "master" and "slave" programs where the "master" program stores the population and distributes the individuals to the "slave" program, and "slave" program evaluates the fitness of individuals and reports the fitness back to "master" program. Additionally, this master-slave can be either synchronous or asynchronous, where the synchronous model waits for all individuals in the population to be evaluated before the reproduction of the next generation, and the asynchronous model only waits for a fraction of the population to be evaluated. Although the asynchronous model makes more efficient use of computational resources due to less CPU idle time, it can lead to a limited search space that is less computationally expensive [28].

Another popular approach is to maintain many subpopulations by running multiple genetic algorithms across numerous processors to solve the same problem. Each processor evolves the subpopulation independently with occasional communication between subpopulations through migration. The migration operation exchanges the individuals among processors. There are usually two parameters involved with the migration process - (a) migration rate, which specifies how many individuals migrate (b) migration interval, which specifies how often migration happens [11, 5]. This approach is also known as the Coarse-Grained/Island Model [35]. These migration parameters can affect the performance/efficiency of the genetic algorithms and can be application-specific [3, 4]. On the other side, there have shown the benefits/promising results of the Island model to optimize problems [19, 34]. In this thesis, we design/implement an Island model within the TENNLab framework described in [18].

We are specifically interested in using genetic algorithms to train spiking neural networks. This is sometimes called neuroevolution [8]. Neuroevolution approaches have been applied to train a variety of neural networks, including recurrent neural networks [32, 9], deep neural networks [12, 33, 37, 36], and spiking neural networks [27, 10]. When a problem is complex, the neuroevolution approach takes a significant amount of time to find an acceptable spiking neural network due to the large population size and high generations/iterations. For that reason, parallelizing the neuroevolution approach is necessary. Various research projects have used the master-slave model to parallelize neuroevolution methods [37, 36, 20, 26].

The NEAT and HyperNEAT projects explore the concept of maintaining diversity in neuroevolution [30, 31]. Rather than evolve a single optimized population, multiple lineages of populations are optimized so that the populations do not become too homogenized. The Islands project allows the manager to manage lineages on its various islands.

Chapter 3

Background

The focus of this research is on genetic algorithm training of spiking neural networks (SNN's). For background material on SNN's, applications, neuroprocessors and training, please see survey papers by Roy *et al* [21] and Schuman *et al* [25], plus overview papers by various TENNLab researchers [18, 22, 15, 24].

TENNLab is a neuromorphic computing research lab at the University of Tennessee, Knoxville. TENNLab has developed a software infrastructure that allows us to train spiking neural networks on different neuromorphic applications, neuroprocessors, encoding and decoding techniques, and training methods [18]. In this work, we rely on the executable produced by the TENNLab framework to implement Island. Each executable is compiled with a given neuroprocessor, application, and EONS. The executable itself is able to parallelize the fitness evaluation of spiking neural networks through multi-threading. This parallelism is limited to the number of cores within one single processor. Figure 3.1 shows the subset of the TENNLab framework used in this thesis. We briefly describe each component below.

1. **Application:** TENNLab supports multiple applications. Each application implements an interface defined by the TENNLab framework, which works in a manner similar to OpenAI Gym [1]. The application presents an agent with observations, and then the agent responds by defining actions

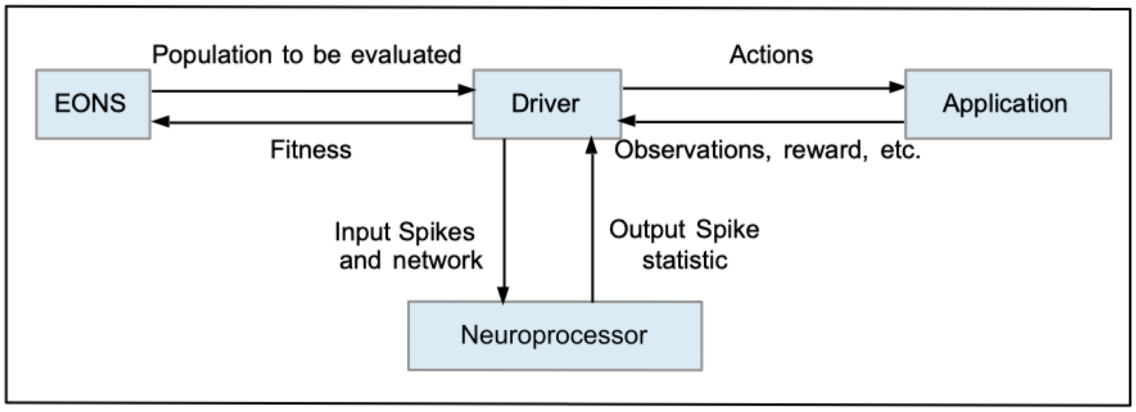


Figure 3.1: A subset of TENNLab framework used in Island

for the application. This cycle repeats until the application is over. The application defines fitness in a manner specific to the application. For example, a classification application may define fitness as the number of correct classifications. A control application may reward a long lifetime, or shooting targets.

2. **Neuroprocessor:** This is what computes on a spiking neural network. The neuroprocessor defines parameters of the network (e.g. discrete or continuous values, features like leak or plasticity), and then allows one to load a network onto the processor, apply input spikes, process, and record the resulting spiking behavior.

3. **EONS:** EONS takes a population of spiking neural networks and their associated fitnesses. Then it runs a genetic algorithm that performs selection, mutation, crossover, and merge operations to generate a new population of networks to be evaluated. EONS considers one population at a time — currently, to manage multiple EONS populations, one typically runs multiple independent EONS jobs.

4. **Driver:** The driver glues all other software components together. It evaluates the current population by obtaining the fitness of each network. It then sends fitness back to EONS to generate the next population. It takes the following steps to evaluate one network:

   (a) Read in observation values from the application and convert these values into input spikes with a selected encoding technique.

   (b) Send input spikes to the neuroprocessor and then wait to receive information about output spikes, such as the spiking time, and spike counts. Then it converts this statistic into actions with a selected decoding technique.

   (c) Send actions to the application.

   (d) Repeat the above steps until the application stops.

To demonstrate our work with Islands, we have chosen specific parts of the TENNLab framework for experimentation. These are:

- The RISP neuroprocessor [16].
- Temporal and "spikes" encoding.
- Winner-take-all decoding
- Three control applications: polebalance, Bowman and SpaceInvaders.

We elaborate further in the following subsections.

3.1 Encoding and Decoding

The encoding and decoding techniques are described in [18, 23]. In this work, we explore the following encoding/decoding JSON specifications used in the TENNLab framework for our experiments.

1. **{ "spikes": { "flip_flop": 2, "max_spikes": 8, "min": 1, "max": 1 } } (encoding):** Each input value corresponds to two input neurons. Assume the input value has an interval of [min, max]. Any value within the interval [min,(min + max)/2] goes into one neuron, and the remaining value goes into another. Then the value is scaled between 0 and 1 corresponding to its bin. Finally, the scaled value is converted to a number of spikes from one to eight and all input spikes have values of 1. Figure 3.2 shows an example of encoding the value 2, in a range of [0,10]. The "flip-flop" attribute means that smaller values in the first bin spike more, while larger values in the second bin spike more. The caption of the example explains further.

2. **{"temporal":{"higher_earlier":true}} (encoding):** Each input value corresponds to one input neuron, and produces one input spike to that neuron, whose time is scaled by the value. A higher input value results in an earlier

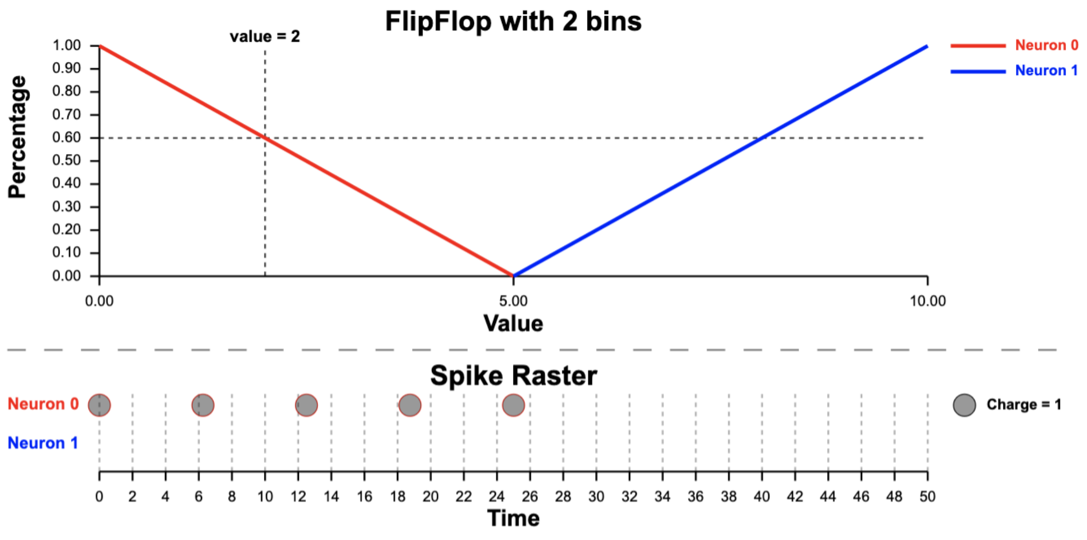


Figure 3.2: An example of spikes-flip-flop encoding with two bins for a value of 2 over an input interval [0,10] for 50 simulation time. First a value of 2 belongs to the first bin. Then we map 2 from [0, 5] to [0, 1], which converts the value to 0.6. In turns, it produces $ceil(0.6 * 8) = 5$ spikes with each spike $50/8 = 6.25$ time units apart. Similarly, a value of 8 will produce the same spikes on the second neuron.

input spike. Figure 3.3 shows an example of temporal encoding for a value of 2.

3. **Winner-take-all:** Each potential value of an action corresponds to one output neuron. The output neuron with the most spike counts wins, and its corresponding action is taken by the application. For example, in the polebalance application, whose actions are "push-left" and "push-right," there are two output neurons, and the neuron that spikes most determines the action. Ties go to the neuron added to the spiking neural network first.

3.2 Reduced Instruction Spiking Processor

The RISP neuroprocessor is a lightweight neuroprocessor that operates on spiking neural networks [16]. RISP stands for "Reduced Instruction Spiking Processor," and its main feature is simplicity. Other TENNLab neuroprocessors, such as Caspian [14], DANNA [6] and DANNA2 [13], support many robust/complex features, such as Spike-timing-dependent plasticity (STDP), refractory period, plasticity and configurable leak. RISP has none of these. RISP only supports configurable neuron threshold, synapse delay, synapse weight, and an optional "full leak" mode. When this "full leak" mode is enabled, all accumulation is lost at the end of the integration cycle. When this "full leak" is disabled, all neurons retain the accumulated potentials. There have been several projects that exploit the simplicity of RISP. For example, the Whetstone project uses backpropagation training methodologies that leverage "full leak" enabled networks as RISP allows [29]. We can also hand build RISP networks that perform binary operations on different sets of input encoders [17].

Figure 3.4 shows an example of RISP networks where neurons A, B, and O each have a threshold of 1. The synapses AO and BO have weights of 0.5, and delays of 1 and 2 respectively. Suppose all three neurons start with a potential of 0, and A and B each spike at time 0. When "full leak" mode is enabled, spikes with weights of 0.5

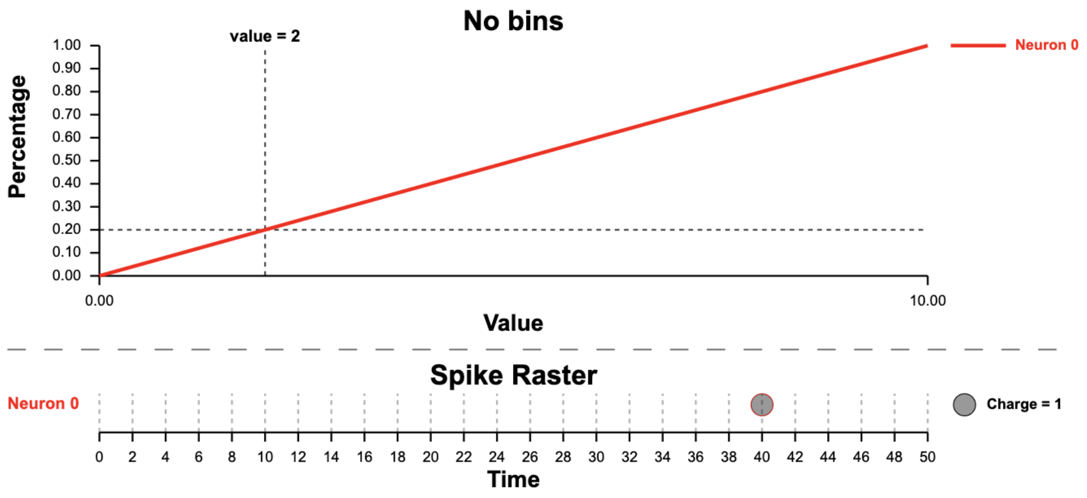


Figure 3.3: An example of temporal encoding for a value of 2 over an input interval [0,10] for 50 simulation time. First we directly map 2 from [0, 10] to [0, 1], which converts the value to 0.2. This produces a spike at time 50 - (0.2 * 50) = 40.

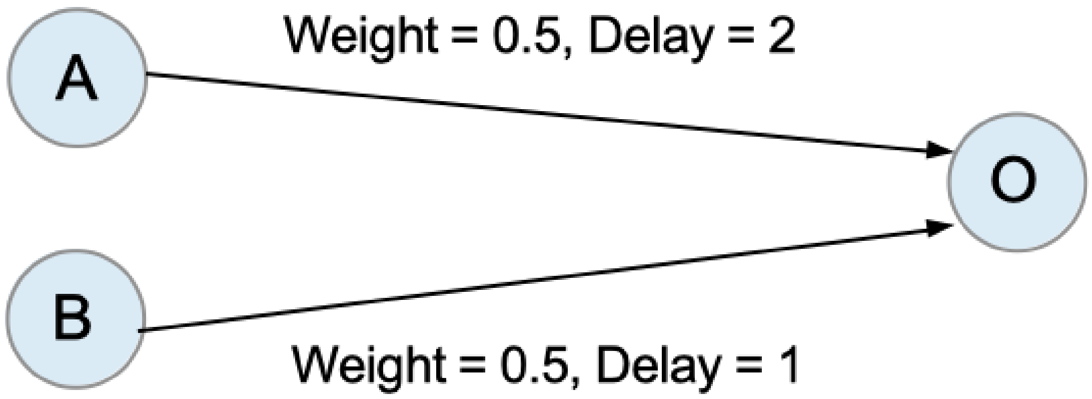


Figure 3.4: An example of a RISP network whose behavior changes significantly with the leak mode.

arrive to neuron O at times 1 and 2, and the neuron leaks away its potential at both timesteps. When “full leak” mode is disabled, neuron O retains its 0.5 potential from timestep 1, and spikes at timestep 2.

In this paper, “full leak” model is disabled. Therefore RISP can maintain some information about the previous states. In theory, this is helpful for training a control application where the future states heavily depend on the previous states, and having neurons retain their potentials is a good thing.

3.3 Applications

In this section, we give a detailed description of three applications that we use to demonstrate Islands: Polebalance, Bowman and Spaceinvaders. These are control applications that have been used in other TENNLab projects [15, 23].

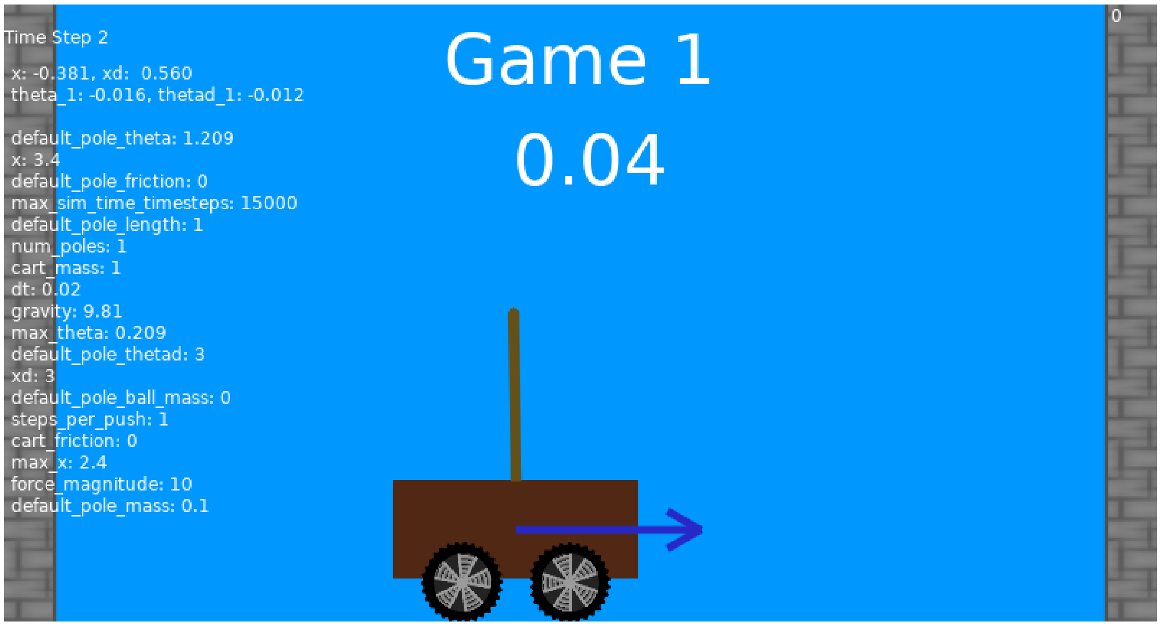
3.3.1 Polebalance

Polebalance is a classic cart-pole control problem. In this problem, there's a pole attached to a cart, which moves on a fixed-size road. The goal of this problem is to prevent the pole from falling over and to prevent the cart from hitting the bounds of the track. The observations of this application are the cart's position on the track, the pole's angle on the cart, the cart's horizontal velocity, and the pole's angular velocity. The actions are to push the cart left or right. The fitness function is shown below. Figure 3.5 shows a screenshot of polebalance application.

$$\frac{number\ of\ timesteaps\ to\ keep\ pole\ balance\ within\ bounds}{maximum\ timesteps}$$

3.3.2 Bowman

Bowman is a LIDAR-based control application [15]. In this problem, birds fly horizontally at different heights and directions across the screen. The player is placed



```
Time Step 2
x: -0.381, xd:  0.560
theta_1: -0.016, thetad_1: -0.012

default_pole_theta: 1.209
x: 3.4
default_pole_friction: 0
max_sim_time_timesteps: 15000
default_pole_length: 1
num_poles: 1
cart_mass: 1
dt: 0.02
gravity: 9.81
max_theta: 0.209
default_pole_thetad: 3
xd: 3
default_pole_ball_mass: 0
steps_per_push: 1
cart_friction: 0
max_x: 2.4
force_magnitude: 10
default_pole_mass: 0.1
```

Figure 3.5: A screenshot of polebalance application. The cart is being pushed to right.

at the bottom center and is equipped with seven LIDAR sensors, a bow and unlimited arrows. The player uses LIDAR readings to make decisions - rotate the bow either to the left or right, shoot an arrow along its current angle of sight, or do nothing. The goal of this problem is to maximize the birds shot while minimizing the number of arrows used. The fitness function is shown below. Figure 3.6 shows a screenshot of bowman application.

$$\frac{(birds\ shot)^2}{birds\ spawned * arrows\ shot}$$

3.3.3 Spaceinvaders

Spaceinvaders is a LIDAR-based control application [15]. In this problem, invaders begin at the top of the screen and move down toward the bottom. The player moves along the bottom and shoots a laser up to kill invaders. The game ends when one of the invaders reaches the bottom of the screen or the play time exceeds the maximum timesteps. The goal of this problem is to maximize the number of invaders eliminated, while minimizing the number of laser shots before invaders reach the bottom of the screen. The fitness function is shown below. Figure 3.7 shows a screenshot of spaceinvaders application.

$$\frac{invaders\ kill}{number\ of\ shots} * \frac{play\ time}{maximum\ timesteps}$$

3.4 Evolutionary Optimization For Neuromorphic System

Evolutionary Optimization For Neuromorphic System (EONS) is a primary training method in the TENNLab framework to design spiking neural networks using genetic algorithms. The detailed implementation of EONS is described in [22]. We also briefly describe EONS processes below.

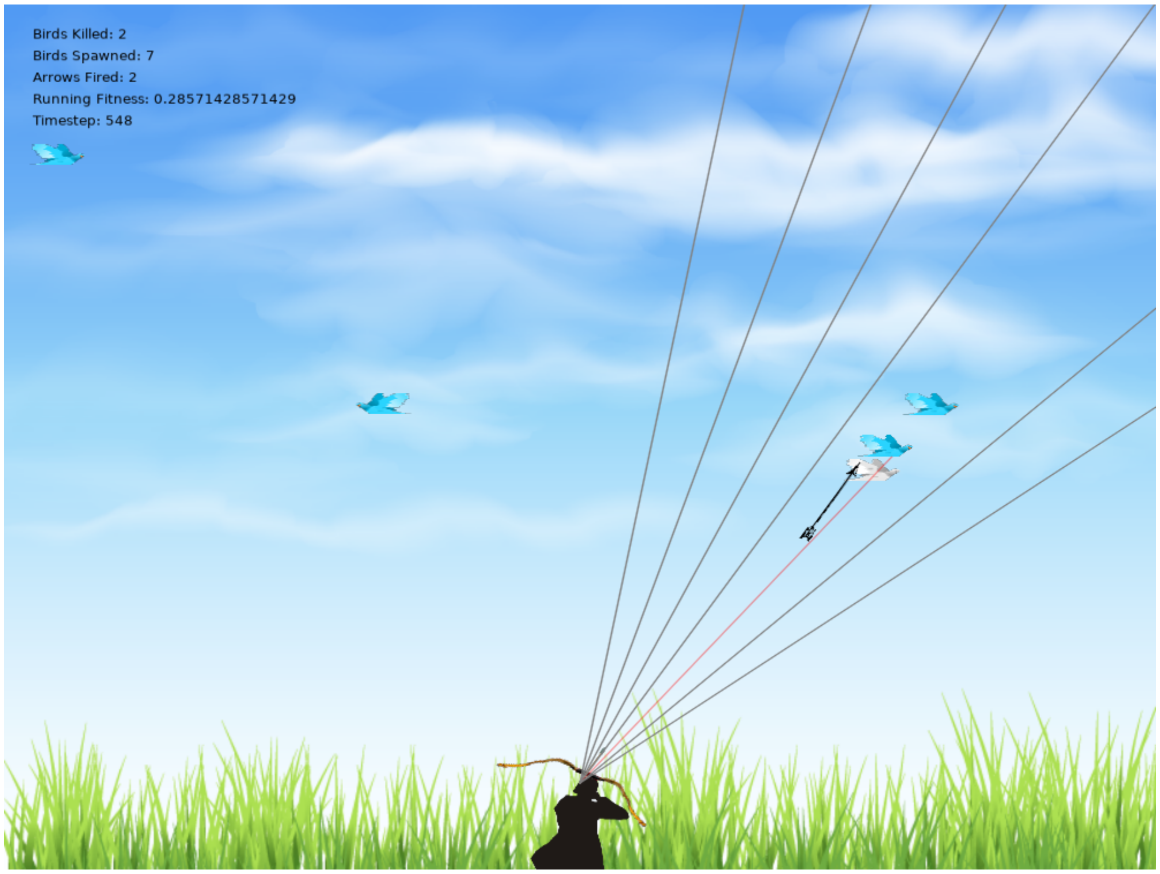
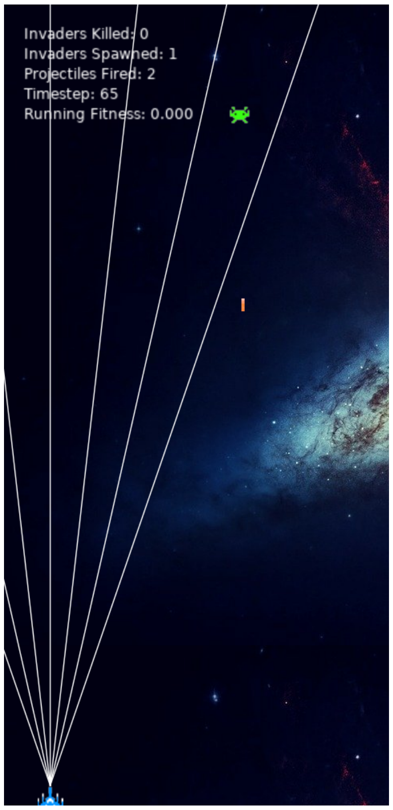


Figure 3.6: A screenshot of bowman application. In this example, the player is shooting three birds with one arrow.



Invaders Killed: 0
Invaders Spawned: 1
Projectiles Fired: 2
Timestep: 65
Running Fitness: 0.000

Figure 3.7: A screenshot of spaceinvaders application. In this example, the player shot its laser, and then moved to the left.

1. EONS starts with generating a random population of networks to be evaluated by an application on a specific neuroprocessor.

2. EONS receives the fitness value of each network from the driver program. EONS selects networks to perform one or more of the following operations, to produce a new population to be evaluated. EONS provides a set of common selection algorithms including but not limited to tournament and Roulette Wheel.

    (a) Mutation: EONS selects one network and does one of these - add a neuron, add a synapse, delete a neuron, delete a synapse, change the property value of the neuron such as threshold, and change the property value of synapse, such as delay or weight.

    (b) Duplication: EONS selects one network and makes a copy of it.

    (c) Crossover: EONS selects two parent networks and generates two child networks. The two parent networks distribute neurons and synapses to the two child networks. Each child network inherits exactly one neuron/synapse from either parent network. Figure 3.8 shows the crossover operation of EONS.

    (d) Merge: EONS selects two parent networks and generates one network that is the union of these two parent networks. In other words, the child network inherits all neurons and synapses from both parent networks.

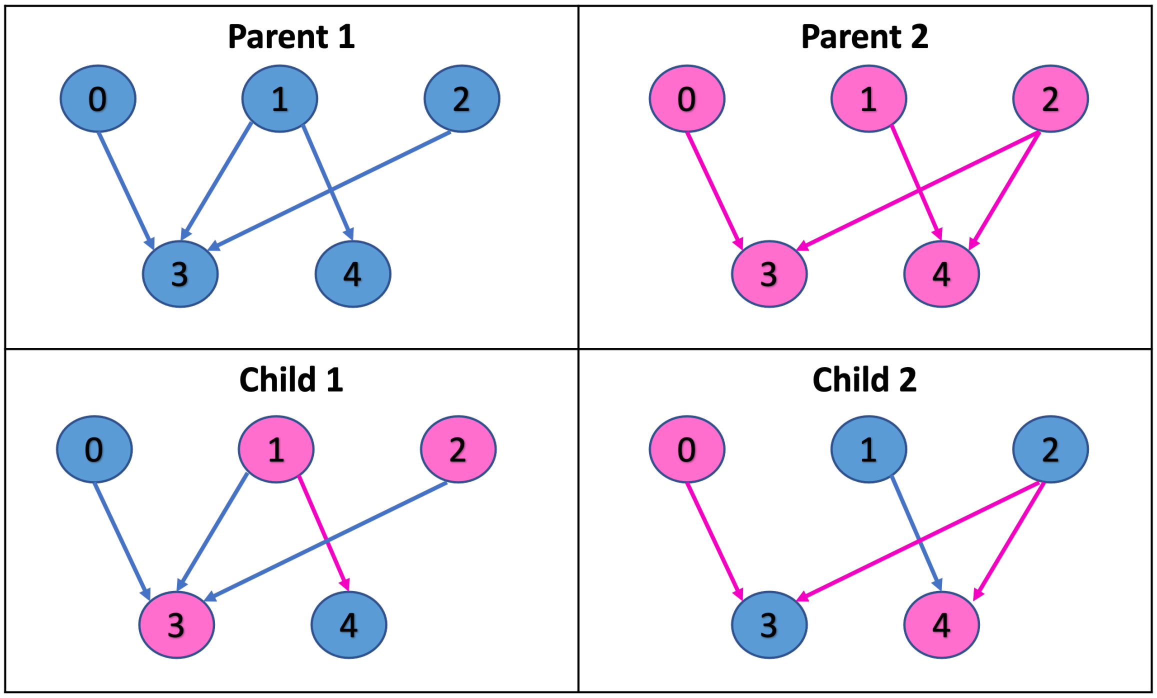


Figure 3.8: Crossover operations of EONS. The same color of neurons and synapses have the same properties

Chapter 4

Software Structure

Evolutionary Optimization for Neuromorphic Systems (EONS) is a genetic algorithm that has trained spiking neural networks (SNN) on a variety of applications including classification and real-time control [15, 22]. Although EONS can achieve thread-based parallelism in respect to network evaluation, it takes a significant amount of time for networks to converge for more difficult problems. However, when coupled with the Island(s) workflow, we can achieve both thread-based parallelism where each EONS run can evaluate the fitness of multiple SNNs concurrently and processor-based parallelism where Island can manage multiple EONS training runs concurrently. At a high level, the Islands framework can manage multiple concurrent EONS jobs split across many processors. It offers different migration policies and population combination strategies to aid in convergence. The structure of Islands framework can be broken into three parts - manager, worker, and monitor. We show them in Figure 4.1 and describe each part of the island framework in detail in the following subsections. The Islands framework is implemented in C++, and we use sockets for communication between the Island manager and Island workers/Island monitor. We use pipes between Island workers and EONS runs.

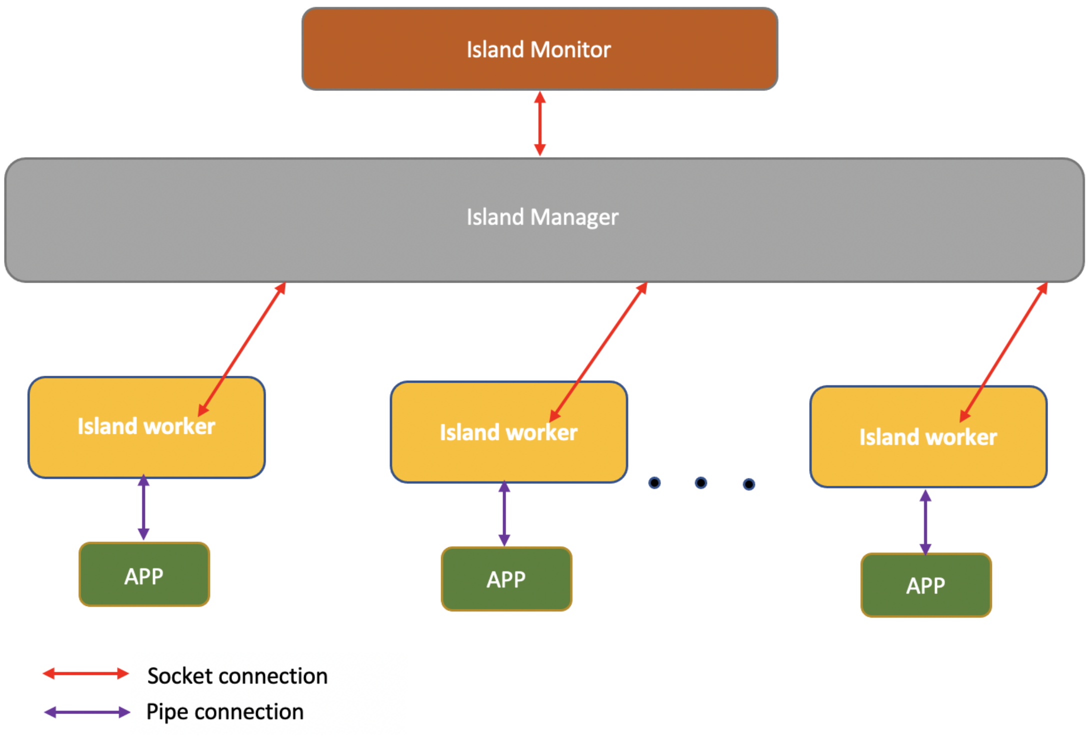


Figure 4.1: Island Structure

4.1 Island Manager

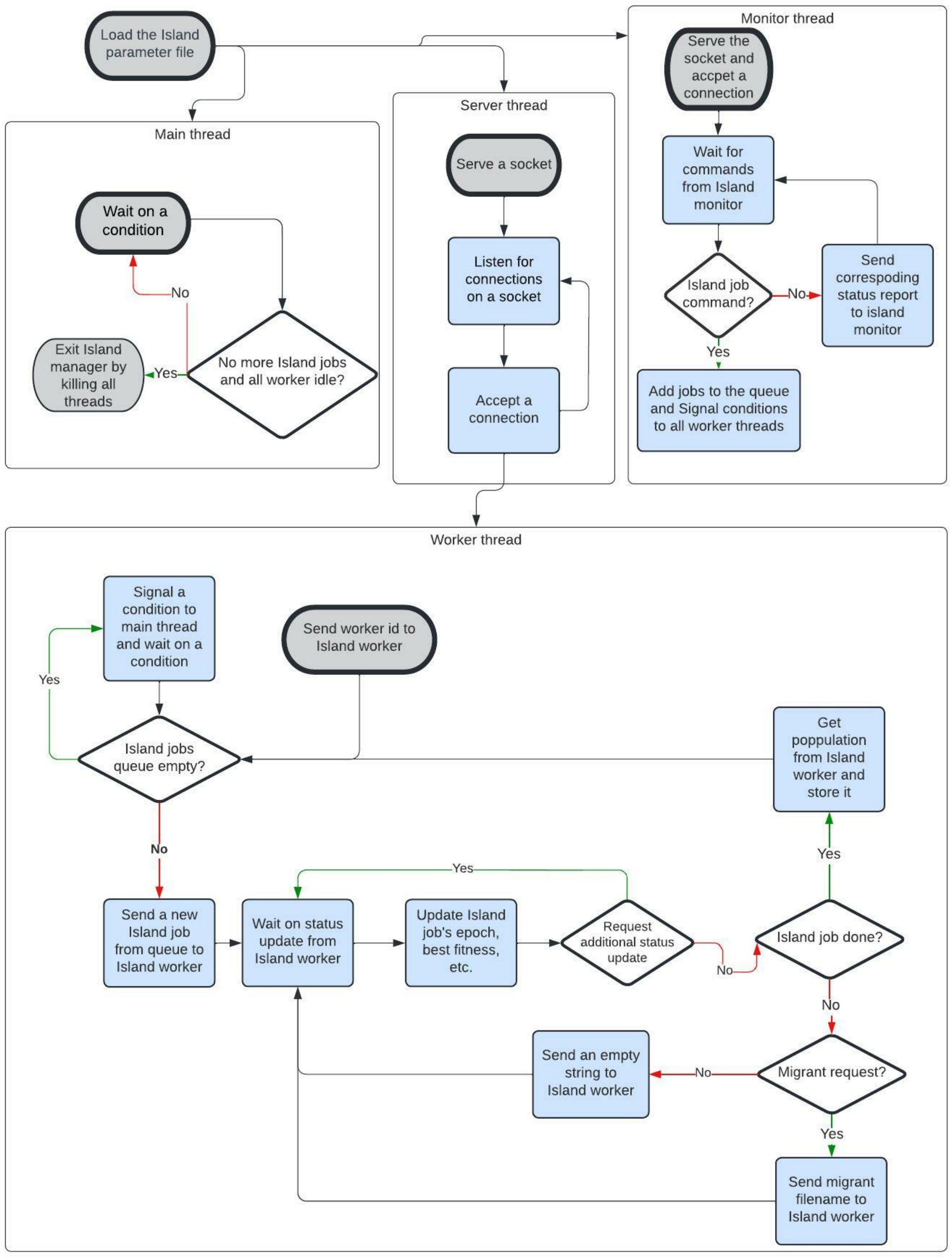
When the Island manager starts, it reads in a JSON parameter file to set up the initial state (The parameters are described in 4.1.1). Then, it spawns one "server" thread and one "monitor" thread. The "server" thread accepts connections from different Island workers and spawns one "worker" thread per connection. We display each thread, including the main thread, below and in Figure 4.2 as well.

Server thread: This creates a socket and waits to accept connections from Island workers. When a connection is established, it spawns one "worker" thread. If there are n Island workers, it will spawn n "worker" threads.

Exit thread (main thread): The "exit" thread is responsible for terminating all worker threads, the monitor thread, and then exiting the Island manager gracefully.

Monitor thread: This creates a socket whose port is different than what the "server" thread uses and accepts a connection from the Island monitor. It processes the monitor's commands accordingly. We describe the Island monitor's commands in detail in the "Island Monitor" section 4.2.

Worker thread: This sends one EONS job at a time from a queue to the corresponding Island worker. At every epoch of an EONS run, it will receive a status update from the Island worker, and then decide whether it will send migrants or not (the migration operation is defined by commands provided to the Island monitor or by an internal migration policy. They are described in section 5.4). Upon the EONS job's completion, it will attempt to send another EONS job to the Island worker.



Load the Island parameter file
Monitor thread
Serve the socket and accpet a connection
Server thread
Main thread
Wait for commands from Island monitor
Serve a socket
Wait on a condition
Send correspoding status report to island monitor
Listen for connections on a socket
No
Island job command?
No
Exit Island manager by killing all threads
Yes
No more Island jobs and all worker idle?
Yes
Add jobs to the queue and Signal conditions to all worker threads
Accept a connection
Worker thread
Signal a condition to main thread and wait on a condition
Send worker id to Island worker
Yes
Get poppulation from Island worker and store it
Island jobs queue empty?
No
Yes
Yes
Send a new Island job from queue to Island worker
Wait on status update from Island worker
Update Island job's epoch, best fitness, etc.
Request additional status update
Island job done?
No
No
Send an empty string to Island worker
No
Migrant request?
Yes
Send migrant filename to Island worker

Figure 4.2: Island manager workflow

4.1.1 Island Manager Parameters

The Island manager reads in a set of parameters to figure out where to store checkpoint files, how to run the application agent, what the parameters of the application agent are, what migration policy is being used, etc. We provide an example JSON file for the **Spaceinvaders** application and give a detailed explanation of each parameter below.

```
{
  "agent_directory": "./cpp-apps",
  "agent_executable": "bin/spaceinvaders_risp -a train",
  "checkpoint_directory": "/yourpath/spaceinvaders_risp",
  "seed_island": 0,
  "monitor_port": 22222,
  "communication_period": 10,
  "max_migrant_pool_size": 256,
  "migration_policy": "none",
  "stagnant": { "epochs": 4, "networks": 4 },
  "agent_params": {
    "encoder": { "spikes":
       { "flip_flop": 2, "max_spikes": 8, "min": 1, "max": 1 } },
    "extra_eons_params": {
        "population_size": 50,
        "num_best": 4,
        "starting_nodes": 2,
        "starting_edges": 46,
        "random_factor": 0
    },
    "epochs": 5000000,
    "threads": 48,
    "episodes": 10,
    "no_show_epochs": false,
    "prune_before_loading": true,
    "show_populations": true,
    "include_networks": true,
    "migrant_population": "-"
  }
}
```

- **agent_directory:** The directory in which we run **agent_executable**.

- **agent_executable:** The basic command to execute the application agent.
- **checkpoint_directory:** The directory in which we store checkpoint files.
- **seed_island:** A random number generator seed. It decides the random number generator seed for each island's (EONS's) run.
- **monitor_port:** A socket port number to which the Island monitor can connect.
- **communication_period:** How often (in terms of epochs) the Island manager and the Island worker send/receive migrants.
- **max_migrant_pool_size:** The size of the migrant pool in Island manager.
- **migration_policy:** A value of "*none*" disables it. See section migration policy evaluation for other migrant policies.
- **stagnant:** We define this as a JSON **"{"epochs": e, "networks": n }"**, which says the island job stops if the best n networks do not improve over e epochs. If n or e is less than or equal to 0, the island jobs run to completion.
- **agent_params:** The parameters for application agent.

4.2 Island Monitor

The Island monitor is essentially a scheduler that allows a user to control how Island jobs are organized and run. The Island monitor features RANDOM, COMB, TRANDOM operations that can be used to add Island jobs to the Island job queue. We describe these three operations below. We also describe more operations in section 5.5, Island Combining Topology.

- **RANDOM n [initial_population_size] [initial_min_fitness] [stag_epochs stag_networks]:** This creates n islands whose initial population is random and of size population_size, with minimum fitness threshold initial_min_fitness. A population size of -1 uses the population size defined in the EONS parameter file. By default, population_size = -1, initial_min_fitness = -1, and stagnation parameter uses default values in the island parameter file.

- **TRANDOM n time_limit [initial_population_size] [initial_min_fitness]:** The usage of this is similar to RANDOM command, but the island job stops when it finishes all EONS's epochs or the running time exceeds **time_limit**.

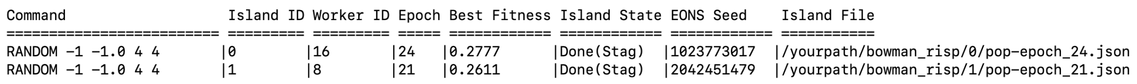
- **COMB [island_id number_of_best_networks epoch] * [stag_epochs stag_networks]:** This creates one island job whose initial population is created by combining some of the best networks from other islands. For example, "COMB 0 25 10 1 25 10" says to combine islands 0 and 1 and use the best 25 networks at epoch 10 from each island to create a new island. The stagnation parameter follows the same rule as RANDOM command.

The Island monitor also provides commands to retrieve the most up-to-date information from the Island manager. We describe these commands below:

- **ISLANDS [state(ALL | RUNNING | STAG | DONE | DISCARD | PAUSE) = ALL] [threshold = -1] [from = 0] [to = MAX_INT]:** By default, the state is ALL. It tells you the one-line status of islands that have state "stat" from "from" to "to" (not inclusive) whose best fitness is greater than or equal to threshold. Figure 4.3 shows an output example.

- **ISLANDTOP num:** This tells you a one-line status of best *num* islands. The output format is the same as **ISLANDS** command.

- **STATUS:** This shows the status, which includes but is not limited to, the total number of islands, the number of stagnant islands, the number of non-initialized islands, the number of initialized but non-stagnant islands, the current best fitness, the number of active workers, etc.



```
Command                   Island ID Worker ID Epoch Best Fitness Island State EONS Seed    Island File
========================= ========= ========= ===== ============ ============ ============ ===========
RANDOM -1 -1.0 4 4       |0        |16       |24   |0.2777      |Done(Stag)  |1023773017  |/yourpath/bowman_risp/0/pop-epoch_24.json
RANDOM -1 -1.0 4 4       |1        |8        |21   |0.2611      |Done(Stag)  |2042451479  |/yourpath/bowman_risp/1/pop-epoch_21.json
```

Figure 4.3: An output example of running command “ISLANDS ALL -1 0 2”

- **IINFO n:** This gives more detailed information about island n, which can be used to reproduce island n's run. Figure 4.4 shows an output example.
- **WINFO:** This shows information about workers.
- **QINFO:** This shows information about island jobs to be done.
- **BF network_file:** This stores the best network to $network\_file$

The Island monitor also provides CIRMIG to perform circular migration among islands.

- **CIRMIG from_island to_island window_size stride num_best_networks epoch:** This does a circular migration between island $from\_island$ and island ($to\_island - 1$) at epoch *epoch*. It works by sliding a fixed size window with a stride of *stride* from island $from\_island$ to island *to_island*. Each island within the window sends $num\_best\_networks$ networks to the island one past the last island on the window. Figure 4.5 shows the command "CIRMIG 0 10 4 2 xxx xxx" where "xxx" is any positive integer values.

4.3 Island Worker

We show the workflow of an Island worker in Figure 4.6. An Island worker receives a command from the Island manager and forks off a new process. This new process allows for multiple threads to evaluate the fitness of networks. The Island worker and the new process communicate via a pipe. At every epoch of an EONS training run, the Island worker will only send epoch and fitness status updates to reduce communication traffic. Additional status update information such as population and best network. can be requested if necessary.

```
Island Id:       0
Run time:        105.00 secs (00:01:44)
Worker Id:       16
EONS seed:       1023773017
Epoch:           24
Island state:    Done(Stag)
Best fitness:    0.27772717909967937
Monitor Command: RANDOM -1 -1.0 4 4 (init_population_size init_min_fitness stag_epochs stag_networks)
Initial Pop:
How to migrate:
App Command:     bin/bowman_risp -a train --encoder '{"spikes":{"flip_flop":2,"max":1,"max_spikes":8,"min":1}}' --episodes 10 --epochs
500000 --extra_eons_params '{"num_best":4,"population_size":50,"random_factor":0,"seed_eo":1023773017,"starting_edges":46,"starting_nod
es":2}' --include_networks --migrant_population - --prune_before_loading --show_populations --threads 48
Island files:    /yourpath/bowman_risp/0/pop-epoch_24.json
Epoch times:     [5.10, 11.10, 16.00, 21.30, 25.60, 29.60, 34.00, 37.70, 39.80, 43.40, 47.50, 51.20, 55.40, 60.30, 64.90, 68.60, 73.40,
 77.40, 82.20, 85.90, 89.70, 93.50, 97.10, 100.90, 105.00]
Epoch fitness:   [0.23, 0.24, 0.24, 0.25, 0.25, 0.25, 0.25, 0.25, 0.26, 0.26, 0.26, 0.27, 0.27, 0.27, 0.27, 0.27, 0.27, 0.28, 0.28, 0.2
8, 0.28, 0.28, 0.28, 0.28, 0.28]
# Stag nets:     [0,1,1,2,2,0,2,2,2,3,2,1,1,1,4,4,4,3,4,3,2,4,4,4,4]
```

Figure 4.4: An output example of running the command "IINFO 0". "Epoch times" shows the accumulated running time at each epoch. "Epoch fitness" shows the best fitness at each epoch. "Stag nets" shows the number of stagnated networks from the best n networks at each epoch where n is defined by JSON "{"stagnant": {"networks": n}}" in Island parameters file.

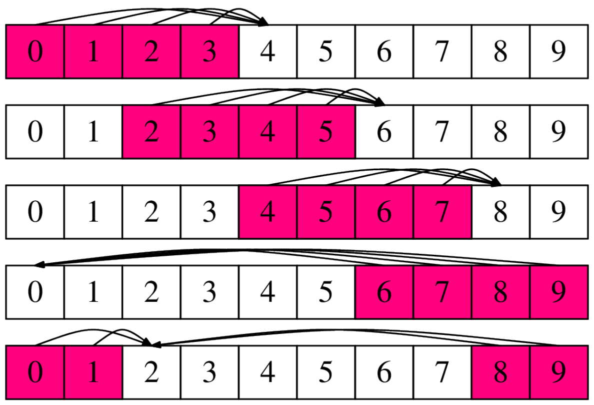


Figure 4.5: Circular migration policy with a stride of 2 and window size of 4. The window is colored in pink. The edge represents the flow of migration.

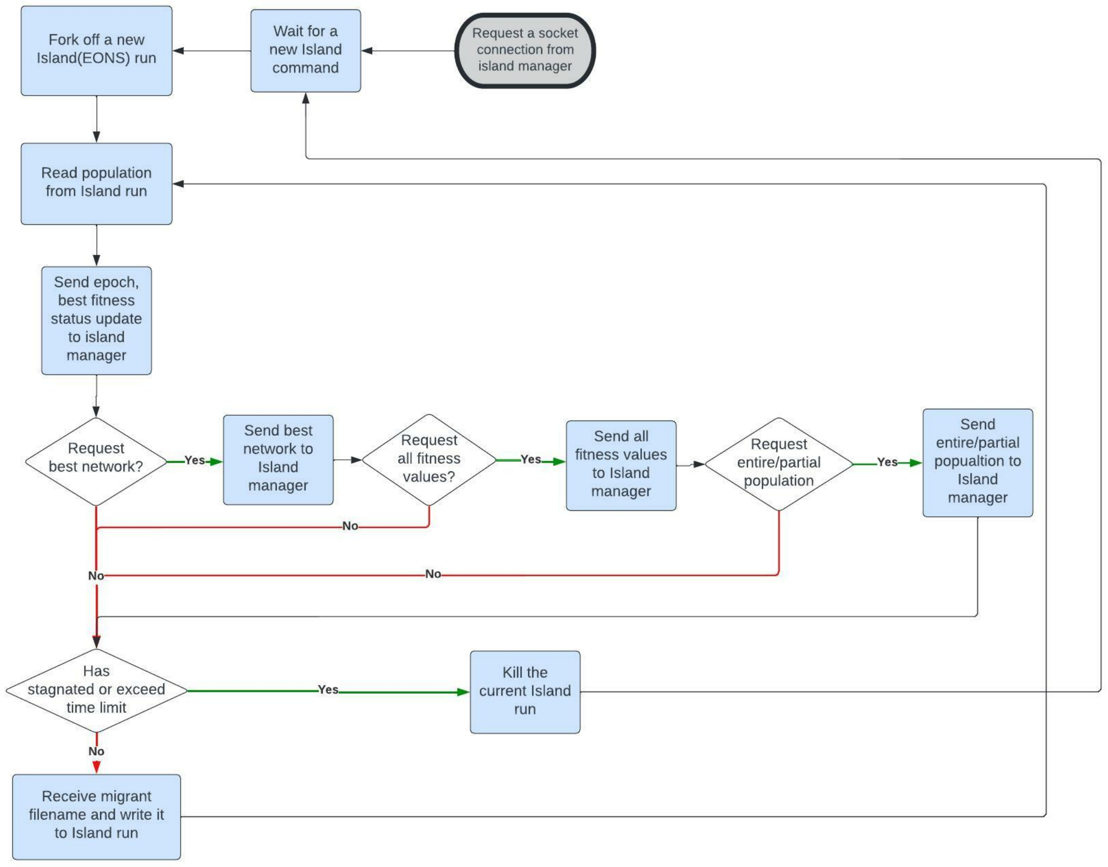

Request a socket connection from island manager
Wait for a new Island command
Fork off a new Island(EONS) run
Read population from Island run
Send epoch, best fitness status update to island manager
Request best network?
Yes
Send best network to Island manager
Request all fitness values?
Yes
Send all fitness values to Island manager
Request entire/partial population
Yes
Send entire/partial popualtion to Island manager
No
No
No
Has stagnated or exceed time limit
Yes
Kill the current Island run
No
Receive migrant filename and write it to Island run

Figure 4.6: Island worker workflow

4.4 Usage

Since the Island manager acts as a server for the Island workers and Island monitor, we have to run the Island manager first. Then we run the Island workers and Island monitor in any order. We show an example of how to run 100 random island jobs using *SLURM* below. We use the argument "activity" to control which component of Island gets run.

```
UNIX> srun --nodelist=chi0 --exclusive -n1 -c45 \
      ./bin/island --activity manager \
      --island_input island_params.json \
      --island_output island_out.json
UNIX> srun -n32 -c48 ./bin/island --activity worker --host chi0 &
UNIX> echo "RANDOM 100" | bin/island --activity monitor \
      --checkpoint_file checkpoint_file_in_island_params \
      --host chi0 --port monitor_port_number_in_island_params
```

Chapter 5

Case Study

In this section, we explore the features of Islands with a case study of optimizing the polebalance, spaceinvaders and bowman applications, described in section 3.3, on the RISP neuroprocessor (section 3.2. The goal of this case study is to demonstrate the variety of functionalities of Islands and the fact that they can improve the performance of optimization. Because the variety of parameters is so high, and fact that general conclusions on optimizations are so hard to draw, this case study does not intend to derive optimal Islands configurations or strategies. Its goal is to demonstrate the possibilities.

We run all of our experiments on the TENNLab Neurocluster computer, composed of Dell PowerEdge C6145s, with following specifications. For all of our experiments, we use 32 nodes and 48 cores per node, which has a total of 1536 cores. We either use 32 or 256 island workers, which consumes 48 and 6 cores per island worker respectively. Additionally, we train all applications with 10 episodes (random starting seeds).

- 36 nodes
- 48 cores per node
- Quad 12 core 64 bit AMD Opteron(tm) Processor 6180 SE
- 96 GB RAM per node

- 1 TB local storage per node

5.1 Encoding/Decoding Choice

We selected the "spikes-flip-flop encoding" for bowman and spaceinvaders, and temporal for polebalance. We selected "winner-take-all" decoding for all applications. These encoding/decoding techniques are described in 3.1. It is worth mentioning because polebalance is such a simple application, it always trains to a perfect fitness with spikes-flip-flop encoding. For that reason, we chose a more challenging encoding technique.

5.2 Determining stagnation parameters

The goal of this first experiment is to define what "stagnation" means. Intuitively, it means that an EONS population is unlikely to improve, and therefore we are better served to migrate networks, combine this population with another population, or simply stop trying to optimize this population, than we are to keep trying to improve the population. Our goal is to determine two parameters that define stagnation:

1. *Stagnant_networks*: If a population has stagnated, then the fitness of the best *stagnant_networks* has not changed.

2. *Stagnant_epochs:* If the best *stagnation_networks* have not changed for *stagnation_epochs*, then the system is stagnated.

There are pragmatic tradeoffs with setting these parameters low vs. high. When they are high, then the likelihood of a population improving is low. High values give one more reliable "stagnation." However, high values mean that EONS spends a lot of computational time confirming that the population has stagnated. On the flip side, when the parameters have low values, they consume far less CPU time, but there is a higher probability that the population may improve were EONS to continue running.

To evaluate this tradeoff, we set *stagnant_networks* to 5 and *stagnation_epochs* to 10, figuring these values are higher than what we will eventually decide as stagnation. We then ran 100 EONS runs, dedicating 48 cores to each run, and having each run terminate when its best 5 networks had not changed for 10 consecutive epochs. Because of the driver's bookkeeping, we may use these runs to determine what happens when *stagnant_networks* is any value less than or equal to 5, and *stagnant_epochs* is any value less than or equal to 10.

We use the following parameters for EONS, of which "starting_edges" is dependent on the number of inputs and outputs. We selected these parameters from our experience with previous research on EONS.

```
{
    "starting_nodes": 2,
    "starting_edges": 2 * (#input + #output),
    "merge_rate": 0,
    "population_size": 50,
    "multi_edges": 0,
    "crossover_rate": 0.9,
    "mutation_rate": 0.9,
    "selection_type": "tournament",
    "tournament_size_factor": 0.1,
    "tournament_best_net_factor": 0.9,
    "random_factor": 0,
    "num_mutations": 3,
    "node_mutations": { "Threshold": 1.0 },
    "net_mutations": { },
    "edge_mutations": { "Weight": 0.65, "Delay": 0.35 },
    "num_best": 5
}
```

The results are in Figure 5.1. For each application, we show the number of epochs until stagnation, the running time until stagnation, the fitness values at stagnation and the heatmap of fitness change along number of stagnant epochs over running time using the median.

As with many optimization experiments (e.g., [23]), it is hard to draw conclusions that span all three applications. It is clear that as we increase *stagnant_networks*

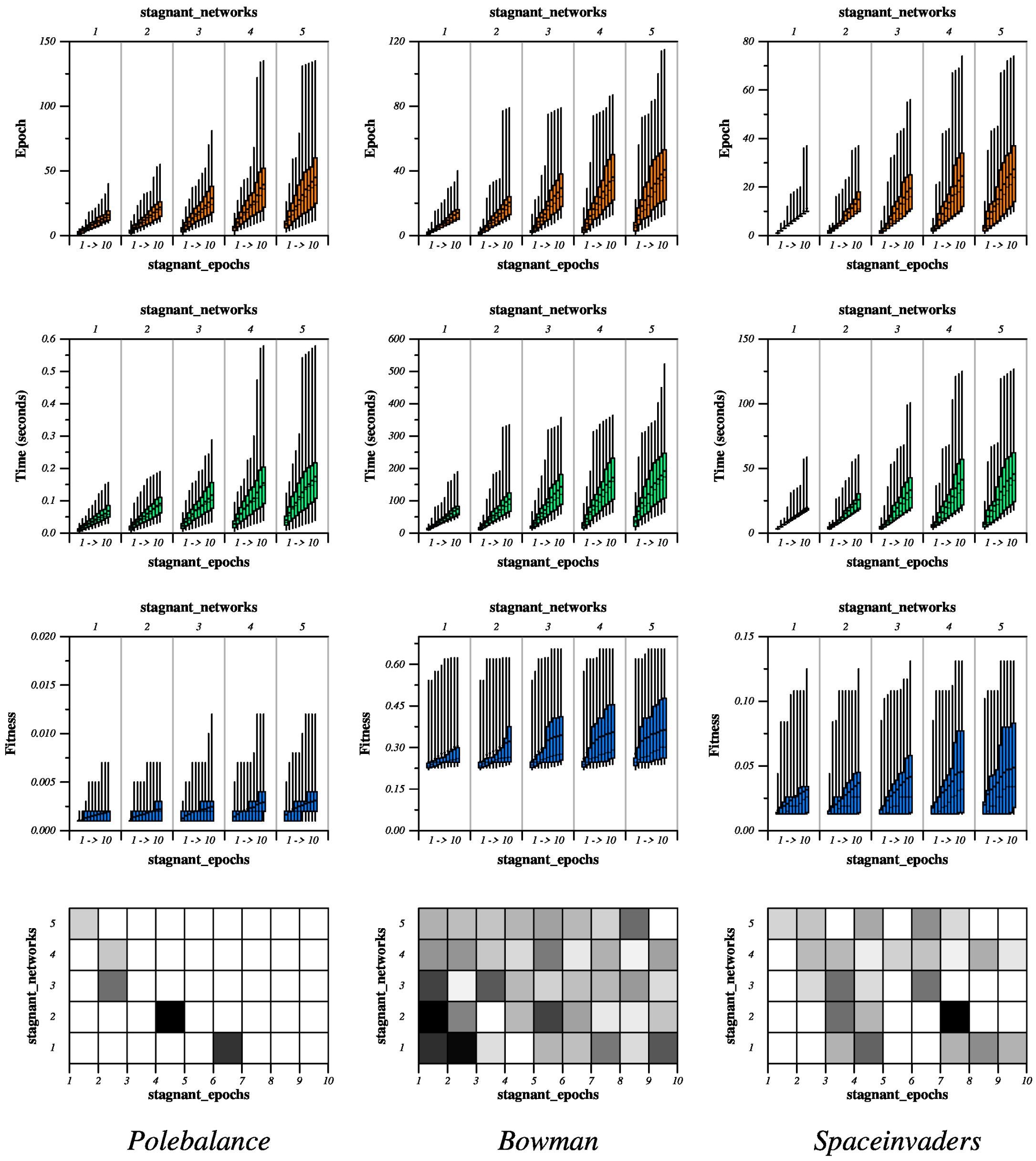


Figure 5.1: The Tukey plot and heatmap for 100 EONS runs of bowman, spaceinvaders, and polebalance. We show the number of epochs until stagnation (first row), the running time until stagnation (second row), and the fitness values at stagnation (third row). We also show the heatmap of fitness change along the x-axis (number of stagnant epochs) over the running time (the last row). The darker the color, the bigger the increase in fitness as the number of stagnant epochs increases.

and *stagnant_epochs*, all three values (epochs, running time, fitness) go up. It is also clear that there are diminishing returns as these values increase. For the remainder of the case study, we chose values of 4 epochs and 4 networks to define stagnation. Our decision traded off the data in Figure 5.1 with pragmatics of keeping running times from being too long. It is a matter of further research to determine the proper stagnation parameters of a given application and optimization.

5.3 Control Experiment

To evaluate the performance of migration policy and the population combining strategies, we need a baseline fitness to compare. Therefore, we define two experiments below without any migration policies or population combining strategies as our baseline.

- **Stagnate-and-quit:** We ran 10,000 independent EONS jobs where each one runs to stagnation and then stops.
- **Do-not-stagnate:** We ran 256 EONS runs and stopped them after a time t.

To allow for comparison, we made sure that the total CPU time of both tasks was equivalent. For example, if each of the 10,000 jobs in **Stagnate-and-quit** took an average of 25.6 seconds, then we stop each of the 256 jobs after 1,000 seconds. Therefore, both experiments consume 256,000 total seconds of CPU time.

We show the parameters of the six runs (three applications, two experiments) in Table 5.1. For the polebalance application, we increased the number of **stagnate-and-quit** jobs to 50,000, because they were so quick (poorly performing polebalance optimizations complete much more quickly than optimizations that perform well). We also show the Island monitor command for each data point.

Figure 5.2 shows tukey plots of the fitnesses of each of these runs. One may interpret these results as follows. If **Stagnate-and-quit** performs better, then we

Table 5.1: The running time, Island monitor command, and number of the Island workers for each control run.

Application	Monitor command	#Island workers	Running time(hr)
Polebalance-Stagnate-And-Quit	RANDOM 50000	32	0.3
Polebalance-Do-Not-Quit	TRANDOM 256 0.3	256	0.3
Bowman-Stagnate-And-Quit	RANDOM 10000	32	8.5
Bowman-Do-Not-Quit	TRANDOM 256 8.5	256	8.5
Spaceinvaders-Stagnate-And-Quit	RANDOM 10000	32	1.93
Spaceinvaders-Do-Not-Quit	TRANDOM 256 1.93	256	1.93

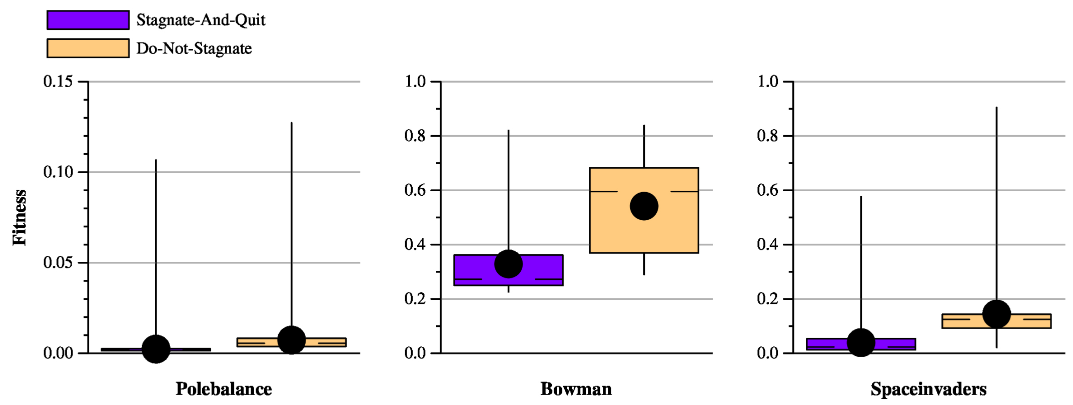


Figure 5.2: Fitness of the control runs.

see true stagnation – we are better off trusting the randomness of many runs instead of running EONS longer. If **Do-Not-Quit** performs better, then our definition of stagnation is likely too aggressive, quitting the simulation too soon, when EONS could provide substantial optimization if allowed to run longer. Figure 5.2 shows the latter – our definition of stagnation_epochs and stagnation_networks was too low in all three applications.

Although the results that follow use this overly aggressive definition of stagnation, they are useful for the following reason – if, by using an aggressive definition of stagnation, we can migrate and combine populations more effectively than simply running EONS longer, then we have identified situations where these actions are useful. We will see this below.

5.4 Migration Policy Evaluation

In this section, we introduce six different migration policies below, which can be defined by a JSON key ‘‘migration_policy” in the island parameter file or via the Island monitor. We evaluate each migration policy by running 256 island jobs for approximately the same amount of time as the control experiment described in section 5.3 via the Island monitor command **“TRANDOM 256 time”**.

- **Greedy:** We define **migration_policy** as a string “greedy”. The greedy migration policy keeps a pool of best networks from all island runs. Upon each migration request, the Islands worker will send all networks that are better than the best of the migrant pool to the Island manager or receive all networks from the migrant pool that are better than the best network of the current population. The period of request and the pool size are defined as “communication_period” and “max_migrant_pool_size” respectively in the island parameter file.

  Generally, we want to keep communication period high and migrant pool size low. This gives each Islands worker enough time to evolve the best networks in the migrant pool while also maintaining genetic diversity among the Islands workers. In particular, we set “communication_period” to 100 and “max_migrant_pool_size” to 10 for our experiments.

- **Only-Stagnate:** We define **“migration_policy”** as a JSON object **{“only _stagnate”: num_migrants}**. When an island stagnates, instead of killing it, we send random *num_migants* networks from the Island manager’s migrant pool to it. How the Island manager stores the networks to the migrant pool from the Island workers works the same way as the ”greedy” migration policy.

  Generally, we want to keep communication period low and migrant pool size high. Each island job tends to stagnate quicker when fitness gets higher. With a higher migrant pool size, it makes sure each request gets relatively different networks from the migrant pool. With a lower communication period, the migrant pool receives the best networks from the Island workers more often, which keeps the migrant pool up-to-date. In particular, we set “communication_period” to 10 , “max_migrant_pool_size” to 256, and “migration_policy” to “{“only_stagnate”: 1 }” for our experiments.

- **Only-All-Stagnate:** We define **migration_policy** as a JSON object **{“only _all_stagnate”: num_migrants}**. This is like the “Only-Stagnate” policy, but the Island manager starts to send random *num_migrants* networks from the migrant pool to the Islands worker when all island jobs stagnate.

  Generally, this policy doesn’t utilize full computational resources all the time because many islands workers will pause to wait for the other islands jobs to stagnate. In particular, we set “communication_period” to 10 , “max_migrant_pool_size” to 256, and “migration_policy” to “{“only_all_stagnate”: 1 }” for our experiments.

- **Circular:** Unlike other migration policies, we will use the command “CIRMIG” (see section “Island Monitor” 4.2) to perform circular migration. With this migration policy we disable the stagnation parameter. We feed the Islands manager with commands "CIRMIG 0 256 1 1 1 epoch" where epoch = 100, 200, 300, ..., 100n (n is large enough) for our experiment.

- **Adjacent-Only-Stagnate:** We define **migration_policy** as a JSON object **{"adjacent_only_stagnate": num_migrants}**. The island n receives *num_migrants* best networks from neighbors, which are islands $(n-1)\%total\_islands$ and $(n+1)\%total\_islands$ where *total_islands* is the number of islands are currently running. In particular, we set "migration_policy" to "{"adjacent_only_stagnate": 1}" for our experiment.

- **Circular-Only-Stagnate:** We define **migration_policy** as a JSON object **{"circular_only_stagnate": num_migrants}**. Unlike **Circular** that is required to define epoch, stride, window, etc. **Circular-Only-Stagnate** does circular migration with a stride of 1 and window size of 1 where each island n receive *num_migrations* migrants from $(n-1)\%total\_islands$ only when all the islands stagnate.

  Like the **Only-All-Stagnate** policy, it also doesn't utilize full computational resources. In particular, we set "migration_policy" as "{"circular_only_stagnate": 1 }" for our experiment.

These six policies may be viewed as two sets of three policies: the first set uses the global migrant pool, and the second set uses neighbor-based migration. Within each set of three, the first policies (**Greedy** and **Circular**) is aggressive and unilateral: islands are injected with migrants at uniform intervals, regardless of whether their populations have stagnated. The second set (**Only-Stagnate** and **Adjacent-Only-Stagnate)** are demand driven: an island only receives migrants if it has stagnated, and if different islands stagnate at different rates, it is handled by having fewer migrants sent to the less-frequently stagnating islands. The last of these policies (**Only-All-Stagnate** and **Circular-Only-Stagnate**) go back to forcing the islands to work on a coordinated schedule, this time stalling islands that stagnate to wait until all of the others have stagnated.

As mentioned above, the **Only-All-Stagnate** and **Circular-Only-Stagnate** policies stall the migration until all islands stagnate, and therefore may not use all of the computational resources allocated for an Islands job. These policies are more appropriate in a resource-sharing environment, where computational resources may be employed by other jobs while the islands are "stalling," than one where resources must be exclusively allocated by a job. In our cluster environment, resources must be allocated exclusively, and we anticipate that these policies will not perform as well as others.

During migration operations, networks from the manager's pool may be sent to a lot of islands. As a result, in most cases, all islands eventually optimize to similar fitnesses. For that reason, we plot a bar graph instead of a Tukey plot in Figure 5.3.

These graphs in Figure 5.3 show some interesting results that warrant further study beyond this thesis. In polebalance and bowman, the two strategies that clearly outperform the others are **Only-Stagnate** and **Adjacent-Only-Stagnate**. These are the two strategies that perform migration on-demand: migrants are only sent to an island when it stagnates, and when an island stagnates, it doesn't have to wait for other islands to stagnate to receive migrants. We suspect that these strategies achieve a good blend of maintaining diversity while performing useful work and keeping their CPU's busy. We also suspect that if given the same overall CPU time, rather than limiting their wall-clock times, the **Only-All-Stagnate** and **Circular-Only-Stagnate** policies will achieve similar if not superior performance to **Only-Stagnate** and **Adjacent-Only-Stagnate**. As stated above, it warrants further study.

With spaceinvaders, all of the policies except **Only-All-Stagnate** and **Circular-Only-Stagnate** perform nearly identically. We conclude that with this application, the overall CPU time of the optimization is more important than how the individual optimizations are managed.

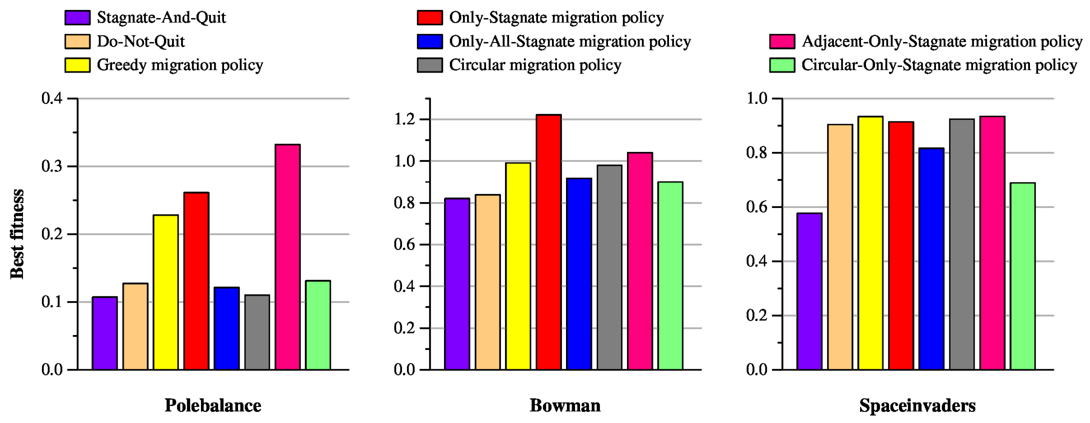


Figure 5.3: Performance differences between independent EONS/island runs (shown in the left two boxes) and islands with migration policy (shown in the right six boxes).

5.5 Island Combining Topology

Island combination is an operation where we create a new island whose initial population comes from some number of best networks from other islands. In this section, we will discuss two combining topologies **TREECOMB** and **DENSETREECOMB**, which can be run via the Island monitor.

- **TREECOMB islands_per_comb level num_best_networks:** This creates a "*level*" level inverted full tree where level l has $islands\_per\_comb^{level-l}$ islands. We start by creating $island\_per\_comb^{level}$ random islands at level 0. Then, for each level we move down, each island at level l is made by combining *islands_per_comb* islands from level $l-1$. This reduces the number of islands by a factor of *island_per_comb* each time. Figure 5.4 shows a graph presentation of command **"TREECOMB 3 3 xxx"** where "xxx" is any positive integer values.

- **DENSETREECOMB islands_per_comb level num_best_networks create_type("set" or "replication"):** This creates a "*level*" *level* tree where each level has exactly $islands\_per\_comb^{level}$ islands. The first level contains random islands. Then for each level we move down, each island at level l is made by combining *islands_per_comb* islands from level $l-1$. This is similar to **TREECOMB**, but in order to keep the same amount of islands at each level, we can do one of two things below:

  - **Replication:** At each level except for the first level, there are *islands_per_comb* identical sets of islands that are created by combining the same islands from level $l-1$. Figure 5.5 shows a graph representation of command **"DENSETREECOMB 3 3 xxx replication"** where "xxx" is any positive integer values.

  - **Set:** We start with one set of islands at level 0. For each set of of islands at level $l-1$, we create *islands_per_comb* sets of islands at level l by selecting

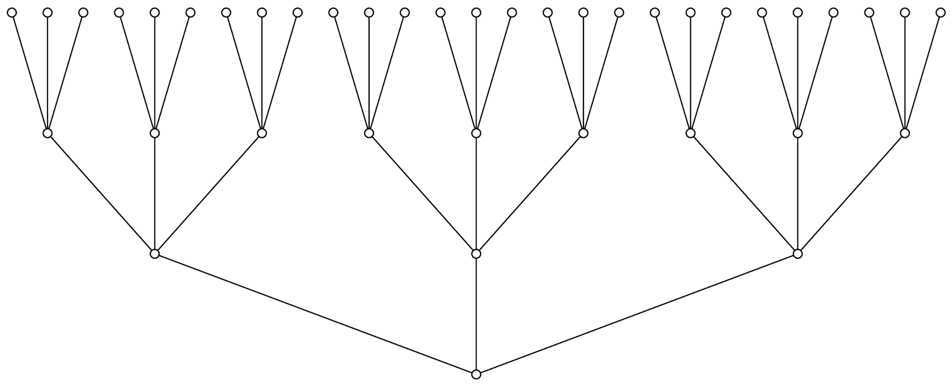


Figure 5.4: TREECOMB with 3 children and 3 levels

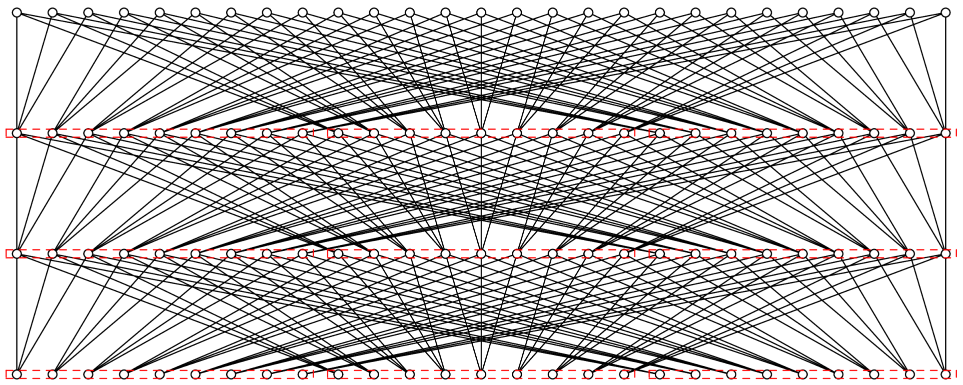


Figure 5.5: DENSETREECOMB with 3 children, 3 levels, and replication creation type. The identical sets are marked by the red bounding box.

islands from level $l-1$ with strides of 1, 2, ..., *islands_per_comb*. Level l has $islands\_per\_comb^l$ sets of islands. Figure 5.6 shows a graph representation of command **"DENSETREECOMB 3 3 xxx set"** where "xxx" is any positive integer values.

The results with different values of "islands_per_comb" are shown in Figure 5.7. For each set of "islands_per_comb" and topology, we chose a value for "level" such that the total number of EONS jobs is the greatest value, and meanwhile we can keep the total CPU time under or close to other experiments. For example, when we set "islands_per_comb" to 2, and performed **TREECOMB** for polebalance, we select a value of 14 for "level," because $\sum_{n=0}^{14} 2^n = 32,767$ and $\sum_{n=0}^{15} 2^n = 65,535$ (We did 50,000 **Do-Not-Quit** polebalance runs).

We may draw a few interesting conclusions from Figure 5.7. First, the DENSETREECOMB techniques outperform the controls in all cases. In most cases, using sets rather than replication was better. We surmise that this is because the sets create more diversity. The advantage is more marked when *islands_per_comb* is larger, underscoring the diversity argument.

The TREECOMB technique did not perform as well, which is likely because of the reduced overall CPU time employed.

5.5.1 Discussion

Studies like this always have deficiencies. Chief among them is the inability to explore a wide range of hyperparameters, and drawing significant conclusions from searches that rely on randomness. To highlight this latter problem, consider the two DENSETREECOMB plots on the lower-left graph in Figure 5.7 (Polebalance, *islands_per_comb* equals 4). The best network, achieved with replication, has a fitness of roughly 0.21. The best network achieved with sets is roughly 0.17. However, the Tukey plots show that the networks in the first through third quartiles using sets are

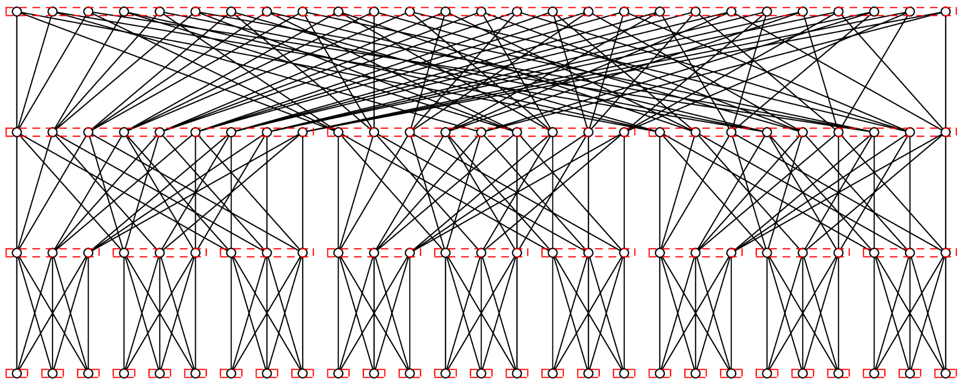


Figure 5.6: DENSETREECOMB with 3 children, 3 levels, and set creation type. Each set is marked by the red bounding box.

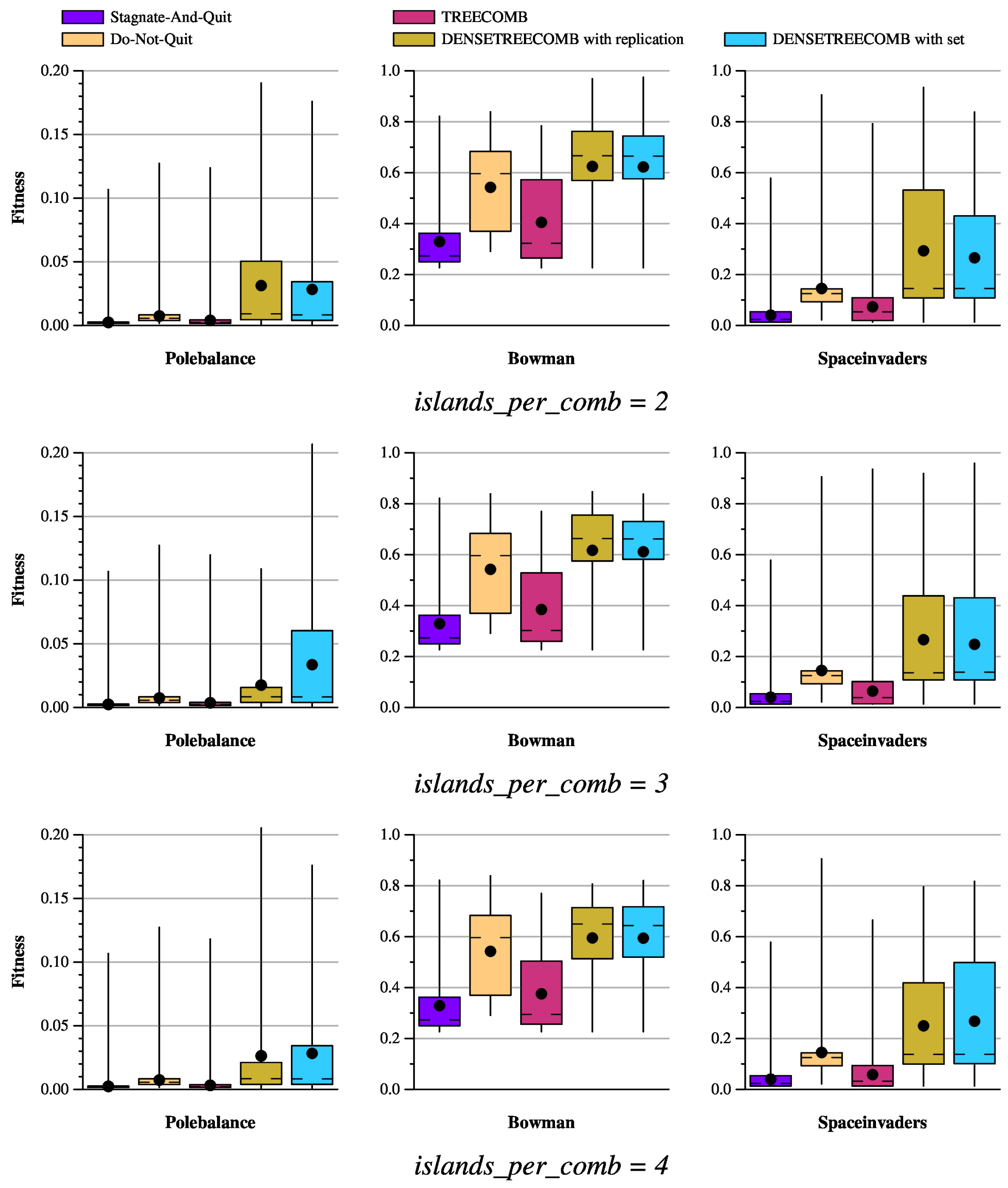


Figure 5.7: Results of combining islands with TREECOMB and DENSETREECOMB.

better than those in the first through third quartiles of replication. Which, then, do we conclude is a better technique?

Acknowledging the difficulty in drawing conclusions, we plot the best networks from each of our tests in Figure 5.8. In these graphs, the migration results generate the best networks by a significant margin in polebalance and bowman. With spaceinvaders, the tree-based combination methods outperform the migrations by a small amount. In each application, the Islands runs produced significantly better networks than the control strategies of independent EONS runs.

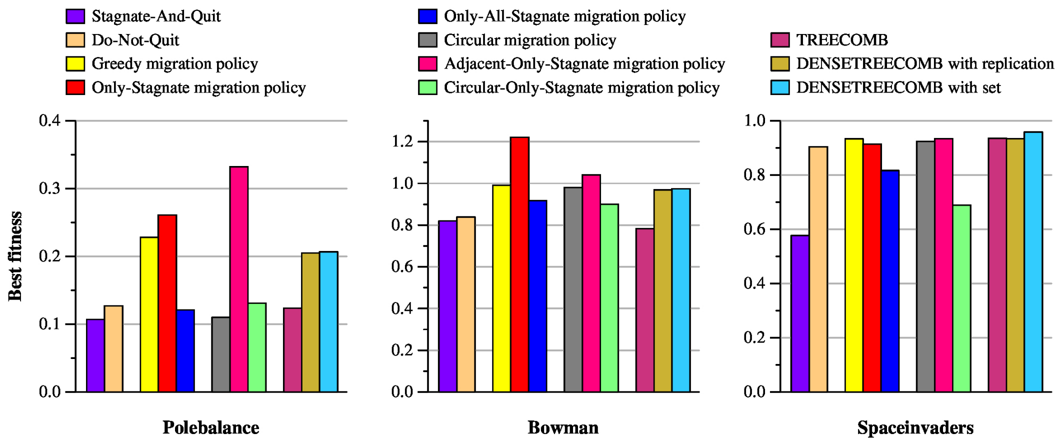


Figure 5.8: Best network in each of the Islands runs.

Chapter 6

Future Work

The work related to this project with the most immediate need is a thorough evaluation of the features of Islands. This is not an easy task. The case study in this Thesis is a good start; however, to truly evaluate Islands, we need more applications, more settings of hyperparameters, and a more rigorous analysis that blends fitnesses, CPU time, and randomness.

It is worth mentioning the variety of hyperparameters that affect each Islands run. First, there are settings from the TENNLab software framework:

- Choice of input encoder.
- Choice of output decoder.
- Constraints from the neuroprocessor (e.g. connectivity, discrete vs. continuous values, leak, plasticity).
- Selection of the number of timesteps for each individual **Run()** call.

Next, there are settings from EONS:

- Selection method (e.g. tournament) and parameters.
- Number of nodes for the initial networks.

- Number of synapses for the initial networks.
- Mutation rate, crossover rate, merge rate.
- Population size.

There are settings from Islands:

- How stagnation is defined.
- Migration strategy.
- Frequency of migration decisions.
- Number of neighbors for circular migration.
- Number of networks to include in migration.
- Number of levels in tree combinations.
- Number of children in tree combinations.
- Set vs. replication in tree combinations.
- Other combination strategies or merged combination/migration.

Finally, there are other operational settings:

- How to define the total computational resources allowed for a job.
- How many independent instances of a job.
- How to analyze fitness.

It will be a research challenge to derive good experiments to fully analyze the features of Islands.

Chapter 7

Conclusion

In this thesis, we have described the software structure of the Islands project and its command-line interface. The goal of the Islands project is to manage multiple EONS populations and provide an easy way to access the data such as fitness, running time, etc. We also introduce six migration policies and two population combing strategies with a case study of optimizing the polebalance, bowman, and spaceinvaders applications on RISP neuroprocessor. This case study demonstrates that the Islands can, in fact, improve the performance of optimization, which helps motivate continued research on the Islands project.

Bibliography

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 5

[2] Erick Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. 1997. 3

[3] Erick Cantú-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pages 91–98, 1999. 4

[4] Erick Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *Journal of heuristics*, 7(4):311–334, 2001. 4

[5] Erick Cantú-Paz et al. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998. 3, 4

[6] M. E. Dean, C. D. Schuman, and J. D. Birdwell. Dynamic adaptive neural network array. In *13th International Conference on Unconventional Computation and Natural Computation (UCNC)*, pages 129–141, London, ON, July 2014. Springer. 10

[7] Juan J Durillo, Antonio J Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize nsga-ii. In *2008 IEEE international symposium on parallel and distributed processing*, pages 1–8. IEEE, 2008. 3

[8] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1):47–62, 2008. 4

[9] Faustino Gomez and Risto Miikkulainen. 2-d pole balancing with recurrent evolutionary networks. In *International Conference on Artificial Neural Networks*, pages 425–430. Springer, 1998. 4

[10] Nikola Kasabov, Valery Feigin, Zeng-Guang Hou, Yixiong Chen, Linda Liang, Rita Krishnamurthi, Muhaini Othman, and Priya Parmar. Evolving spiking neural networks for personalised modelling, classification and prediction of spatio-temporal patterns with a case study on stroke. *Neurocomputing*, 134:269–279, 2014. 4

[11] Shyh-Chang Lin, William F Punch, and Erik D Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*, pages 28–37. IEEE, 1994. 4

[12] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*, pages 293–312. Elsevier, 2019. 4

[13] J. P. Mitchell, M. E. Dean, G. Bruer, J. S. Plank, and G. S. Rose. DANNA 2: Dynamic adaptive neural network arrays. In *International Conference on Neuromorphic Computing Systems*, Knoxville, TN, July 2018. ACM. 10

[14] P. Mitchell. CASPIAN: An integrated software and FPGA neuromorphic development platform. In *International Conference on Neuromorphic Computing Systems*, Knoxville, TN, July 2019. ACM. 10

[15] J. S. Plank, C. Rizzo, K. Shahat, G. Bruer, T. Dixon, M. Goin, G. Zhao, J. Anantharaj, C. D. Schuman, M. E. Dean, G. S. Rose, N. C. Cady, and

J. Van Nostrand. The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems. In *44th Annual GOMACTech Conference*, Albuquerque, March 2019. 5, 12, 14, 19

[16] J. S. Plank, C. Zheng, B. Gullett, N. Skuda, C. Rizzo, C. D. Schuman, and G. S. Rose. The case for RISP: A reduced instruction spiking processor. arXiv:2206.14016, 2022. 8, 10

[17] James Plank, Chaohui Zheng, Catherine Schuman, and Christopher Dean. Spiking neuromorphic networks for binary tasks. In *International Conference on Neuromorphic Systems 2021*, pages 1–9, 2021. 10

[18] James S Plank, Catherine D Schuman, Grant Bruer, Mark E Dean, and Garrett S Rose. The tennlab exploratory neuromorphic computing framework. *IEEE Letters of the Computer Society*, 1(2):17–20, 2018. 4, 5, 8

[19] Hossein Rajabalipour Cheshmehgaz, Mohammad Ishak Desa, and Antoni Wibowo. Effective local evolutionary searches distributed on an island model solving bi-objective optimization problems. *Applied Intelligence*, 38(3):331–356, 2013. 4

[20] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017. 4

[21] K. Roy, A. Jaiswal, and P. Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575:607 – 617, 2019. 1, 5

[22] C. D. Schuman, J. P. Mitchell, R. M. Patton, T. E. Potok, and J. S. Plank. Evolutionary optimization for neuromorphic systems. In *NICE: Neuro-Inspired Computational Elements Workshop*, 2020. 1, 5, 14, 19

[23] C. D. Schuman, J. S. Plank, G. Bruer, and J. Anantharaj. Non-traditional input encoding schemes for spiking neuromorphic systems. In *IJCNN: The International Joint Conference on Neural Networks*, pages 1–10, Budapest, 2019. 8, 12, 33

[24] C. D. Schuman, J. S. Plank, M. Parsa, S. R. Kulkarni, N. Skuda, and J. P. Mitchell. A software framework for comparing training approaches for spiking neuromorphic systems. In *IJCNN: The International Joint Conference on Neural Networks*, pages 1–10, July 2021. 5

[25] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank. A survey of neuromorphic computing and neural networks in hardware. arXiv:1705.06963, May 2017. 5

[26] Catherine D Schuman, Adam Disney, Susheela P Singh, Grant Bruer, J Parker Mitchell, Aleksander Klibisz, and James S Plank. Parallel evolutionary optimization for neuromorphic network training. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 36–46. IEEE, 2016. 4

[27] Catherine D Schuman, James S Plank, Adam Disney, and John Reynolds. An evolutionary optimization framework for neural networks and neuromorphic architectures. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 145–154. IEEE, 2016. 4

[28] Eric O Scott and Kenneth A De Jong. Evaluation-time bias in asynchronous evolutionary algorithms. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1209–1212, 2015. 3

[29] William Severa, Craig M Vineyard, Ryan Dellana, Stephen J Verzi, and James B Aimone. Training deep neural networks for binary communication with the whetstone method. *Nature Machine Intelligence*, 1(2):86–94, 2019. 1, 10

[30] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002. 4

[31] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009. 4

[32] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002. 4

[33] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the genetic and evolutionary computation conference*, pages 497–504, 2017. 4

[34] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. Island model genetic algorithms and linearly separable problems. In *AISB International Workshop on Evolutionary Computing*, pages 109–125. Springer, 1997. 4

[35] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7(1):33–47, 1999. 4

[36] Steven R Young, Derek C Rose, Travis Johnston, William T Heller, Thomas P Karnowski, Thomas E Potok, Robert M Patton, Gabriel Perdue, and Jonathan Miller. Evolving deep networks using hpc. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–7. 2017. 4

[37] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. Optimizing deep learning hyper-parameters through an

evolutionary algorithm. In *Proceedings of the workshop on machine learning in high-performance computing environments*, pages 1–5, 2015. 4

Vita

Chaohui Zheng is originally from Changle, Fujian, China. He graduated from Fujian Changle No.1 Middle School in 2016. Then he attended Volunteer State Community College in 2017. He transferred to the University of Tennessee, Knoxville, to study Computer Science in 2018. After graduation in 2021, he decided to pursue a Master of Science degree in Computer Science at the University of Tennessee, Knoxville. While at the University of Tennessee, he worked as a teaching assistant for a data structure and algorithm class since the fall of 2019. He also worked as a research assistant for TENNLab from the summer of 2020 until August 2022.