A Spiking Neuron Implementation of Genetic Algorithms for Optimization

Siegfried Ludwig[1], Joeri Hartjes[1], Bram Pol[1], Gabriela Rivas[1], and Johan Kwisthout[2][0000−0003−4383−7786]

[1] School for Artificial Intelligence, Radboud University
Montessorilaan 3 6525 HR Nijmegen, Netherlands
siegfried.m.ludwig@protonmail.ch

[2] Donders Center for Cognition, Radboud University
Montessorilaan 3 6525 HR Nijmegen, Netherlands
j.kwisthout@donders.ru.nl

Abstract. We designed freely scalable ensembles of spiking neurons to carry out the operations required to run a genetic algorithm, thereby opening up possibilities for making use of efficient neuromorphic hardware. Two types of implementation are explored that offer a complexity trade-off between computational space and time, with both designs having linear energy complexity. The designs were implemented in a simulator to successfully solve the one-max optimization problem, serving as a proof of concept for running genetic algorithms as spiking neural networks.

Keywords: neuromorphic computing · genetic algorithm · spiking neural networks

1 Introduction

Neuromorphic computing ranges back to the term being coined in 1990 [1], in which the first implementation consisted of very large scale integration (VLSI) with analog components mimicking the biological neural systems. Much research has been done since this time, and in the last few years the energy efficiency of such architectures have become an increasingly dominant research subject. Spiking neural networks (SNN) are known as a type of neuromorphic implementation which have exceptional energy saving properties, compared to other systems [2]. SNNs augment artificial neural networks with the spiking dynamics found in biological neurons [3]. Based on leaky integrate-and-fire (LIF) neurons [4], SNNs transmit information by means of timing and energy spikes, released when the potential difference inside a neuron reaches a certain threshold. This is because such hardware is modeled after the brain in that its activation is event-driven and asynchronous. On top of that, SNN's property of local information storage effectively avoids the von Neumann bottleneck arising from an idling processor while retrieving data from memory [5]. Researching the possibility of implementing various existing algorithms in such SNNs leads the way to a future in which

real life applications of such algorithms currently implemented on von Neumann architectures could be replaced.

Implementing algorithms as SNNs to run them on neuromorphic hardware has been done for sorting [6], constraint satisfaction [7], shortest path and neighborhood subgraph extraction problems [8]. The striking similarity between a genetic sequence and a neural spike train inspires the implementation of a genetic algorithm (GA) as a SNN, which could make use of recent hardware developments in neuromorphic computing.

The use of evolution-inspired algorithms has been proven a viable solution for tackling problems of optimization, bringing in advantages for optimisation over traditional methods. For instance, GA systems [9, 10] may provide the opportunity for difficult problem solving such as multi-objective optimisation [11] and have found applications in various practical settings (see [12] for a review). As in natural evolution, GAs work by modifying the characteristics of individuals in a population across several iterations. This is done by means of reproduction (*crossover*) and random gene mutation. With each run, individuals with an arrangement of genes with a higher fitness value are allowed to preferentially reproduce and carry over their genetic information into the next *generation*. In this study, each individual solution (*chromosome*) is represented as a binary bit sequence, in which each bit represents the value of a gene.

Our main aim was to investigate the feasibility of implementing a GA using spiking neurons with the potential for future implementation on neuromorphic hardware, such as Intel's Loihi [13] or IBM's TrueNorth [14]. Our design consists of binary genetic sequences, which are represented as neuronal spikes and are processed by LIF neurons with context-dependent parameters. The chosen optimization problem is the *one-max problem* due to its simplicity and wide use in the literature on genetic algorithms; the objective of which is to produce a fully active genetic sequence, in this case a fully active spike train. The neural network was implemented and tested using a spiking neuron simulator[3].

We considered two candidate possibilities for encoding the binary genetic sequence in neural ensembles. Firstly, the genetic sequence can be represented sequentially as a spike train, with a spike indicating a 1 and no spike indicating a 0. An ensemble in this design processes one bit at a time. The second way of representing a binary genetic sequence is parallel, using a separate neuron for each position of the genetic sequence. These two encodings are expected to offer a complexity trade-off between computational space and time.

In the following, the high-level architecture of the SNN is presented, followed by details on the sequential and parallel implementations. We then conduct a complexity analysis of both implementations with regard to space, time and energy, in order to assess the tractability of our design.

[3] https://gitlab.socsci.ru.nl/j.kwisthout/neuromorphic-genetic-algorithm

2 High-Level Architecture

The genetic algorithm consists of initializing and evaluating a starting population and then repeatedly performing selection, crossover, mutation, and evaluation on the population until termination. It is implemented as a single recurrent SNN, consisting of specialized neural ensembles for each operation (Figure 1). The topology of the network gives a fitness hierarchy, with the fittest chromosomes being at the top and conversely the least fit chromosomes being at the bottom. The network architecture is static during run-time and no learning of the weights is required.

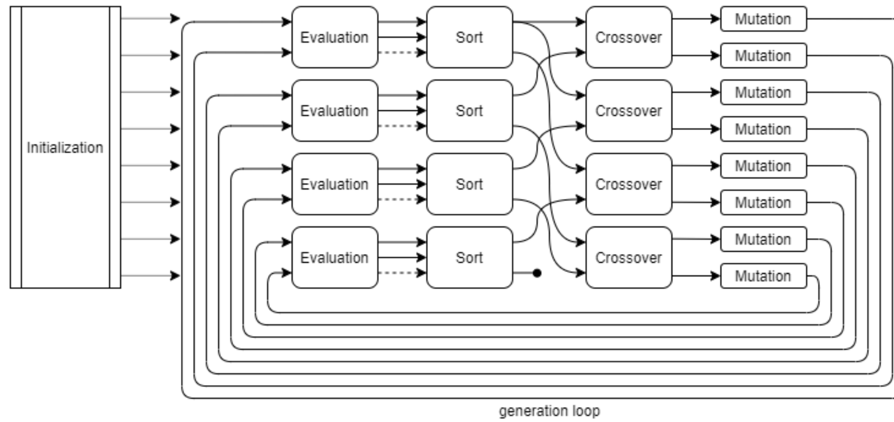

Fig. 1: High-level architecture of the genetic algorithm network, depicted with 8 chromosome lanes as solid arrows (each arrow can represent multiple neural connections in the parallel design). After *mutation*, all chromosomes are connected back to *evaluation* to close the generation loop, resulting in a single large neural network. The dashed arrows from *evaluation* to *sort* represent the evaluation result. To increase the number of chromosomes, the pattern of the second pair of the four lane pairs is repeated.

After initialization, the chromosomes enter the evaluation ensembles in pairs, where they are evaluated against each other and then potentially swapped to bring the chromosome with higher fitness to the top. This setup corresponds to a single pairwise bubble sort step and over time ranks chromosomes by their fitness, which is necessary for selection. The use of only a limited number of bubble sort steps in each generation will lead to incomplete sorting, but is more efficient and leads to some variety in the ranking of the chromosomes while still avoiding the removal of very promising individuals from the bottom. This design is related to the ranking selection mechanism [15].

Selection is implemented in the connections from the sorting ensembles to the crossover ensembles, by eliminating the bottom chromosome and connecting

the top chromosome twice. This results in better solutions propagating more successfully over time. In addition to potentially moving up one lane in the sorting ensemble itself, the winner of the pairwise evaluations moves up by another lane after sorting to ensure upwards mobility, as otherwise the same pairs would be compared each iteration. Conversely, the inferior chromosome moves down by a number of lanes after sorting.

Crossover is then performed on each pair of chromosomes. This reproduction is implemented with a stochastic crossover method, which splits two sequences at a random point and swaps all subsequent genes between the individuals.

After crossover, each chromosome is processed individually in a mutation ensemble. Mutation is carried out by assigning a probability for flipping the activity of each bit in a given sequence. In our designs, we use a probability of $p = \frac{1}{n}$, where n is the length of the chromosome, but other mutation rates are possible. We do not apply mutation on the top two chromosomes of each generation in order to allow for stable one-max solutions. Again, this choice is more up to the design of the genetic algorithm than the implementation as a spiking neural network.

To close the generation loop, the outputs of the mutation ensembles connect back into the evaluation ensembles, forming a recurrent neural network.

Scaling the network up for a larger population size or longer chromosomes is straight-forward beyond a small minimum size, by repeating whole ensembles and repeated elements within certain ensembles.

3 Neural Ensembles for Genetic Algorithms

3.1 Sequential Design

In our sequential design of the GA, the chromosome is processed one bit at a time, which more closely resembles genetic processing in nature. Sequential processing allows a small neural ensemble to process arbitrary lengths of chromosomes over time without itself growing in size. The implementation relies on a lead bit, which precedes every chromosome and is always active. This allows the signaling of the arrival of a new chromosome, ensuring correct processing. In the following, neurons will be ascribed different types based on their function in the ensemble. However, they are all based on the LIF neuron model.

Evaluation Ensemble The sequential one-max evaluation ensemble (Figure 2) makes use of 8 neurons and 11 internal connections. It takes as input two chromosomes and gives as output two chromosomes as well as a spike on a separate neuron serving as an indicator in case the bottom chromosome has a higher fitness than the top chromosome. The membrane potential of the accumulator neuron (ACC) is increased with each active bit in the bottom chromosome, and decreased with each active bit in the top chromosome. Note that the ACC neuron is like all other neuron types used here just a LIF neuron with specific parameters. The activation neuron (A), activated by the lead bit, then makes the ACC

neuron fire or not based on the final membrane potential of the ACC neuron. Only in the case of a membrane potential higher than zero will the indicator neuron fire, and will the chromosomes' ranking switch. A reset neuron (R) is responsible for spiking but suppressing the ACC as to prevent interference of previous chromosome comparisons with current iterations. Clearing the potential of the ACC neuron could alternatively be done using membrane leakage over time, but that would result in a less predictable design.

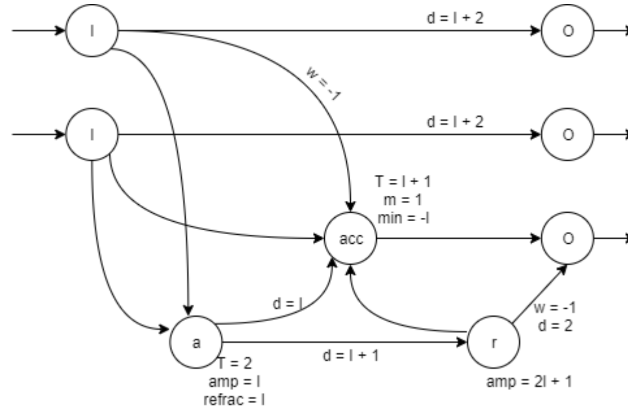

Fig. 2: Sequential one-max evaluation ensemble. (I) Input, (acc) Accumulator neuron, (a) Activation neuron, (r) Reset neuron, (o) Output.

Bubble Sort Ensemble The bubble sort ensemble (Figure 3) consists of 10 neurons and 15 internal connections. It takes two chromosomes plus a fitness indication as input and gives two chromosomes as output. It uses gate (G) neurons to open or close the identity and swap lanes connecting input and output and thereby controlling whether the incoming chromosomes are swapped or propagated as identity. This is achieved by giving the swap gate neurons a threshold of two, which means they can only fire if an input comes from the gate control (GC) neuron. The GC neuron is activated by the gate control activation (GCA) neuron, which takes the fitness indicator input coming from the evaluation ensemble. The GC neuron uses a recurrent connection to keep the swap gates open and the identity gates closed until the chromosomes passed through entirely, at which point it is deactivated by a delayed spike coming from the GCA neuron.

Crossover Ensemble The crossover ensemble (Figure 4) works similarly to the bubble sort ensemble, except that identity and swap gates are not open or closed for the whole chromosome, but switch activation at a random point. It uses 13 neurons and 27 internal connections. The ensemble could be simplified to only use one gate control (GC) neuron as in the bubble sort ensemble, but

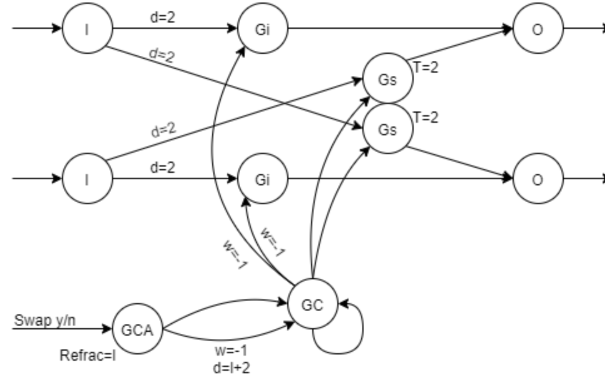

Fig. 3: Bubble sort ensemble in the sequential design. (I) input, (Gi) identity gate, (Gs) swap gate, (GC) gate control, (GCA) gate control activation, (O) output.

has been implemented with two in this project. The random crossover point is implemented via a stochastic (S) neuron and a stochasticity control (SC) neuron. The S neuron gets constant input from the SC neuron, while also generating a random membrane potential each time step. If this combined potential crosses the S neuron's threshold the identity GC neuron is deactivated and the swap GC neuron is activated. If S spikes, The S neuron also deactivates the SC neuron, since only one crossover point is desired.

Mutation Ensemble Finally, the mutation ensemble (Figure 5) stochastically turns a 0 into a 1 and conversely a 1 into a 0, independently for each bit excluding the lead bit. It uses 6 neurons and 12 internal connections. The first stochastic neuron (S1) gets a positive input from each spike in the input and adds a random membrane potential, which can cross the threshold and lead to a spike. A spike from S1 suppresses the ensemble output, thereby turning a 1 into a 0. The other stochastic neuron (S2) always gets input from the control (C) neuron and adds a random membrane potential, but it is suppressed by every spike in the input. If no spike comes from the input, it has a chance of firing and turning the ensemble output from a 0 to a 1. The control neuron is activated and finally deactivated by the control activation (CA) neuron.

Full Network Behavior Each chromosome is passed through the ensembles in its lane, as described in the high-level architecture (see Figure 1). In the sequential design, a chromosome can still be processed in one ensemble while already entering into the next (e.g. crossover to mutation), since here each bit can be handled independently. An exception to this is the evaluation ensemble, which needs to accumulate the full chromosome to make an evaluation. It therefore breaks the time-constant flow through the other ensembles and leads to a time

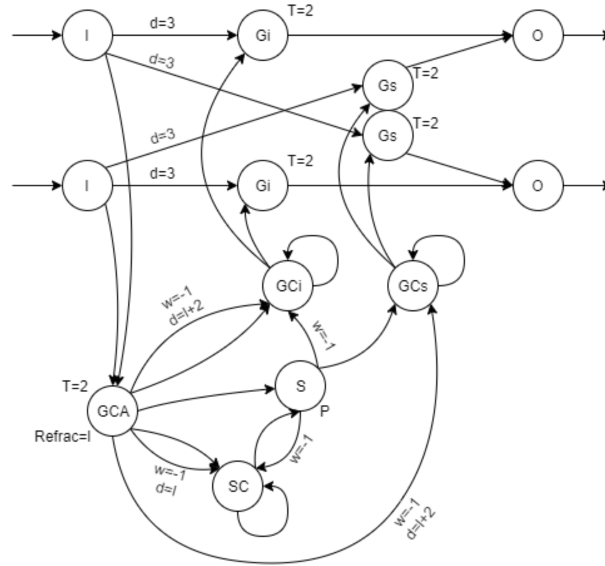

Fig. 4: Crossover ensemble in the sequential design. (I) input, (Gi) identity gate, (Gs) swap gate, (GCi) identity gate control, (GCs) swap gate control, (GCA) gate control activation, (S) stochastic, (SC) stochasticity control, (O) output.

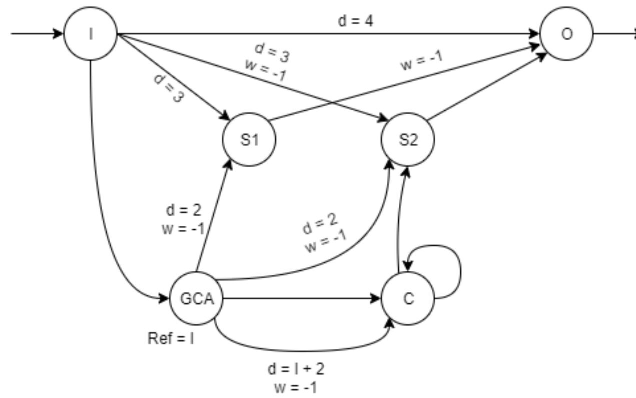

Fig. 5: Mutation ensemble in the sequential design. (I) input, (S1) stochastically flips 1 to 0, (S2) stochastically flips 0 to 1, (C) constant, (GCA) gate control activation, (O) output.

dependency on the chromosome length. On the upside this prevents chromosomes from being longer than the execution cycle, which could otherwise lead to the beginning of the next generation interfering with the end of the last for long chromosomes.

3.2 Parallel Design

In the parallel implementation, every gene of the chromosome gets processed at the same time. Instead of using a single spike train to represent the chromosome, multiple neurons are used that each represent one gene of the chromosome. A set of neurons can then represent the binary code of the chromosome by either spiking or not. Its advantage is that the entire binary code of the chromosomes can be conveyed in a single time step, but requires more neurons as chromosomes get longer. A generation of the entire algorithm in parallel design takes exactly eleven simulation time steps. Again, all neuron types presented here are simple LIF neurons with specific parameters. The different ensembles used in the algorithm will be explained below.

Evaluation & Bubble Sort Ensemble In the parallel design the evaluation step is combined with the bubble sort step. The goal of the evaluation is to have the chromosome with the highest fitness be transferred to the first n output neurons where n is the length of the chromosome. One lead bit is present for each chromosome pair which enables functionality in the other ensembles of the GA, however for this segment it is of no use and therefore linked directly to its corresponding output neuron. Also for this reason the decision was made to omit the lead bit altogether in Figure 6. By taking advantage of all information contained in the chromosome being available at once, the evaluation and sorting ensembles could be combined. This enables the comparison of the fitness through the use of one Accumulator neuron (ACC) to which all input genes are connected (excluding the lead bit). The sign of the connection weights leading to the ACC results in it becoming active only if the lower chromosome has a greater fitness than the top chromosome by at least one gene. Subsequently, the activation of the ACC will determine whether either the identity gates or the swap gates are activated. These are responsible for transferring the activity from the input to the output neuron of the same, or the 'adversarial chromosome', respectively.

Each of the input neurons are connected to both a dedicated identity gate and a dedicated swap gate, with these being connected to the identity neuron or the neuron on the other chromosome in the same position. The connection from the input to the gates is delayed by one time step, however, to allow for synchronous arrival of the spike and the spike coming from the ACC. The connections between the ACC and the gates are weighted such that by default the identity gates have a threshold low enough that a spike from the input neurons will be enough to spike the gate as well while the threshold of the swap gates is too high. As soon as the ACC is activated however this spike is no longer enough for the identity gates, while the extra activation coming from the ACC to the swap gates lowers

their threshold enough to let the spike pass from the input neuron to the correct output neuron on the side of the 'adversarial chromosome'.

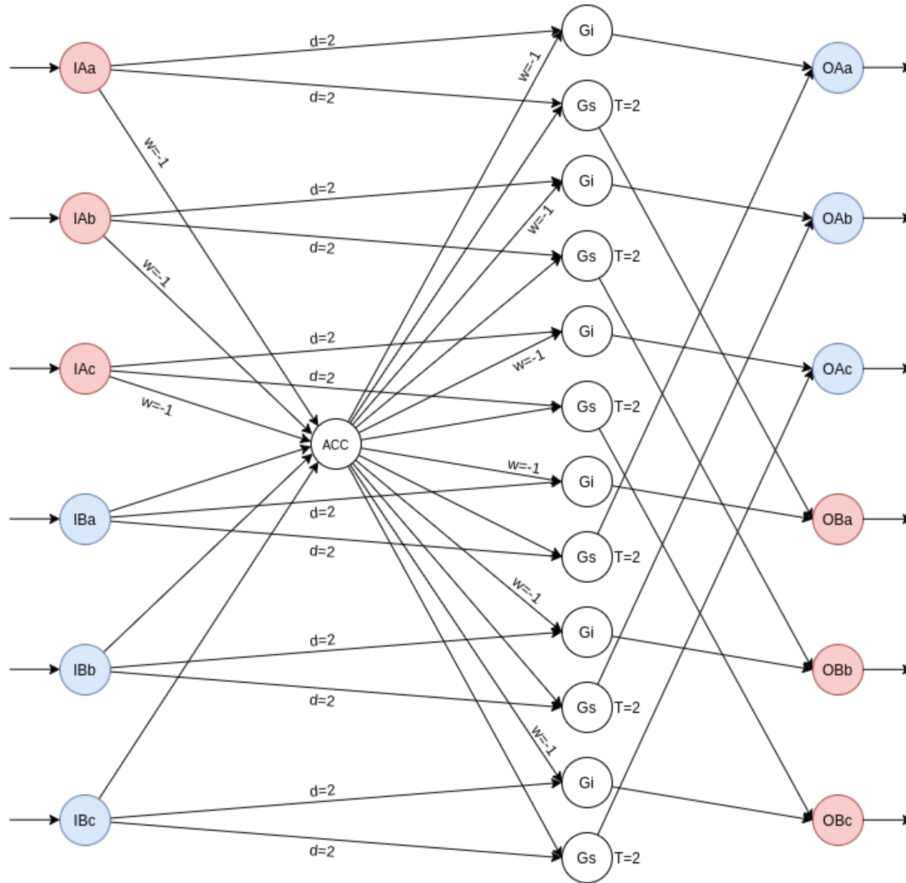

Fig. 6: Ensemble responsible for the evaluation and sorting in the parallel design, applied to a pair of chromosomes consisting of three genes each. Using an accumulator neuron (ACC), the ensemble determines which of the chromosomes has higher fitness and places the winner in the top lanes.

Crossover Ensemble The parallel crossover ensemble can be seen in Figure 7. The first gene of every chromosome always ends up in the same output chromosome. The last gene is always crossed over and ends up in the opposite chromosome. To decide where the genes in between go, a 'random point maker' has been designed (see Figure 8), which is activated by the lead bit. The input to the second layer of the random point maker spikes with a probability of $p = \frac{1}{n-2}$,

where n is the chromosome length. If activated, the node in this second layer transfers this spike to all nodes in the third layer on the same level or below, ensuring that once a gate opens the gates below also open.

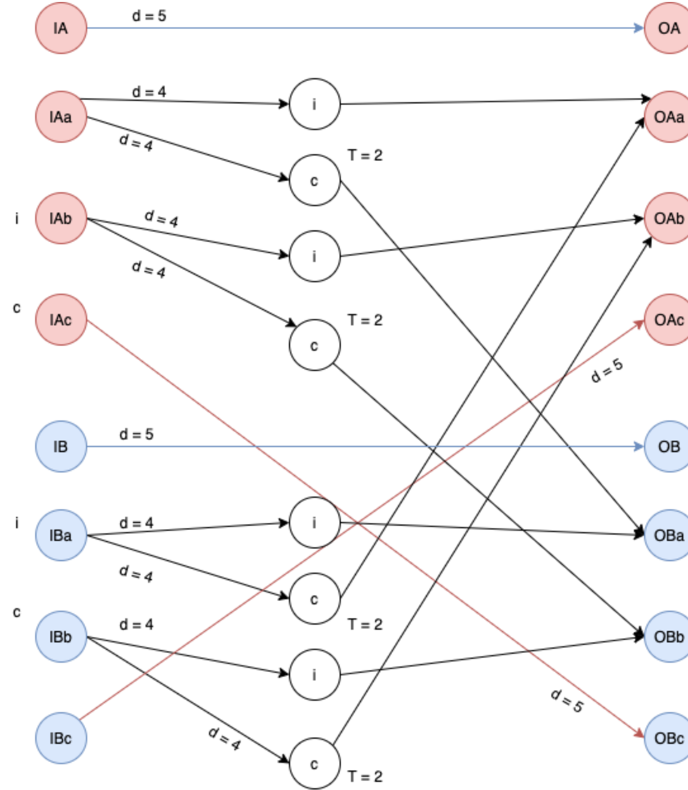

Fig. 7: The parallel crossover ensemble. The first gene of a chromosome is always sent to the same position and the last gene is always crossed over. For the genes in the middle, the random point maker determines whether they are crossed over or not.

The third layer of the random point maker (the gates) connect to the identity and crossover nodes in the crossover ensemble. When a gate neuron of the random point maker gets a spike, it closes the identity gate and opens the crossover gate of both chromosomes at that level. This way, initially the crossover ensemble will transfer genes to the same output chromosome, but at a random point will switch to crossing genes over to the other output chromosome. The crossover ensemble, together with the random point maker, takes five time steps to run for any chromosome length n. The number of neurons in the ensemble is $10n - 11$, meaning linear growth. The number of connections does not show linear growth,

because the connections between the second and third layer of the random point maker grow with $\frac{n^2+n}{2}$, which is quadratic growth. Because of this, the number of connections in the whole ensemble grows quadratically.

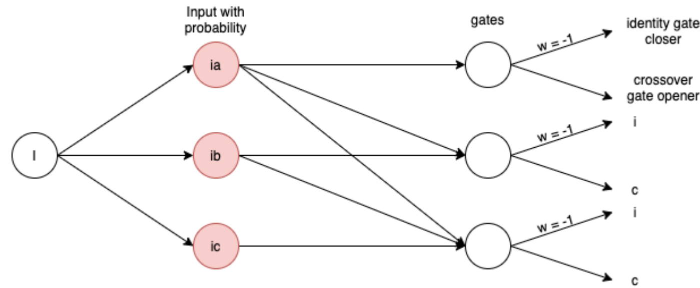

Fig. 8: The random point maker that connects to the gates of the crossover ensemble. This ensemble determines the point of the chromosomes where the identity ends and the swapping of genes with each other starts. It makes sure that there is an equal probability for every point in the chromosome to be the start of crossing over the remaining genes.

Mutation Ensemble The final ensemble in the parallel design is responsible for the stochastic mutation of the genes in the chromosomes, meaning turning a 1 into a 0 or vice versa (Figure 9). The way it is implemented is through assigning a probability P to each of the genes, and therefore neurons, to switch their activity. Except for the lead bit (Ia), every input-neuron (Ib, Ic) is connected to two neurons, and both of their thresholds are influenced by the switching-probability through $T = 2 - P$. A noise factor is present in both intermediate neurons, its function being to add randomness as to whether a neuron will mutate or not. In the diagram the first of the two intermediate neurons is responsible for potentially turning off the activation in case that the input neuron has spiked, and the other is responsible for the opposite. Each of the input neurons is connected directly to its corresponding output neuron, however this connection is delayed such that its spike is delivered synchronously to the potential spike of one of the two intermediate neurons. The role of the lead bit is essential to the mutation ensemble, as its guaranteed activity allows for the potential activation of the two intermediate neurons, which otherwise have no chance of reaching their threshold. Combining the stochastic nature of the intermediate neurons, together with the configuration of the intermediate neurons then has the desired effect of a random mutation of the gene together with the appropriate switching of its value.

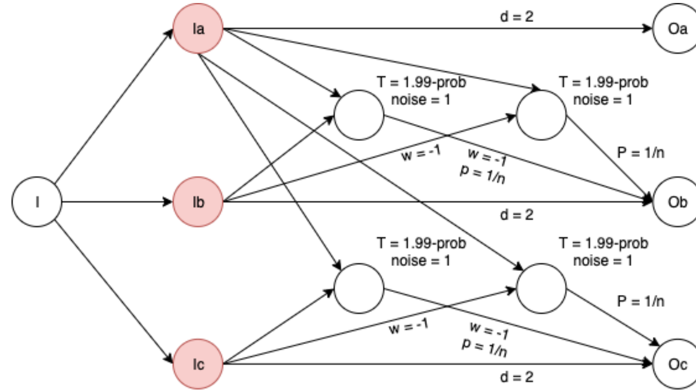

Fig. 9: The parallel mutation ensemble. This ensemble makes sure that every gene's bit has a chance to be swapped.

4 Analysis

A raster plot of the output neurons of the sequential bubble sort ensemble is given in Figure 10. It shows the improving solution quality over time, with the top chromosome reaching one-max, and also shows some resemblance of the fitness hierarchy, with better solutions being closer to the top (subject to imperfect sorting). Figure 11 confirms that the top chromosome in the hierarchy has a higher than average fitness, which specifically shows that even the single pairwise bubble sort step at each generation is enough to at least approximate a fitness ranking.

To assess the tractability of our two designs, a complexity analysis is performed. Computational complexity for neuromorphic computing is considered in terms of space, time, and energy, measured as the number of spikes. For this analysis, all three complexities have been considered with regard to the number of chromosomes and the chromosome length. Comparing the complexity of the sequential and parallel design shows a space-time trade-off between the two (Table 1), with the sequential design requiring less space but more time. Both designs have linear space complexity in the number of chromosomes, both in terms of the number of neurons and the number of connections. The sequential design has much lower space requirements however.

Regarding the chromosome length, the sequential design has constant space complexity, while the parallel design is linear in the number of neurons and quadratic in the number of connections. This is the least favorable of all measured behaviors. It is specifically caused by the current implementation of randomly determining a crossover point. Both designs have constant time complexity in the number of chromosomes, with time measured in simulation steps per generation.

While the parallel design also has constant time complexity in the chromosome length, the sequential design has linear time complexity. The sequential design inherently needs to have at least linear time complexity in the chromo-

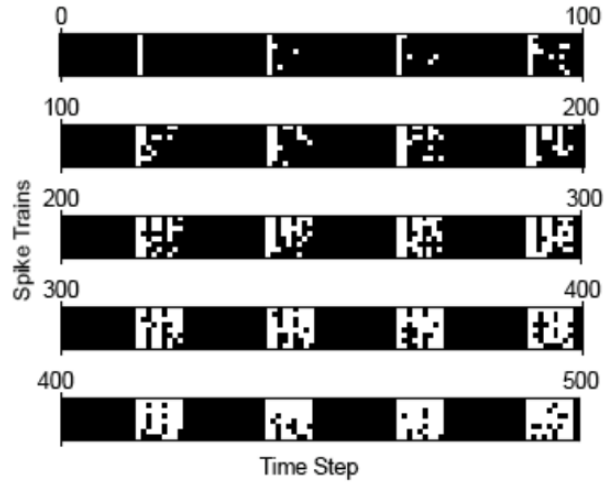

Fig. 10: Raster plot of bubble sort output neurons over time in the sequential design (8 chromosomes of length 8, for 500 steps). Each row of pixels depicts a neural spike train over 500 simulation steps. The solution quality is improving over time, with the top chromosome reaching one-max.

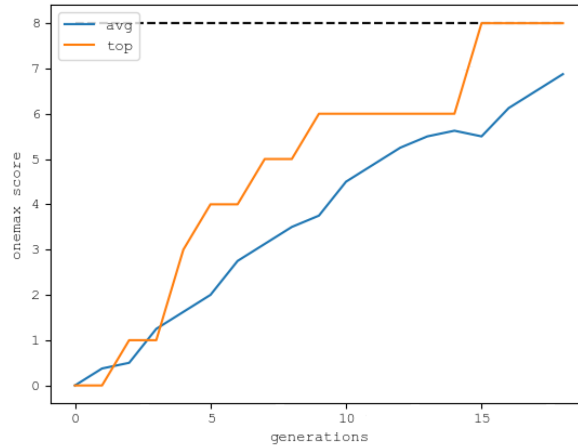

Fig. 11: Average and best solution quality over generations (8 chromosomes of length 8, for 500 steps). The fitness hierarchy results in the top chromosome having better fitness than the average. While this plot comes from the sequential design, the parallel design behaves similarly.

Table 1: Complexity analysis of space, time, and energy (number of spikes) for both the sequential and the parallel design. There is a trade-off between space and time comparing the two designs, with the sequential design requiring less space but more time.

		n_chromosomes		len_chromosomes	
		sequential	parallel	sequential	parallel
space	neurons	$O(n)$	$O(n)$	$O(1)$	$O(n)$
	connections	$O(n)$	$O(n)$	$O(1)$	$O(n^2)$
time		$O(1)$	$O(1)$	$O(n)$	$O(1)$
energy		$O(n)$	$O(n)$	$O(n)$	$O(n)$

some length, as the full chromosome needs to be accessed before an evaluation can be made. This is an advantage for the parallel design, as the full chromosome is available at once. Both designs have linear energy complexity in the number of chromosomes and in the chromosome length, when measuring energy as the average number of spikes required to process one generation.

5 Discussion

A fully functioning genetic algorithm has been successfully implemented as a spiking neural network with two different designs, representing chromosomes sequentially as a spike train over time or as parallel spikes at a single time step. Both implementations are freely scalable beyond a small minimum number of chromosomes, with arbitrary chromosome lengths. The complexity analysis of space, time and energy shows the tractability of this approach with the exception of the quadratically growing number of connections required for the parallel design when increasing chromosome length. The sequential design is at most linear in any of the analyzed complexities.

The design has not yet been implemented on neuromorphic hardware. Since fairly standard leaky integrate-and-fire neurons were used, however, and no learning is required, translating the design to an implementation in neuromorphic hardware should be relatively straight-forward.

For future work, the design of the crossover ensembles could be adapted to support gene lengths of more than one bit (a chromosome consists of a number of genes, which itself could consist of a number of bases/bits). Practically this just means that the random crossover point should only be allowed at transition points between genes, so at fixed intervals. This would allow for more complex behavior of the genetic algorithm.

More work needs to be done on the evaluation strategy, which under the current design requires a unique neural ensemble purpose-built for the optimization task at hand and thereby presents a hurdle for practical application. One possibility for a more general approach would be to train a spiking neural network to

perform approximate evaluations for the given task, instead of hand-engineering the neural ensemble for exact solutions as is performed in this paper.

References

[1] Carver Mead. "Neuromorphic electronic systems". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636.

[2] Catherine D Schuman et al. "A survey of neuromorphic computing and neural networks in hardware". In: *arXiv preprint arXiv:1705.06963* (2017).

[3] Samanwoy Ghosh-Dastidar and Hojjat Adeli. "Spiking neural networks". In: *International journal of neural systems* 19.04 (2009), pp. 295–308.

[4] Anthony N Burkitt. "A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input". In: *Biological cybernetics* 95.1 (2006), pp. 1–19.

[5] Aaron R Young et al. "A review of spiking neuromorphic hardware communication systems". In: *IEEE Access* 7 (2019), pp. 135606–135620.

[6] Samya Bagchi, Srikrishna S Bhat, and Atul Kumar. "O(1) time sorting algorithms using spiking neurons". In: *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2016, pp. 1037–1043.

[7] Chris Yakopcic et al. "Solving constraint satisfaction problems using the loihi spiking neuromorphic processor". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1079–1084.

[8] Catherine D Schuman et al. "Shortest path and neighborhood subgraph extraction on a spiking memristive neuromorphic implementation". In: *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop*. 2019, pp. 1–6.

[9] John H Holland. "Genetic algorithms". In: *Scientific american* 267.1 (1992), pp. 66–73.

[10] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, 1992.

[11] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons, 2001.

[12] Manoj Kumar et al. "Genetic algorithm: Review and application". In: *Available at SSRN 3529843* (2010).

[13] Chit-Kwan Lin et al. "Programming spiking neural networks on Intel's Loihi". In: *Computer* 51.3 (2018), pp. 52–61.

[14] Paul A Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673.

[15] L Darrell Whitley et al. "The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best." In: *Icga*. Vol. 89. Fairfax, VA. 1989, pp. 116–123.