Institut für Telematik
Pervasive Computing Systems / TECO
Prof. Dr.-Ing. Michael Beigl

Neural Evolution for Augmenting Topologies in Printed Neuromorphic Circuits

Master's Thesis

by

Yuhong Wang

Department of Informatics

Responsible Supervisor: Prof. Dr. Michael Beigl

Supervising Staff: Haibin Zhao

Project Period: 15/03/2023 – 15/09/2023

Abstract

In the context of fast-paced advancements in the Internet of Things and artificial intelligence, the demand for custom solutions in emerging fields like smart packaging and smart bandages has risen significantly. These innovative applications require electronics that are not only ultra-low-cost but also highly flexible and customizable, particularly for edge processing tasks. Traditional silicon-based electronics often fall short in offering cost-effectiveness. In contrast, printed electronics have emerged as a powerful alternative. They use additive manufacturing techniques to create custom electronic circuits at ultra-low cost. These electronics are particularly versatile, with a choice of materials and substrates contributing to notable flexibility and bio-compatibility. To further equip them with computational abilities, there is an increasing interest in printed neuromorphic circuits. These circuits effectively merge the benefits of neuromorphic computing with the capabilities of printed electronics. Nevertheless, the intrinsic constraints of printed electronics, namely large feature sizes and notably reduced integration density, pose challenges for compact application areas. To address these challenges, we propose a method inspired by the Neural Evolution for Augmenting Topologies algorithm. Unlike traditional gradient-based optimization methods, such as pruning, our approach simultaneously optimizes the learnable parameters and the topology structure of printed neuromorphic circuits, thereby facilitating a reduction in circuit area. Through experiments conducted on 11 different datasets, we demonstrate the effectiveness of our approach. Experimental results reveal that, with the proposed approach, 3.1× reduction of the circuit area can be realized while maintaining 100% of classification accuracy of the gradient-based pruning method.

Keywords: Neural Architecture Search, Printed Electronics, Neuromorphic Circuits, Evolutionary Algorithms, Topology Optimization.

Contents

List of Figures

List of Tables

1. Introduction

With the rapid advancements in the the Internet of Things (IoT) and artificial intelligence (AI), there is a growing demand for solutions tailored to new applications, such as smart packaging [1] and smart bandages [41]. These applications call for ultra-low-cost, super-soft, and highly customizable electronics for the measuring and processing signals at the edge [49]. However, even the cheapest traditional silicon-based electronics, such as Micro-controller Units[1], fall short in delivering the desired cost-efficiency.

In this regard, printed electronics (PE) emerge as a promising alternative. Since PE can be manufactured additively, high-customized circuits can be produced at (sub-cent) ultra-low costs. Additionally, the appropriate choice of materials and substrates allows these printed devices to demonstrate remarkable flexibility [22, 8], and bio-compatibility [19].

In order to enhance the efficiency of PE across various applications and equip PE with computational capabilities, there has been a growing shift towards printed neuromorphic circuits [43]. This is due to their combination of the advantages of neuromorphic computing [34] with the capabilities of PE. Such circuits can emulate basic operations of artificial neural network (ANN), such as weighted-sum and nonlinear activation functions, through a combination of straightforward circuit primitives like resistor crossbars and printed non-linear circuits. Additionally, as this emulation requires only a stack of multiple simple circuit primitives whose operations are fully differentiable, it enables a highly efficient design and optimization process through gradient-based approaches.

However, printed neuromorphic circuits still suffer from the inherent limitations of PE, namely large feature sizes and reduced integration density. Such constraints present a particularly challenging situation for applications like smart bandages and compact intelligent packaging, where the size of the circuit area is critical. In order to reduce the circuit area, the topology of printed circuit, especially the number of neurons and their connections, must be optimized. But the common training

[1]https://pic-microcontroller.com/world-top-10-cheapest-microcontrollers-mcus

methods for neuromorphic circuits, based on the chain rule, are intrinsically gradient-based optimization processes. These methods are generally inadequate for circuit topology optimization, as topology optimization is a discrete process that doesn't offer meaningful gradient information.

Considering this concern, we decided to utilize evolutionary methods to train the printed neuromorphic circuit without necessitating gradient information. Drawing inspiration from Neural Evolution for Augmenting Topologies (NEAT), we propose a Evolutionary Algorithm (EA) to optimize both the topology and learnable parameters of printed circuits. Through experiments on 11 datasets, we demonstrated the superior performance of our proposed algorithm. Compared to the baseline methods, it achieved a more compact circuit area while maintaining equivalent accuracy, e.g., 3.1$\times$ reduction in area could be achieved without any accuracy decrease compared to the gradient-based pruning method.

The organization of the rest of this thesis is structured as follows:

- Chapter 2 provides an overview of PE, analog printed neuromorphic circuits, printed neural network and relevant literature. At the end of this chapter, a comparison is made among various neural network topology optimization methods, namely Neural Architecture Search (NAS), and the reasons for choosing the EA are explained.
- Chapter 3 introduces the general process and features of the NEAT algorithm.
- Chapter 4 details the modeling of the circuit area and explain the key parts of our method for training printed neuromorphic circuits.
- Chapter 5 first presents the experimental setup for our proposed approach and then this approach is evaluated and discussed.
- Chapter 6 concludes this thesis.

2. Background

In this chapter, we will provide readers with the necessary background and knowledge.

2.1 Printed Electronics

Printed electronics has gained significant attention and undergone notable development in recent years, providing a novel approach to manufacturing electronic components and devices through printing processes [12]. The fundamental concept of PE technology aims to achieve cost-effective, flexible, and highly customizable manufacturing of electronic devices by utilizing printing technology for additive component fabrication.

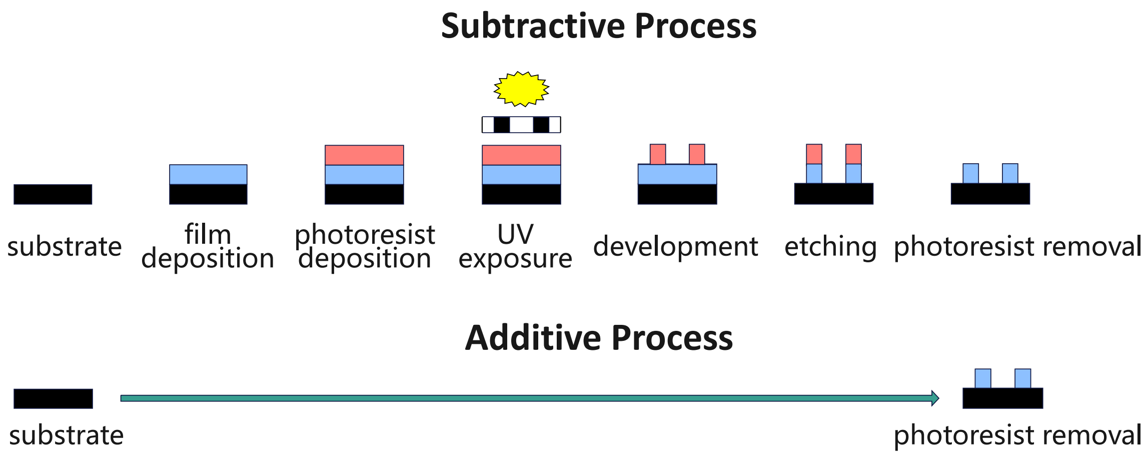


Figure 2.1: Difference between the lithograph-based subtractive process and an additive process for manufacturing electronic circuits.

Figure 2.1 shows the main difference between the lithograph-based subtractive process for silicon-based electronics and an additive process for manufacturing printed electronic circuits. Compared to conventional lithograph-based silicon electronics, PE requires less infrastructure and procedures for fabrication, and thus, exhibits lower manufacturing cost.

PE employ various printing techniques, including gravure printing, flexographic printing, screen printing, and inkjet printing, to fabricate electronic assemblies. These techniques can be categorized into contact printing processes (e.g., gravure, flexographic, and screen printing) and non-contact printing processes (e.g., inkjet printing). Contact printing technologies offer advantages such as low cost, fast production speeds, good repeatability, and suitability for high-volume production. Conversely, non-contact printing technology is slower but excels in digital creation or transmission of information on substrates without the need for printing plates, making it ideal for prototyping and on-demand production. Diverse printing methods offer tailored advantages to meet specific application requirements. These techniques precisely deposit functional materials, including conductive inks and semiconducting materials, to form essential electronic components like transistors, sensors, and radio frequency identification (RFID) devices. The key properties and parameters of the different printing techniques used PE are summarized in Table 2.1 [18, 17].

Printing Process	**Film Thickness (µm)**	**Printing Speed (m/min)**	**Resolution (µm)**	**Image Carrier**
Screen	3–60	0.6-100	30	Stencil
Inkjet	0.01-0.5	0.02-5	20	Digital
Flexo	0.17-8	5-180	15	Polymer Plates
Gravure	0.02-12	8-100	15	Engraved Cylinder

Table 2.1: Key properties and parameters of printing techniques used in printed electronics.

The potential applications of PE span a wide range of industries and continue to expand. In the realm of wearable electronics, PE enables the development of conformable sensors [23] and displays [26] that can seamlessly integrate into clothing and accessories. This breakthrough technology empowers continuous health monitoring, real-time data tracking, and immersive user experiences. For instance, Sun et al. [41] developed an electronic smart band-aid, which can be used not only for wound protection but also for self-powered sensing of movements and human-machine interaction. Furthermore, PE finds practical application in smart packaging [1], where it embeds interactive features like temperature sensors, expiration indicators, and security tags. These intelligent packages enhance consumer engagement, ensure product freshness and authenticity, and optimize supply chain logistics.

In summary, PE represents an emerging technology that enables ultra-low-cost, flexible, and highly customizable manufacturing of electronic devices through printing processes and a spectrum of materials.

2.2 Printed Analog Neuromorphic Circuit

Printed neuromorphic circuits represent a cutting-edge advancement in electronic circuitry, seamlessly merging the principles of neuromorphic computing with the versatility of PE. These circuits emulate the intricate functionality of ANNs, a cornerstone of modern machine learning applications.

Neuromorphic computing is a groundbreaking computational paradigm that draws inspiration from the intricate signal processing mechanisms of the human brain, which has demonstrated remarkable expressiveness across diverse domains [44]. This approach aims to imitate the brain's neural networks and their operations, such as weighted-sum calculations followed by nonlinear activations. Various hardware-level implementations have been developed for digital neuromorphic systems [34]; However, these digital approaches are not optimally aligned with PE requirements due to the large feature sizes and lower integration densities of PE. As a result, the analog neuromorphic computing strategy has garnered increasing interest within the PE sector, given its capability to significantly reduce the need for hundreds of transistors in digital circuits [43]. A fundamental aspect of analog neuromorphic computing lies in its capability to directly handle sensory data, bypassing the need for converting it into digital signals. This avoids the necessity for costly analog-digital converters (ADCs), which would otherwise be essential in the signal processing pipeline. Furthermore, machine learning-based techniques can be employed for training of analog printed neuromorphic circuits, guaranteeing an efficiency design and optimization process. These analog printed neuromorphic circuits enable handle basic sensor processing tasks and can find application in various scenarios. For instance, they can be used to monitor wound characteristics [41] such as temperature, pH levels, and the levels of substances like glucose, uric acid, and lactic acid. They can also be employed to assess the freshness of fruits [1], as well as measure the skin's electrical activity in human body measurements [47].

Analog printed neuromorphic circuits consist of two main components: a matrix multiplication unit and a nonlinear activation unit. The former is implemented by printed resistor crossbars, whose values are usually obtained by learning. The latter is implemented by printed nonlinear circuits, which are usually fixed for all tasks during training. Each neuron in the printed neuromorphic circuit is composed of these two units, which can perform the basic operations of neural networks: weighted sum and activation function. The matrix multiplication unit can realize a large-scale parallel computation of weighted sums for multiple neurons, while the nonlinear activation unit can introduce nonlinearity and sparsity into the neural network output. Furthermore, in order to overcome the restrictions imposed by the hardware, the design could encompass additional subcircuits, such as those for negative weight circuits. By connecting multiple printed neurons in series or in parallel, complex neural network architectures can be constructed. A detailed introduction to each individual component will be provided in the subsequent sections.

2.2.1 Resistor crossbar

The resistor crossbar array stands as a fundamental architecture within printed neuromorphic circuits, mirroring the weighted-sum operations intrinsic to ANNs. The weighted-sum operation serves as a standard process for conducting ANN inference. This innovative architecture takes advantage of resistive elements to enable parallel and distributed computing, paving the way for efficient and high-density information processing.

Resistor crossbar structures find versatile applications across various domains. In the context of memory applications, they serve as the foundation for non-volatile memory devices like Resistive Random-Access Memory (ReRAM)-based graph processing accelerator [36]. By harnessing the principle of resistive switching, where a material's resistance can be altered between high and low states, the crossbar architecture enables efficient and high-density data storage. Additionally, these structures play a crucial role in in-memory computing paradigms [35], where computational tasks are performed directly within the memory. This minimizes the need for extensive data movement between memory and processing units. Leveraging the inherent analog computing capabilities of resistor crossbars, this approach offers energy-efficient solutions for in-memory computations.

Figure 2.2 demonstrates a typical resistor crossbar in a printed neuromorphic circuit with two input voltages V_1, V_2 and one output voltage V_z. In the crossbar, there are also two internal constant voltages, namely $V_{\mathrm{b}}^{\mathrm{C}} = 1V$ and GND that refers to 0V. Each external input voltage and internal constant voltage is connected to the corresponding resistor, namely R_1^{C}, R_2^{C}, $R_{\mathrm{b}}^{\mathrm{C}}$, and $R_{\mathrm{d}}^{\mathrm{C}}$. Here, the superscript $(\cdot)^{\mathrm{C}}$ indicates the variables in the crossbar subcircuit. Following Kirchhoff 's law [21], we obtain

$$\sum_i \frac{V_i - V_z}{R_i^{\mathrm{C}}} + \frac{V_{\mathrm{b}}^{\mathrm{C}} - V_z}{R_{\mathrm{b}}^{\mathrm{C}}} - \frac{V_z}{R_{\mathrm{d}}^{\mathrm{C}}} = 0.$$

By expressing the resistance R as the corresponding conductance $g = 1/R$, this equation can be formulated to

$$V_z = \sum_i \frac{g_i^{\mathrm{C}}}{g_{\mathrm{sum}}^{\mathrm{C}}} V_i + \frac{g_b^{\mathrm{C}}}{g_{\mathrm{sum}}^{\mathrm{C}}}, \tag{2.1}$$

where $g_{\mathrm{sum}}^{\mathrm{C}}$ refers to the sum of all conductances in the crossbar, namely g_1^{C}, g_2^{C}, $g_{\mathrm{b}}^{\mathrm{C}}$ and $g_{\mathrm{d}}^{\mathrm{C}}$. Here, $g_i^{\mathrm{C}}/g_{\mathrm{sum}}^{\mathrm{C}}$ corresponds to the weights in an ANN, while $g_{\mathrm{b}}^{\mathrm{C}}/g_{\mathrm{sum}}^{\mathrm{C}}$ corresponds to the bias on each neuron. Therefore, the crossbar's output can be described by the weighted-sum of input voltages, where the weights and bias are represented by conductance values. This approach enables the realization of the targeted weighted-sum operation through the precise printing of designated conductance values within the crossbar structure. Section 2.3 explains further on the design aspects of resistors within the crossbar configuration.

2.2.2 Printed non-linear circuit

A non-linear activation function represents the another element essential for computations within ANNs as outlined in [7]. This function is directly applied to the output derived from the weighted-sum operation as illustrated in Figure 2.2. In the domain of printed neuromorphic circuits, various nonlinear circuits with characteristic curves resembling classic activation functions have been proposed. Examples of these include the Rectified Linear Unit (ReLU) function [42] and the sigmoid function [27].

An inverter-based printed non-linear circuit is depicted in Figure 2.3. The advantage of this circuit lies in the presence of a super-linear interval between the input and output voltages. This characteristic effectively reduces signal loss at the output of the layer and provides possibilities for cross-layer amplification [43].

The nonlinear circuit's transfer characteristic can be elucidated through a modified version of the hyperbolic tangent (tanh) function, specifically referred to as follows:

$$V_a = \mathrm{ptanh}(V_z) = \eta_1^{\mathrm{A}} + \eta_2^{\mathrm{A}} \cdot \tanh((V_z - \eta_3^{\mathrm{A}}) \cdot \eta_4^{\mathrm{A}}),$$

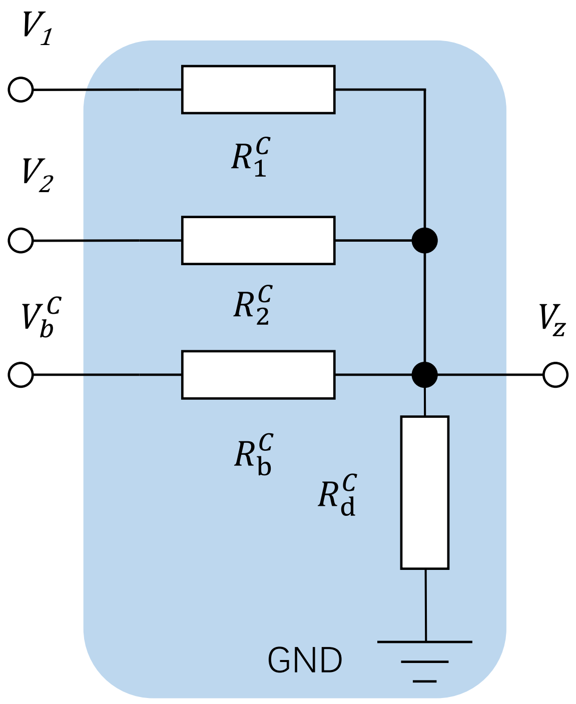


Figure 2.2: The schematic of the 2-input crossbar (V_1 , V_2) implementing the weight-sum operation.

where $\boldsymbol{\eta}^{\mathrm{A}} = [\eta_1^{\mathrm{A}}, \eta_2^{\mathrm{A}}, \eta_3^{\mathrm{A}}, \eta_4^{\mathrm{A}}]$ is the vector of auxiliary parameters that translate and scale the original tanh function [43]. The superscript $(\cdot)^{\mathrm{A}}$ signifies the association of the parameters with the activation function. In this work, we employ a consistent printed non-linear circuit, where the auxiliary parameters are $\boldsymbol{\eta}^{\mathrm{A}} = [0.290, 0.710, -0.017, 20]$.

2.2.3 Negative weight circuit

By observing equation 2.1, it becomes clear that the weight configurations within the crossbar are expressed by the conductances, which are constrained exclusively to positive values. Negative multiplications between inputs V_i and weights $g_i^{\mathrm{C}}/g_{\mathrm{sum}}^{\mathrm{C}}$ (e.g., $g_i^{\mathrm{C}}/g_{\mathrm{sum}}^{\mathrm{C}} < 0$) as well as the bias $g_{\mathrm{b}}^{\mathrm{C}}/g_{\mathrm{sum}}^{\mathrm{C}}$ are not possible, as the resistances are physically only positive. To overcome this problem and realize negative weights, [43] propose an inverter-based transfer function (depicted in Figure 2.4). This function transforms positive neuron input voltages into negative voltages, essentially mimicking the operation of $V_i \cdot (-1)$. In other words, the realization of negative weights and bias in resistor crossbar is achieved through the inversion of the corresponding inputs.

One notable advantage in contrast to other established methodologies [2] lies in the selective utilization of the negative weights circuit. This circuit is employed exclusively when required, strategically positioned ahead of crossbar resistors that necessitate weight negation [46]. Thus, the resulting circuits offer less circuit area, less power consumption and reduced transistor counts.

The transfer characteristic curve of the negative weight circuit exhibits nonlinearity and, similar to the printed tanh (ptanh) circuit, can be expressed using a modified negative tanh function as follows:

$$\mathrm{neg}(V_{in}) = -(\eta_1^{\mathrm{N}} + \eta_2^{\mathrm{N}} \cdot \tanh((V_{in} - \eta_3^{\mathrm{N}}) \cdot \eta_4^{\mathrm{N}})),$$

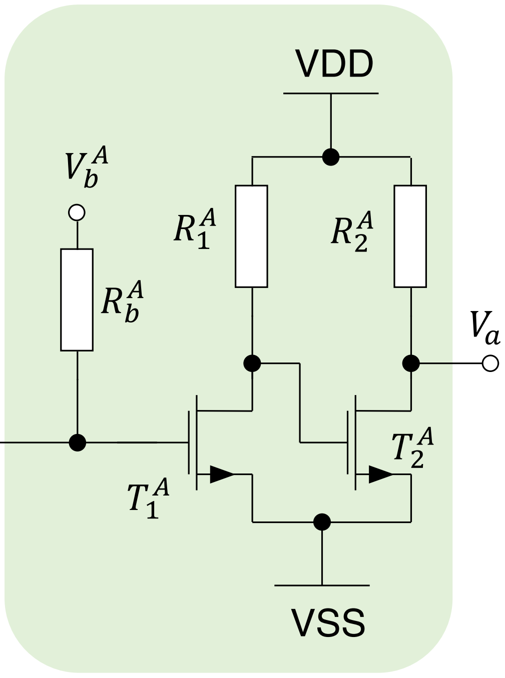


Figure 2.3: The schematic of inverted-based activation function for realizing the hyperbolic tangent (tanh) function.

where $\boldsymbol{\eta}^{\mathrm{N}} = [\eta_1^{\mathrm{N}}, \eta_2^{\mathrm{N}}, \eta_3^{\mathrm{N}}, \eta_4^{\mathrm{N}}]$ is the vector of auxiliary parameters that translate and scale the original tanh function. The superscript $(\cdot)^N$ signifies the association of the parameters with the negative weight circuit. In this work, we employ a consistent negation circuit, where the auxiliary parameters are $\boldsymbol{\eta}^{\mathrm{N}} = [-0.006, 1.024, 0.016, 1.006]$.

By interconnection of these three primitive subcircuits, printed neurons can be constructed, with arbitrary number of neural inputs. Subsequently, based on the printed neuron design, this framework enables the realization of printed neuromorphic circuits tailored to deal with complicated computing tasks.

2.3 Printed Neural Network

By interconnecting the aforementioned circuit primitives, printed neuromorphic circuits exhibit the potential to achieve computational capabilities. To fully exploit the computational capacity of printed neuromorphic circuits, a design and optimization stages is necessary. This optimization process takes into account the targeted application domains of PE, while also addressing the demand for cost-effective solutions. Reconfigurable components are not employed during the operation of these circuits for on-device training; instead, an off-device, software-centric design approach is adopted. Once the circuit design is confirmed, the manufacturing phase begins. To this regard, the concept of a printed neural network (pNN) is introduced [43]. The pNN, as a machine-learning-driven model, aims to simulate the behavior of printed neuromorphic circuits. Within the pNN framework, the learnable parameters correspond to the component values in the actual printed neuromorphic circuits. The general workflow entails initially determining the network's topology, including the node count and layers. By training the pNN on the target dataset (dataset of

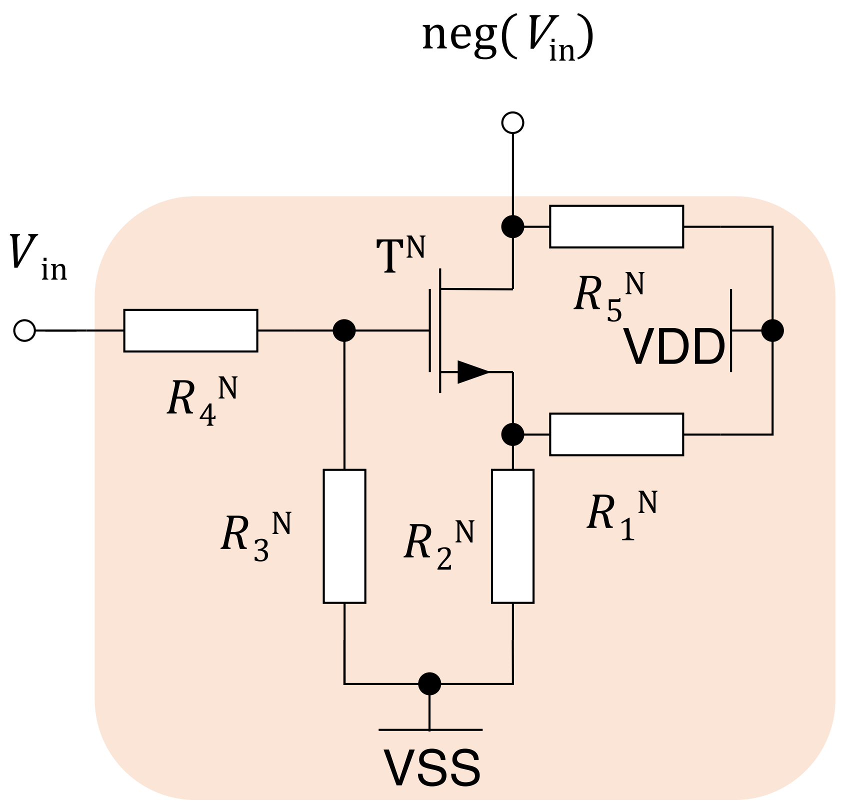


Figure 2.4: The schematic of the negative weight circuit.

the specific task), optimal parameter configurations can be obtained; subsequently, through various applicable printing techniques, on-demand manufacturing can be achieved conveniently and cost-effectively. In conclusion, the training process of the pNN can be perceived as the design and optimization process for the corresponding printed neuromorphic circuit.

In pNN, the learnable parameter for weighted-sum, referred to as the surrogate conductance θ_i. The surrogate conductances encode the value of a respective conductance through their absolute value, i.e., $g_i^C = |\theta_i|$, while the sign of θ_i encodes if the input to the respective resistor should be inverted (negative weight),i.e., sign(θ_i). Through this, the weights in a printed neuron is modelled as

$$w_i = \frac{g_i^{\mathrm{C}}}{g_{\mathrm{sum}}^{\mathrm{C}}} = \frac{|\theta_i|}{\sum_j |\theta_j|},$$

Consequently, the weighted-sum can be expressed by

$$\sum_i V_i \cdot w_i \cdot \mathbb{1}_{\{\theta_i \geq 0\}} + \mathrm{neg}(V_i) \cdot \ w_i \cdot \mathbb{1}_{\{\theta_i < 0\}}, \tag{2.2}$$

where $\mathbb{1}_{\{\cdot\}}$ is an indicator function returning 1 if the respective condition is true, else 0. It is worth noting that the equation 2.2 can encompass the bias term and GND term by augmenting the input voltage with a value of 1 (for the bias term, denoted as V_b^{C}) and a value of 0 (for the GND term).

Consequently, the weight-sum will pass through the ptanh function for activation, the output of a printed neuron can be obtained by

$$\mathrm{ptanh}(\sum_i V_i \cdot w_i \cdot \mathbb{1}_{\{\theta_i \geq 0\}} + \mathrm{neg}_{\mathbf{q}^{\mathrm{N}}}(V_i) \cdot\ w_i \cdot \mathbb{1}_{\{\theta_i < 0\}}). \tag{2.3}$$

2.4 Circuit Compactification

In modern electronics, the importance of compact circuit design is increasingly evident, especially as mobile and IoT devices continually demand smaller sizes, lower power consumption, and enhanced performance. A compact design not only allows electronics to be more streamlined and lightweight but also potentially reduces material and manufacturing costs.

Such designs typically feature shorter interconnections and smaller circuit components, which contribute to decreased resistive and capacitive losses, thereby further reducing power consumption.

From a consumer's perspective, the reduced size and energy consumption undoubtedly make products more appealing, catering to desires for portability and extended battery life.

Furthermore, compact circuit designs enhance flexibility, enabling adaptation to a variety of applications and packaging requirements. A higher level of integration means more functionalities can be accommodated within smaller spaces. For PE, the push towards compactness is equally pronounced, especially in space-constrained applications like on-skin computing [40] and near-sensor computing [49]. To our knowledge, no new materials or manufacturing techniques currently exist that could further miniaturize neuromorphic circuits. However, considering the inherent error-tolerance of neuromorphic computing, approximate computing [32, 3] can be employed to simplify circuit designs. The incorporation of multifunctional components is another path; for instance, certain ReRAMs can double as storage units and compute elements, promoting a more compact design approach.

Given the strong correlation between circuit area and its topology, our approach emphasizes optimizing the topology of neuromorphic circuits. Specifically, we aim to reduce the number of neurons and their interconnections to achieve a more compact design.

2.5 Neural Architecture Search

The optimization of the topology structure of pNN is achieved through a novel field known as Neural Architecture Search (NAS). NAS is an automated methodology used to discover the optimal neural network architecture specifically tailored to a given task. Traditional approaches heavily rely on the expertise and intuition of human experts for the design of deep learning models. However, as neural networks continue to grow in scale and complexity, manual tuning becomes increasingly challenging and time-consuming. The primary goal of NAS is to replace manual tuning by employing automated search methodologies to identify network architectures that are superior in performance and efficiency.

Additionally, NAS methods can identify effective architecture that traditional manual design methods may overlook, introducing a variety of network architectures. NAS finds application in diverse tasks such as image classification, speech recognition, and object detection, and it can be adapted to tasks that require specific model architectures.

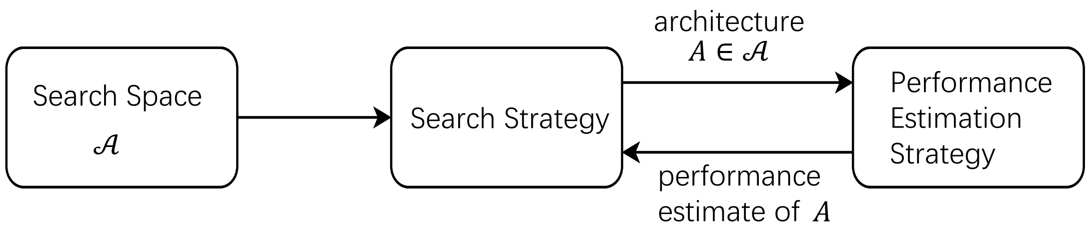


Figure 2.5: An overview of neural architecture search pipeline [13].

The core elements of NAS algorithms are the search space, search strategy, and performance estimation strategy [13]. The search space encompasses the collection of neural network architectures that can be explored, forming the solution space. The search strategy defines the approach used to explore the search space. The performance estimation strategy evaluates the performance of a possible network architexture. Figure 2.5 illustrate the NAS search pipeline.

Diverse search strategies are available for investigating the neural architecture space, including evolutionary methods, reinforcement learning (RL), and gradient-based methods. Below, we detail the commonly used NAS methods.

2.5.1 Reinforcement Learning-Based Optimization

Zoph et al. [50] were pioneers in employing RL for NAS. RL comprises five main components: the agent, the state, the action, the environment, and the reward. Within the NAS paradigm, the agent is the controller, often realized using a Recurrent Neural Network (RNN) [10]. The controller learns to propose or select neural architectures by interacting with an environment. The state typically represents the current neural architecture or parts of it. Actions refer to modifications to the architecture, such as adding layers, changing layer types, or adjusting hyperparameters. Once an action is taken and a new architecture is proposed, the architecture is trained and validated, producing a performance metric (e.g., accuracy or loss value). This metric serves as the reward that is fed back to the agent.

Over multiple rounds, the controller learns from prior decisions and their associated rewards. It iteratively refines its proposals during subsequent iterations, aiming to maximize the reward, leading to the discovery of neural architectures that exhibit superior performance. The process of a RL-based NAS algorithm is illustrated in Figure 2.6.

Different RL approaches differ in how they represent the controller's policy and how they optimize. For instance, Zoph et al [50] initially trained this network with REINFORCE policy gradient algorithm, but subsequently adopted proximal policy optimization in their further studies [51]. In contrast, Baker et al. utilized Q-learning as their chosen strategy to direct the controller's policy updates [5].

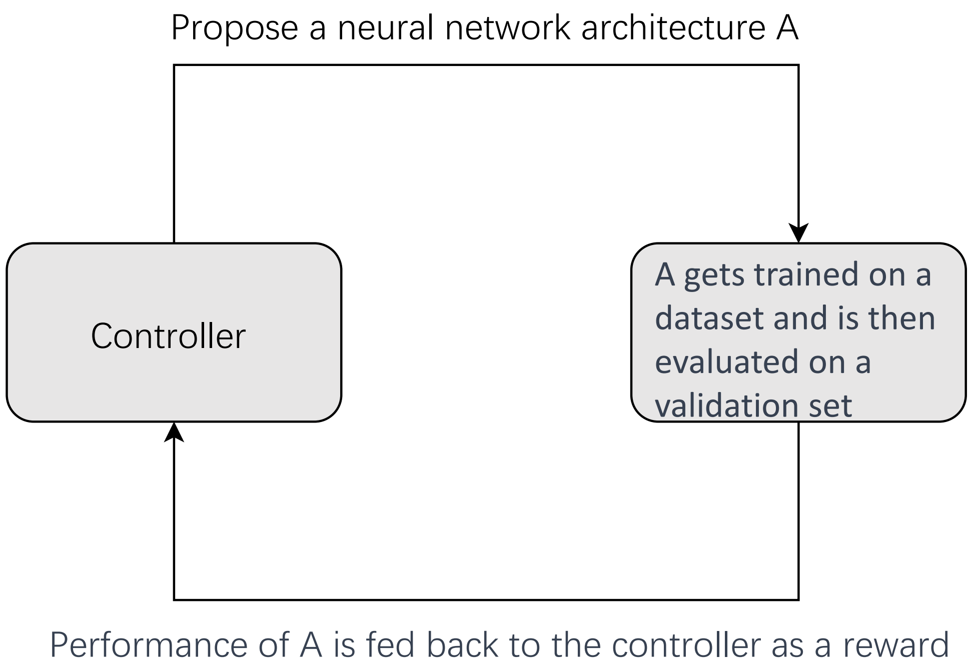

Propose a neural network architecture A
Controller
A gets trained on a dataset and is then evaluated on a validation set
Performance of A is fed back to the controller as a reward

Figure 2.6: An overview of reinforcement learning-based neural architecture search pipeline.

In comparison with other NAS techniques, RL-based NAS exhibits superior scalability, although at a higher computational expense. For instance, the authors in [50] took 28 days and 800 K40 GPUs to search for the best-performing architecture, and a subsequent simplification of the search space in a later work still necessitated 4 days employing 450 GPUs [51]. MetaQNN [5] also took 10 days and 10 GPUs to complete its search.

Liu et al. introduced a method called Progressive Neural Architecture Search (PNAS) [24], which adopts a hierarchical approach to progressively refine the search space. Additionally, the method employs approximate evaluation techniques to evaluate the performance of the architectures. This method is up to 5× more efficient than the RL method of Zoph et al. [51] in terms of number of models evaluated, and 8× faster in terms of total compute.

The Efficient Neural Architecture Search (ENAS) [29] exceeds this approach by adopting a parameter-sharing strategy. In this strategy, all sample architectures are treated as sub-graphs within a supernet. This approach facilitates parameter sharing among architectures, eliminating the requirement to train each child model independently. Consequently, ENAS accomplished the search for the optimal architecture on the CIFAR-10 dataset in approximately 10 hours using a single GPU, a speed nearly 1000× faster than that of the method introduced by Zoph et al. [50].

2.5.2 Evolutionary-Based Optimization

EAs draw inspiration from the principles of natural selection and biological evolution [4]. These optimization algorithms operate on a population of potential solutions, iteratively evolving this population across generations to enhance solution quality. Figure 2.7 shows the overview of the EAs.

In the context of NAS, evolutionary-based NAS initializes with a population of diverse neural network architectures. Each member of this population carries its

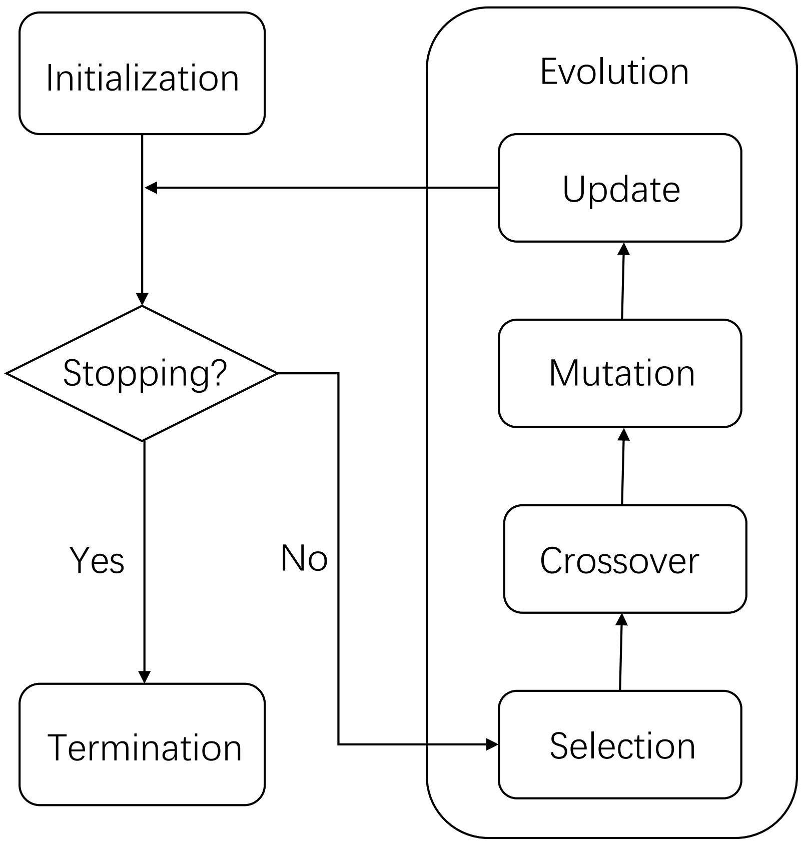


Figure 2.7: Overview of the evolutionary algorithms. [15]

unique genetic coding, symbolic of its structure, type of layers, hyperparameters, and inter-layer connections. The architectures then undergo an evaluation phase. Typically, they're trained on a dataset and their performance is subsequently evaluated on a validation set. This performance, often denoted in accuracy or loss, acts as the 'fitness' score for the architecture.

Based on the available fitness scores, the algorithm chooses specific architectures to act as parents for the next generation. Although high-performing architectures have a greater chance of being selected, a few slots are set aside for the underdogs to maintain diversity. These chosen architectures then experience crossover, which is a mixing of attributes like layers or hyperparameters. To enable the exploration of more novel structures and ensure diversity, random mutations are introduced. This could involve adjusting a layer type or hyperparameters.

After the crossover and mutation process, some or all of the existing architectures may be replaced by the newly formed ones. The best-performing architectures, referred to as the 'elite', are often preserved in order to consistently enhance performance. The algorithm terminates either after reaching a predefined number of generations or when significant performance improvement becomes difficult to achieve.

Notable instances in this field include NEAT [38]. NEAT algorithm begins with simple networks and allows their complexity to evolve gradually, incorporating concepts like speciation and gene history. NEAT evolves not just the topology but also the neural weights. AmoebaNet [31] is another notable method that incorporates asynchronous evolution to enhance search diversity. It has demonstrated outstanding performance on both CIFAR-10 and ImageNet datasets.

The inherent nature of EAs encourages diversity, allowing for the avoidance of local optima and comprehensive exploration of the search space. Moreover, their versatility makes them applicable to various types of neural networks. However, a significant drawback is their computational intensity, especially when training each architecture from scratch. As a result, researchers actively explore efficiency-enhancing techniques like weight sharing or surrogate modeling.

2.5.3 Gradient-Based Optimization

Gradient-based NAS utilizes gradient information from backpropagation [33] to guide and optimize structural decisions within the search space. This approach differs from traditional NAS techniques that often rely on discrete decision-making. The objective of gradient-based NAS is to transform the architecture search problem into a continuous and differentiable context, which allows for the direct application of optimization algorithms like gradient descent. This transformation is typically achieved by softening discrete structural selection parameters into a continuous probability distribution, such as using the softmax function.

Among them, Differentiable Architecture Search (DARTS) [25] is one of the most famous gradient-based NAS methods. In DARTS, each layer of the model has multiple operation options, and the weights of these operations are learnable. Through training, these weights converge, resulting in larger weights for certain operations. Finally, the operation with the highest weight is chosen as the operation for each layer, resulting in the final network structure. This strategy enables simultaneous optimization of both network architecture and network weights using gradient descent. This joint optimization not only simplifies the search process but also significantly enhances its efficiency. More specifically, gradients for architectural parameters are computed by backpropagating through the performance on a validation set, while gradients for network weights are determined based on the performance over the training set.

However, these approaches have certain drawbacks, such as the requirement to store all possible operations, which could significantly increase memory usage. Additionally, similar to other gradient-based optimization techniques, these methods are prone to becoming trapped in local optima. To address these challenges, researchers have introduced enhanced techniques like Fair DARTS [11] and PDARTS [9]. The latter dynamically adjusts the search space during the search process.

Among various NAS strategies, we selected the evolutionary-based NAS approach to train our printed neuromorphic network. The foundational principle of EAs is relatively straightforward, focusing primarily on the fitness function and continuously evolving its population through operations such as mutation and crossover. EAs significantly outperform traditional reinforcement learning approaches across a broad range of benchmark tasks [39]. Compared to gradient-reliant methods, EAs offer enhanced robustness. Furthermore, they possess an almost limitless search space, enabling the exploration of network architectures that are challenging for gradient-based methods, thereby facilitating the identification of innovative structures.

However, evolutionay-based NAS has its limitations, such as substantial computational consumption. But given our target application typically demands smaller network architectures, this limitation can be minimized. In conclusion, for the printed neuromorphic network, the evolutionary-based NAS strategy emerges as an appropriate choice.

In summary, this chapter provides an overview of the foundational knowledge and relevant literature. Our focus is primarily on the printed neuromorphic circuits domain, aiming to achieve a compact design by reducing circuit area. Thus the circuit's topology must be optimized. We reviewed various common NAS methods and, after thorough consideration, chose the EA approach to apply to printed neuromorphic circuit design.

3. NEAT

The previous chapter provided background knowledge and related works for this thesis. The final section compared various common NAS methods. For our application scenario, we chose the EA for compact circuit design because of its unlimited search space and alignment with PE. Among various EA, NEAT inspired us.

This chapter introduces the NEAT algorithm, giving readers an understanding of the NEAT algorithm and laying the groundwork for understanding the method we propose subsequently.

NEAT distinguishes itself from traditional EAs, being specifically designed for the simultaneous evolution of neural network weights and architectures. In contrast to conventional EAs, NEAT integrates several strategies, ensuring that the network structure can flexibly adapt to a diverse range of task requirements while circumventing prevalent pitfalls in the evolution process [37].

Initially, each network is directly connected from input nodes to output nodes without any hidden nodes. The algorithm then selects two 'parent' networks and combines their gene structures to produce a new 'child' network, ensuring compatibility between the two parent networks. The mutation phase follows, involving the random alteration of a connection's weight, the addition of a new connection between two previously unconnected nodes randomly, and the insertion of a new hidden node by splitting an existing connection. To ensure that new, innovative network structures will not be eliminated before they can be truly explored, NEAT employs a strategy known as 'speciation' where networks with similar structures are classified into the same 'species'. This ensures that crossovers occur only between networks of the same species. Each generation selects and replicates the best-performing networks based on their performance in a given task into the next generation.

The advantages of this method include the simultaneous optimization of structure and weights, and starting with simpler structures can help prevent early overfitting. Through speciation, NEAT ensures structural diversity in the population, thus preventing early convergence. However, there are some disadvantages as well, such as the evolutionary process of NEAT requiring multiple evaluations of a large number of networks, leading to high computational costs. Moreover, certain parameters, like the speciation threshold and mutation probabilities, might need adjustments for different tasks. The limitations of these disadvantages can be alleviated since PE is designed to supplement silicon electronics specifically in edge contexts, where circuits are typically of a smaller scale.

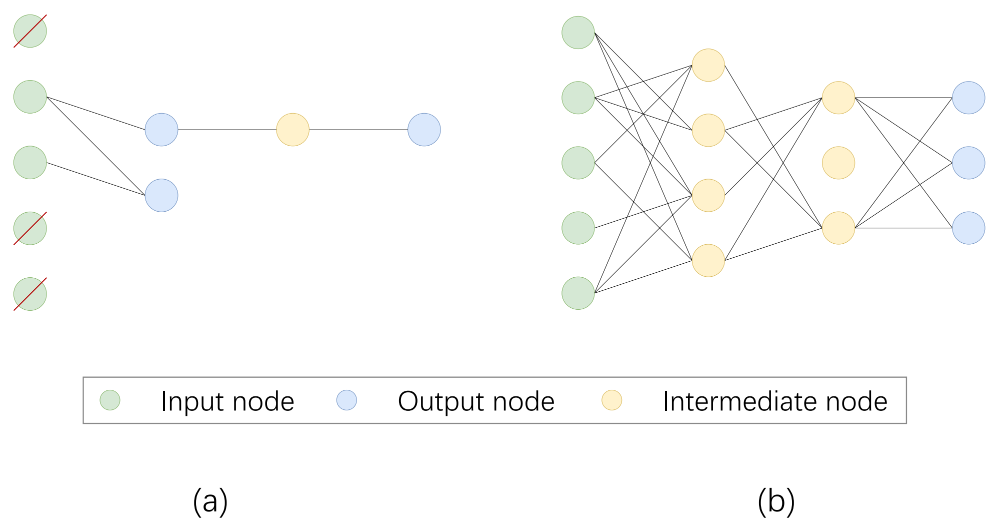


Figure 3.1: Examples of network with comparable accuracy. (a) Result network through NEAT algorithm. (b) Pruned network.

Compared to traditional backpropagation methods, the uniqueness of NEAT is that it doesn't require a pre-defined network structure. This implies that NEAT could potentially outperform in tasks where the optimal network structure is uncertain. Moveover, NEAT demonstrates a significant advantage in discovering novel structures. For instance, Figure 3.1 presents a comparison between the network structures discovered by NEAT and gradient-based pruning approach. The two example networks can achieve equivalent accuracy, but the result network from NEAT algorithm is smaller than pruned network and has a unique topology. Notably, NEAT have the capability to extend the connectivity of the output node to other nodes, thereby functioning as intermediaries as well. Some output node can be seen as intermediate node. This kind of innovative network is almost impossible to be generated from pruning.

In summary, from the perspective of EAs, NEAT offers an efficient and adaptive strategy for the evolution of neural network weights and structures. Its innovative techniques and highlights position it as a preferred solution for many tasks.

4. Methodology

In the compact design of printed neuromorphic circuits, minimizing the area is crucial. A reduced area can directly translate to cost savings, as the fabrication cost of printed circuits often correlates with the space they occupy. Additionally, circuits with smaller areas facilitate higher integration density, which is essential for incorporating complex neuromorphic systems on a single substrate. The reduction in circuit area leads to a more streamlined and compact design. The compactness of these circuits enhances their portability, making them well-suited for wearable and handheld devices, especially in the context of edge computing.

To integrate circuit area directly into the training objective, we initially develop a model in Section 4.1, which quantifies the space occupied by the printed neuromorphic circuits.

Following this, in Section 4.2, we introduce our proposed EA-based approach to simultaneously optimize both the circuit parameters and topology.

Finally, we employ a gradient-based pruning method that may also achieve area optimization and forms the baseline of our proposed method. In Section 4.3, the pruning method is introduced and compared with the EA method.

4.1 Circuit Area Model

This section models the circuit area, laying the foundation for incorporating the area into the objective equation in subsequent steps.

Figure 4.1 illustrates a printed resistor with supplementary layers of conductive ink post-fabrication, adjusting its conductivity accordingly. We can achieve varying resistance values by printing different numbers of ink layers. This implies that, despite differences in resistance values, the occupied area remains roughly consistent. Thus, we directly infer the standard area of a resistor from the scale provided in this figure, namely $A^{\mathrm{R}} = 0.15\mathrm{mm}^2$.

Similarly, we can directly determine the area occupied by nonlinear circuits in printed neuromorphic circuits, such as printed tanh-like non-linear circuit and negative

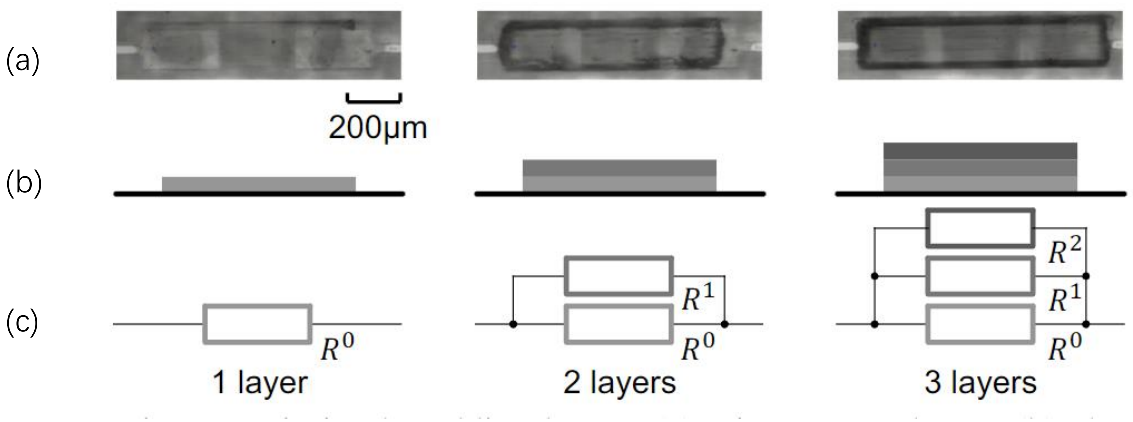


Figure 4.1: Resistor reprinting by adding layers. (a) Microscope photos, (b) Physical schematics, (c) Circuit diagrams [48].

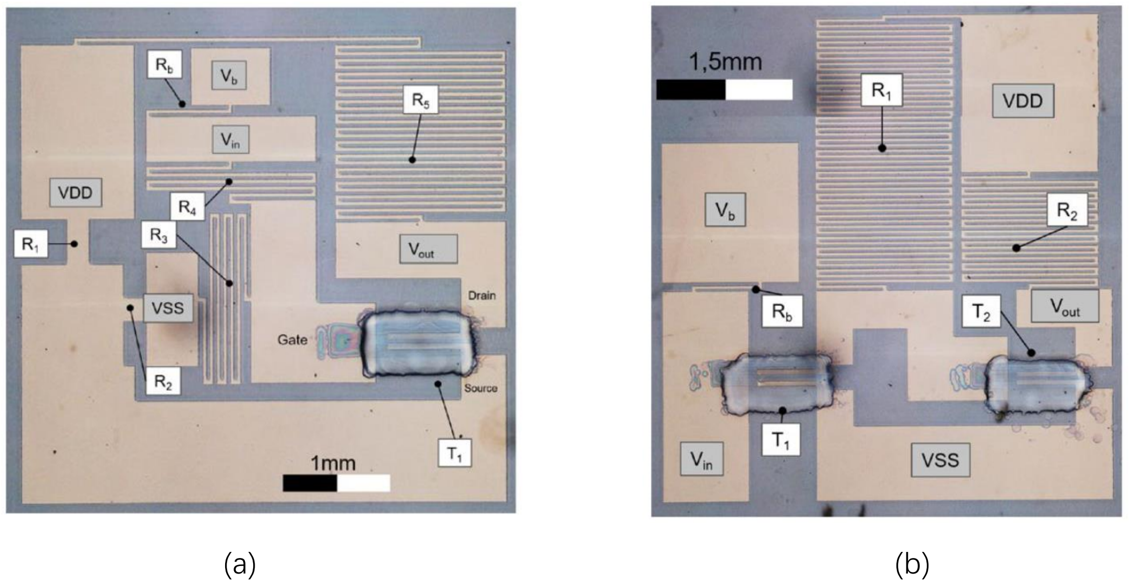


Figure 4.2: Photos of the primitives in printed neuromorphic circuits. (a) Negative weight circuit. (b) Tanh-like non-linear circuit [43].

weight circuit. Given that these circuits have predefined and fixed configurations, their respective areas are directly read from Figure 4.2, with $A^{\mathrm{N}} = 22.7\mathrm{mm}^2$ and $A^{\mathrm{A}} = 30\mathrm{mm}^2$.

Thus, we approximate the total circuit area by

$$A = N^{\mathrm{R}}A^{\mathrm{R}} + N^{\mathrm{N}}A^{\mathrm{N}} + N^{\mathrm{A}}A^{\mathrm{A}}, \tag{4.1}$$

wherein N^{R}, N^{N} and N^{A} represent the counts of resistors, negation circuits, and ptanh circuits, respectively.

In summary, the circuit area model established in this section can be utilized for our compact circuit design.

4.2 Proposed EA-Based Method

Given the application context of the printed neuromorphic circuit and drawing inspiration from the NEAT algorithm, we propose an EA method for compact circuit design, which simultaneously trains both learnable parameters and topology. Below are the key parts of our proposed EA-based method.

4.2.1 Genome

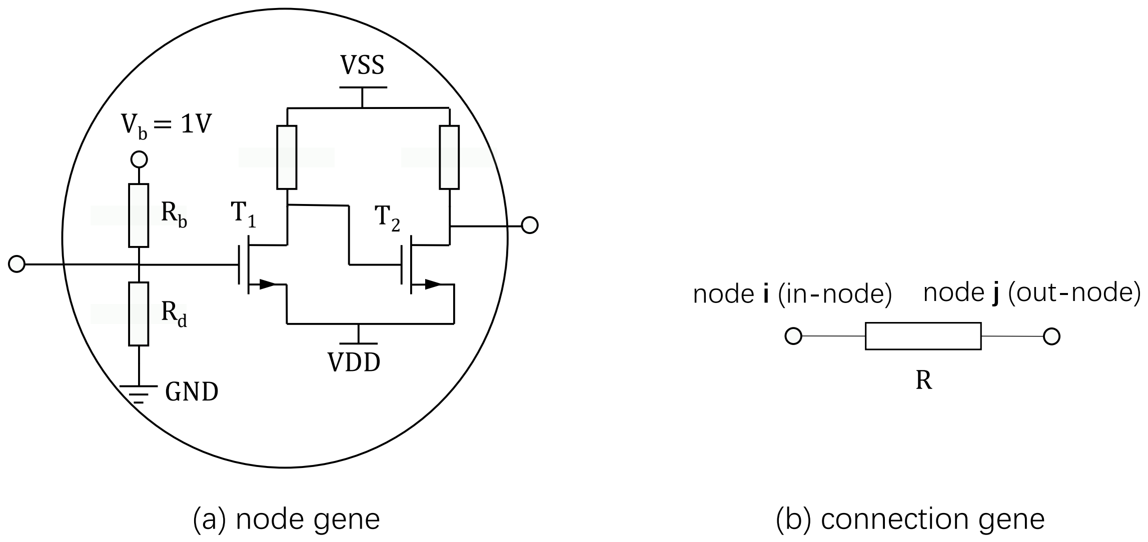


Figure 4.3: The schematic of the genes in printed neuromorphic circuits. (a) Node gene. (b) Connection gene.

In our proposed method, a population of individual genomes is maintained. Each genome contains two sets of genes that describe how to build a pNN, namely node genes and connections genes. The structure of a node gene and a connection gene is illustrated in Figure 4.3.

Node genes, each of which specifies a single printed neuron. Within each node gene, R_{b} and R_{d} from the resistor crossbar (refer to Section 2.2.1) are incorporated as learnable parameters, followed by a consistent printed tanh-like non-linear activation circuit (refer to Section 2.2.2). Every node possesses a distinct global index for explicit identification.

Connection genes, each of which specifies the connection between the in-node and the out-node. A learnable parameter R (refer to the resistance R in Figure 4.3 (b)), corresponding to R_i^{C} in the resistor crossbar, is contained in a connection gene. Additionally, each connection gene is also supplemented by a learnable boolean parameter that signifies whether or not the corresponding connection is expressed in neural network(either enabled or disabled). The identification of connection genes relies on the indices of the nodes they connect. However, the order of nodes is important, e.g., connection gene (i, j) and connection gene (j, i) represent 2 different genes.

Combining various node genes and connection genes, we can construct genomes. Each genome is able to emulate a printed neural network. Figure 4.4 depicts a printed neuromorphic circuit composed of node genes and connection genes.

4.2.2 Speciation

As networks evolve, the addition of nodes or connections can temporarily diminish their performance. How then can we ensure and optimize these novel structures

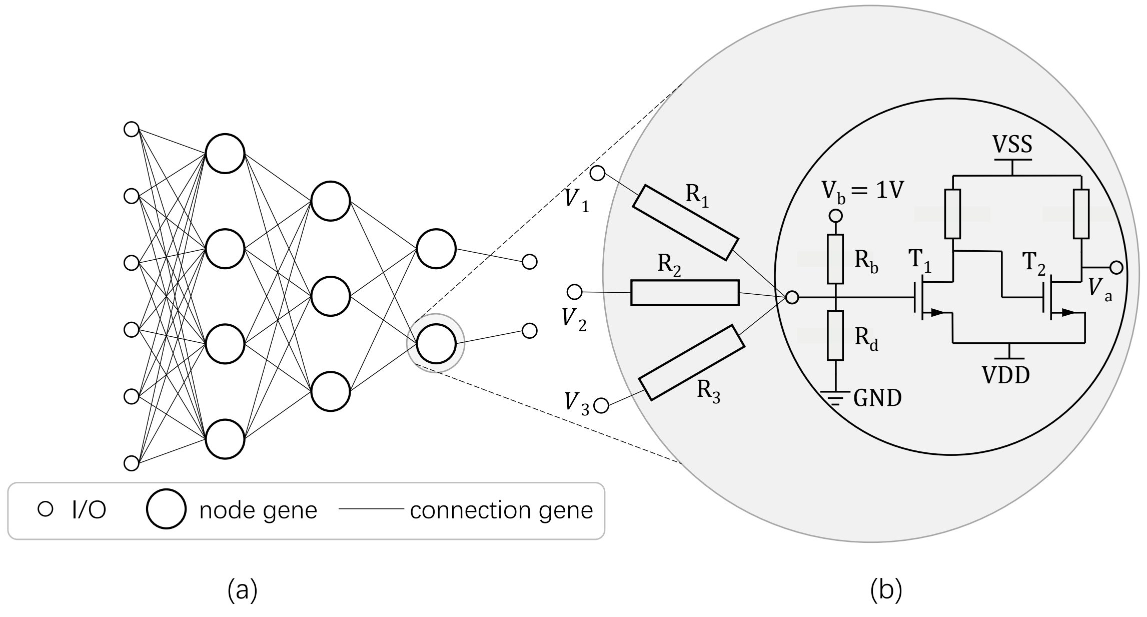


Figure 4.4: The schematic of a printed neuromorphic circuit. (a) Example printed neuromorphic circuit with topology 6-4-3-2. (b) A printed neuron composed of 3 connection genes (left) and 1 node gene (right).

before they're prematurely culled from the population? NEAT solves this dilemma through speciation. Similarly, in our approach, we categorize the population based on the similarity of their genomes, where each genome competes only within its designated species rather than against the entire population.

During the species partitioning process, the algorithm first identifies the best representative for each existing species and removes these representatives from the pool of unclassified genomes. Subsequently, for each remaining unclassified genome, the algorithm calculates its genomic distance to each of the representatives to determine its species assignment. This genomic distance is computed based on the similarity in connection genes, node genes within the genomes.

The nodes distance is expressed as

$$D^{\mathrm{N}} = \frac{\sum D_{ij}^{\mathrm{N}} + (c \cdot N_{\mathrm{disjoint\_nodes}})}{N_{\mathrm{max\_nodes}}},$$

and the connections distance is expressed as

$$D^{\mathrm{C}} = \frac{\sum D_{ij}^{\mathrm{C}} + (c \cdot N_{\mathrm{disjoint\_connections}})}{N_{\mathrm{max\_conn}}},$$

thus, the distance between 2 genomes can be determined by

$$D = D^{\mathrm{N}} + D^{\mathrm{N}},$$

where c denotes compatibility disjoint coefficient (explained later in Section 5.1.4), $N_{\mathrm{disjoint\_nodes}}$ and $N_{\mathrm{disjoint\_connections}}$ denotes the number of disjoint genes between 2 genomes, $N_{\mathrm{max\_nodes}}$ and $N_{\mathrm{max\_conn}}$ represents the number of genes in the genome that has a greater number of genes among the two genomes. D_{ij}^{N} signifies the distance between homologous node genes (genes have same identification) as

$$D_{ij}^{\mathrm{N}} = (|R_{\mathrm{b}_i} - R_{\mathrm{b}_j}| + |R_{\mathrm{d}_i} - R_{\mathrm{d}_j}|) \cdot k,$$

where k symbolizes the compatibility weight coefficient (explained later in Section 5.1.4). Similarly, D_{ij}^{C} signifies the distance between homologous connection genes, determined by

$$D_{ij}^{\mathrm{C}} = |R_i - R_j| + \mathbb{1}_{\{e_i \neq e_j\}} \cdot k,$$

where e_i and e_j means the connection states of these genes(enable or disable).

If a genome is not sufficiently similar to any existing representative (i.e., the distance exceeds a predefined compatibility threshold), it becomes the representative of a newly formed species. Finally, the algorithm updates the species collection accordingly.

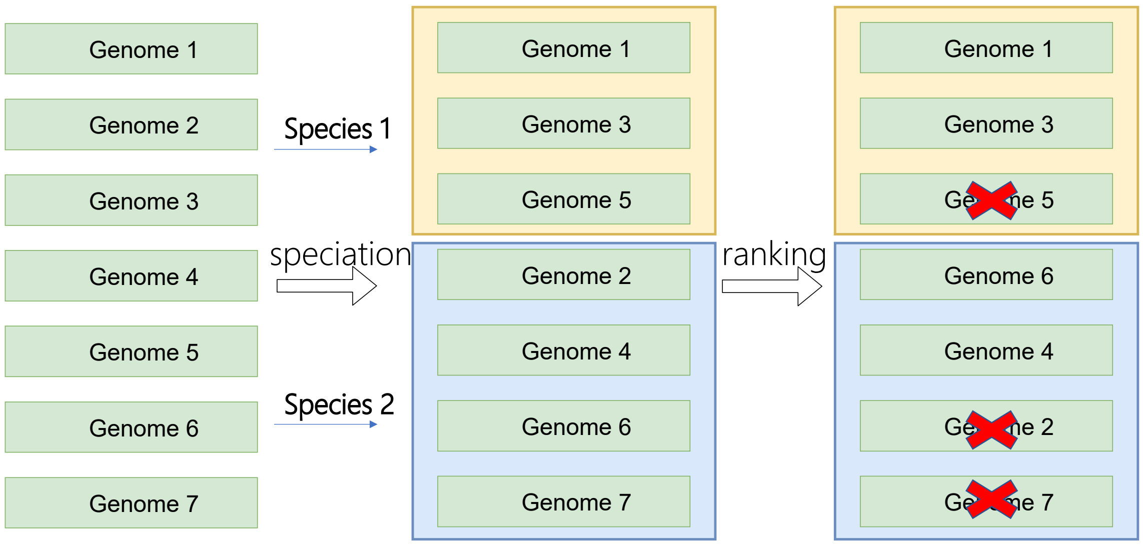


Figure 4.5: Process of selection.

As shown in Figure 4.5, these genomes are divided into 2 species. And then sort the genomes in each species according to their fitness value. The fitness value evaluate the performance of each genome. The best-performing $M\%$ of each species is randomly selected for crossover (at least 2), where M is a hyperparameter named survival threshold.

4.2.3 Crossover

Crossover serves as a strategy to combine attributes from two parent individuals to produce new offspring. In the proposed method,the crossover process takes into special consideration of the structural differences between neural networks, allowing even those with different structures to be crossed. This approach is particularly important for neural networks in our method, as their architecture, such as nodes and connections, can change over the course of evolution.

After the process of speciation, the top M% genomes, ranked based on their fitness, from each species are selected as parents for crossover. Among these genomes, two

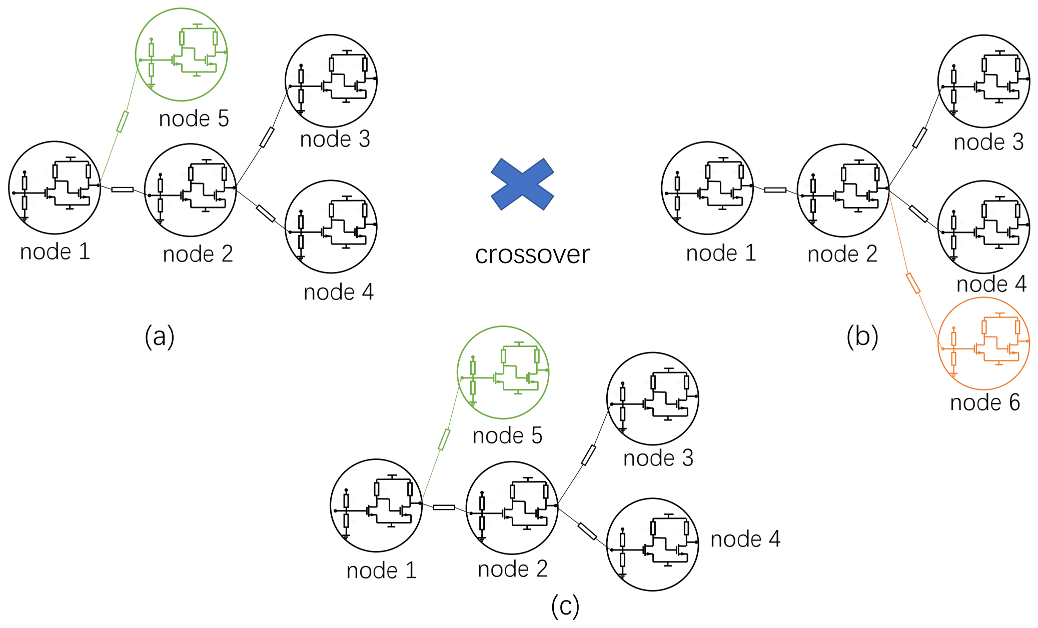


Figure 4.6: Example for crossover. (a) Parent genome 1. (b) Parent genome 2. (c) Offspring.

genomes are randomly selected for crossover. The one with higher fitness is referred to as the parent genome 1, and the other as parent genome 2. An example for crossover is shown in Figure 4.6. The process iterates through all the genes in parent genome 1. For each gene in genome 1, it checks whether a matching gene exists in genome 2. If no match is found, this gene is considered to be inconsistent and is directly copied from the parent with higher fitness, namely parent 1. However, if both parents possess the gene, it means they are homologous genes; the offspring will then randomly inherit from one of these parents. As for the genes present only in parent 2, they are not inherited in our approach.

In summary, the core objective of crossover is to inherit structural and parametric attributes from both parent genomes, ensuring that genome with higher fitness have a greater influence on the resulting offspring.

4.2.4 Mutation

Our proposed approach aims to optimize both the topology and weights of printed neural networks. In this framework, mutations play an essential role by injecting necessary diversity into the evolution of the neural network, thus enhancing the potential to find optimal solutions. We broadly categorize mutations in our method into two types: parameter mutations and topological mutations.

For parameter mutations, they primarily target the resistance values in node genes and connection genes, corresponding to R_b and R_d in Figure 4.3 (a) and R in (b). These resistance values undergo random modifications, which might be subtle adjustments or replacement by a new value from a random distribution.

On the other hand, topological mutations can be further divided into the following sub-types:

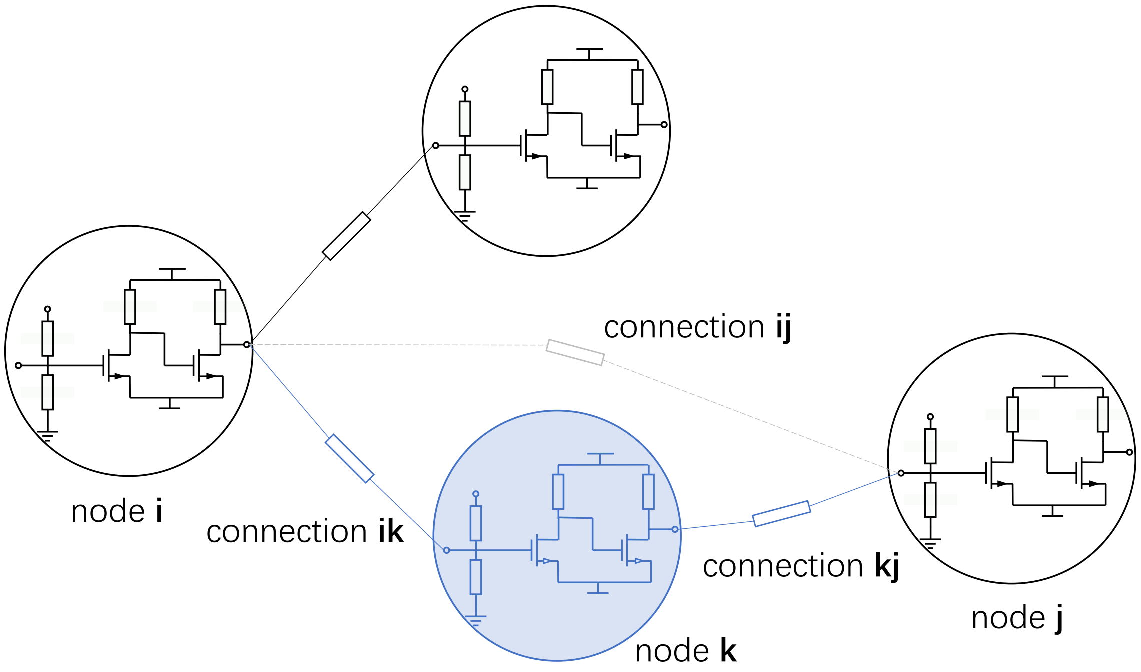


Figure 4.7: Example for mutation of adding node.

Mutate add node

In this 'add node' mutation, as shown in Figure 4.7, an existing connection is split, with the new node being placed where the old connection once was. It's worth noting that, the original connection, refer to connection ij, is set to a disable state(rather than delete). The original starting node is linked to the new node with a resistance of 1 (refer to connection ik), and the new node is then connected to the original ending node, retaining the resistance of the old connection ($R_{kj} = R_{ij}$). This operation contributes to the increasing complexity of the network topology.

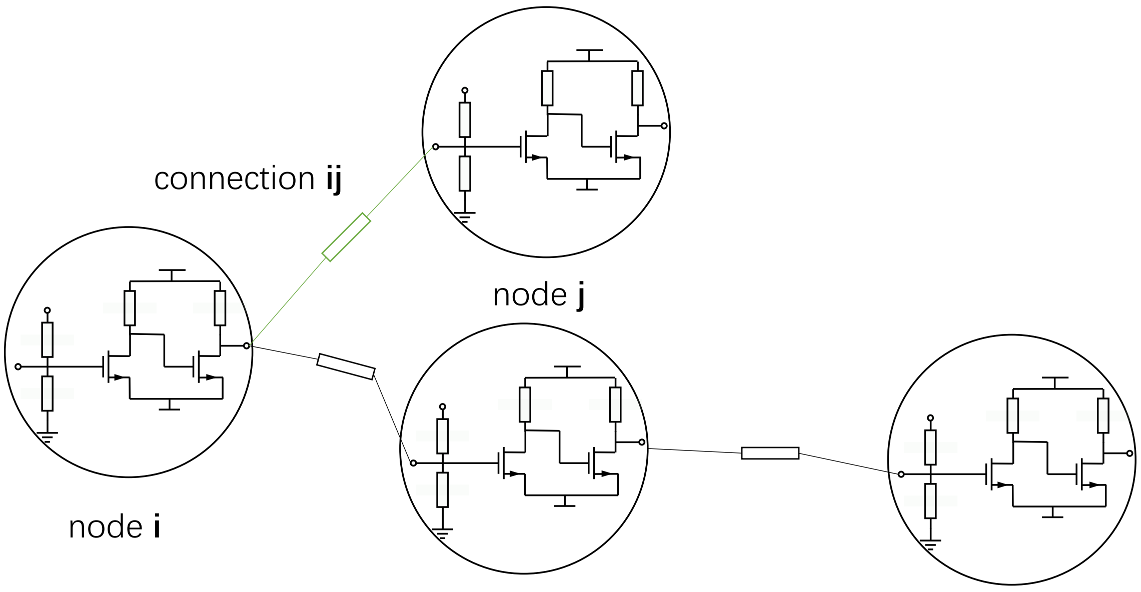


Figure 4.8: Example for mutation of adding connection.

Mutate add connection

In this 'add connection' mutation, two nodes are randomly selected. If there is no existing connection between them, a new one is established, as shown in Figure 4.8. However, if a connection previously existed but is currently disabled, it gets re-enabled.

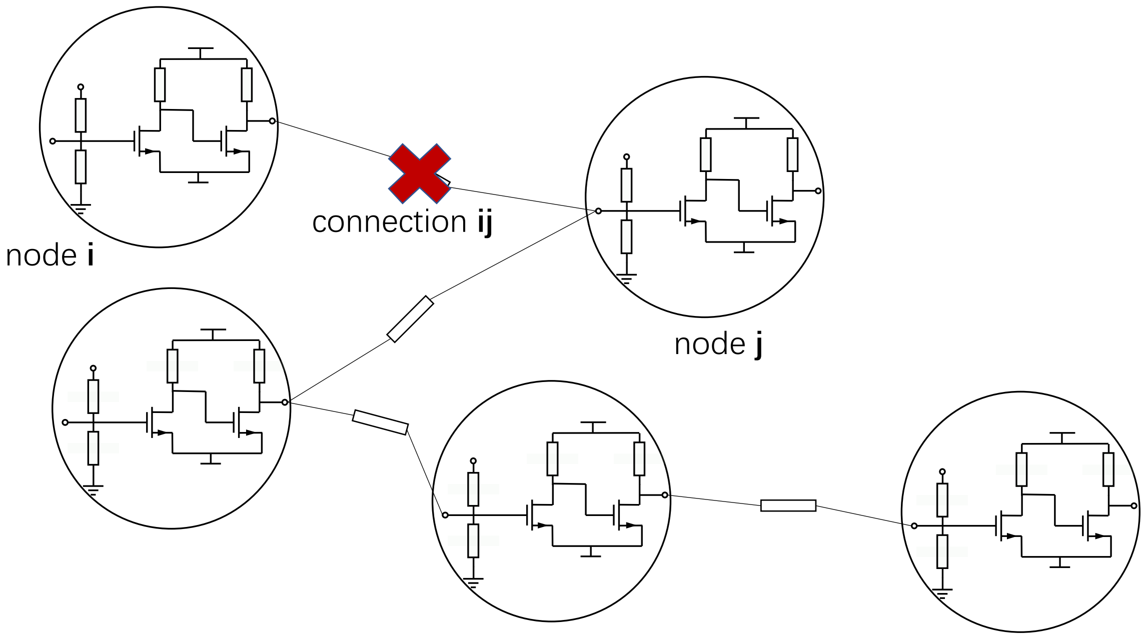


Figure 4.9: Example for mutation of removing connection.

Mutate remove connection

In this 'remove connection' mutation, as shown in Figure 4.9, a connection is randomly chosen and then deleted.

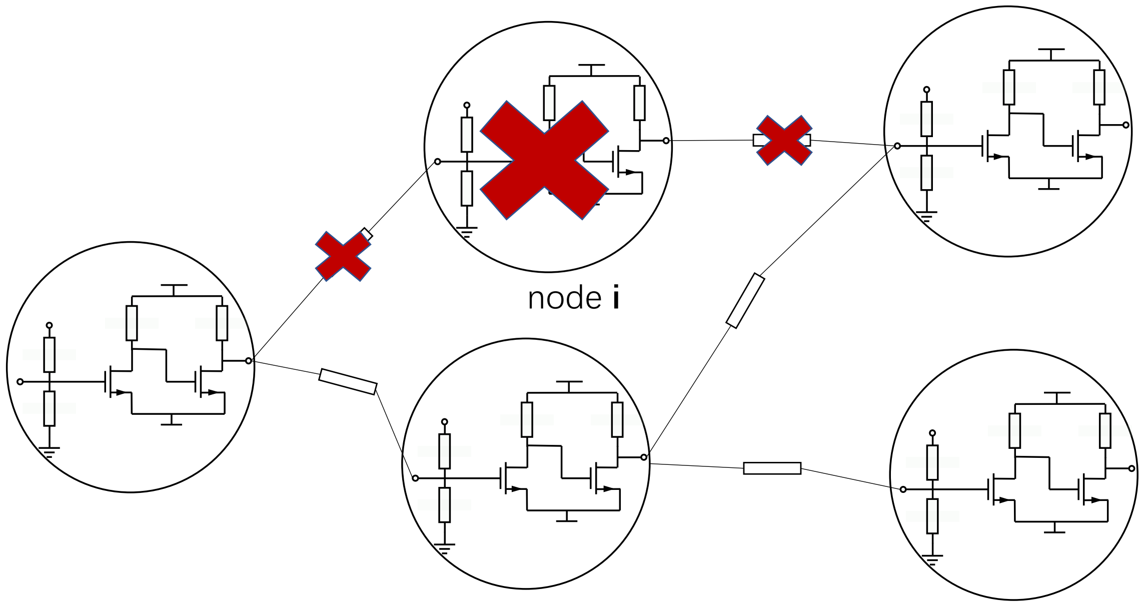


Figure 4.10: Example for mutation of removing node.

Mutate remove node

In this 'remove node' mutation, as shown in Figure 4.10, a node is randomly selected, and it, along with its associated connection genes, is entirely removed. It's worth noting that the output nodes cannot be removed; thus, they won't be selected for deletion.

Through iterative cycles of the aforementioned generational processes, the fitness of the genomes undergoes progressive optimization. When the predetermined termination criterion is met, indicating that the genomes have reached a satisfactory level of optimization, the corresponding topological structures and parameters can be transposed into the required printed neuromorphic circuit for further fabrication.

4.3 Comparison to Gradient-Based Pruning

While the existing gradient-based training for pNCs can not directly address topological problems such as circuit area, we still adopt it as the baseline of our proposed method by incorporating strategies as pruning (removing parameters below a certain threshold) and straight through estimator [6]. Therefore, this section presents some details of the EA-based method, as well as a comparison with gradient-based training for compact printed neuromorphic circuits in terms of initialization and area estimation.

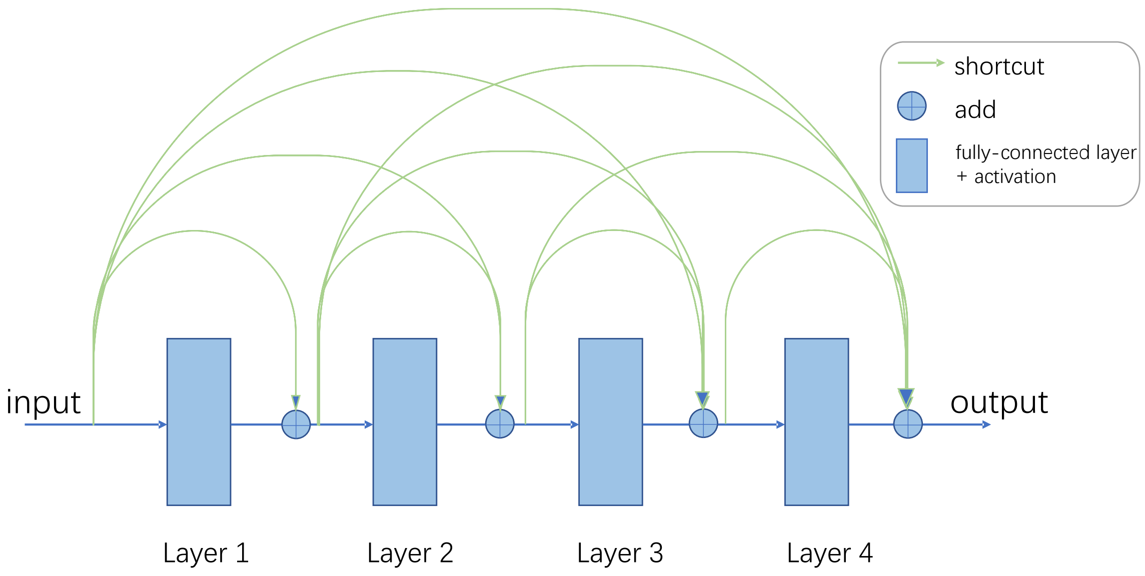


Figure 4.11: Schematic of the fully connected dense network with all possible shortcuts. Pruning this network serves as our baseline.

4.3.1 Initialization

From a topological perspective, as gradient-based pruning forms a dense-to-sparse approach, circuit topology is initialized as a multi-layered, fully-connected network with all possible shortcuts to guarantee a sufficient good result. The schematic of an example baseline network is illustrated in Figure 4.11. Each layer represent a fully connected layer followed by an activation. The green curves indicate shortcuts that only have weighted-sum, with no activation. The network comprises a total of C_m^2 shortcut connections, where m represents the number of network layers, including the input and output layers.

In contrast, the EA approach progresses from a sparse to a dense structure by progressively expand the circuit size, therefore, it starts with a topology with only output nodes without any connections.

From a parametric perspective, initialization of learnable parameters can greatly influence the success of learning [14]. For printed neuromorphic circuits, the physical properties of the printed materials introduce additional constraints. For example, the range of feasible conductance values $g_i \in [g_{\text{min}}, g_{\text{max}}]$. A poor choice in initialization could lead the circuit to operate outside its optimal or reliable range.

In this regard, we can initialize the conductance g_i uniformly around g_{min} with a random offset up to 0.01. Additionally, g_{d} should be initially set to the highest possible value in order to allow for maximum decoupling, namely $g_{\text{d}} = g_{\text{max}}$.

However, in gradient-based pruning network, vanishing gradients [16] can occur because of the combination of initialization and activation function. To address this problem, g_{b} should be initialized by

$$g_{\text{b}} = \frac{\eta_3^A}{1-\eta_3^A}(\sum_i g_i + g_{\text{d}}).$$

Since in EA-based method, g_{b} , g_i and g_{d} are initialized simultaneously, g_{b} cannot be initialized as shown above. Therefore, we initialized g_{b} in the same way as g_i.

4.3.2 Area estimation

Achieving area optimization through the minimization of Equation 4.1 within the EA framework is relatively straightforward, as it simply involves evaluating resistances and totaling number of node genes and connection genes. In contrast, accomplishing the same task using gradient-based methods presents a unique set of challenges, primarily owing to the discrete nature of device counts.

As illustrated in Figure 3.1 (b), when all inputs to a neuron are pruned, the neuron itself, along with its associated g_b, g_d, and ptanh circuits, can also be eliminated. Specifically, N^{A} is expressed by

$$\max_i \left\{[\mathbb{1}_{\{g_1>0\}}, \mathbb{1}_{\{g_2>0\}}, ..., \mathbb{1}_{\{g_i>0\}}, ...]\right\}. \tag{4.2}$$

However, within the context of gradient-based optimization, the indicator function acts as a step function, yielding gradients of either 0 or ∞ for $\frac{\partial N^{\text{A}}}{\partial g_i}$. g_i will not be modified for the purpose of reducing N^{A}. To address this issue and enable the optimization of N^{A} through g_i, we introduce the soft count of ptanh circuits, denoted by $N^{\text{A}}_{\text{soft}}$. In the forward pass of the soft count, $N^{\text{A}}_{\text{soft}}$ is still calculated by Equation 4.2, however, in the backpropagation, a relaxed function,

$$\max_i \left\{[\text{sigmoid}(g_1), \text{sigmoid}(g_2), ..., \text{sigmoid}(g_i), ...]\right\},$$

is employed to generate the gradient for updating g_i. Compared to Equation 4.2, the indicator function is relaxed as a sigmoid function. This kind of separate treatment for the forward and backward pass is also referred to as the straight-through gradient estimator [6].

In a similar manner, the counts of resistors and negative weight circuits can also be represented using the indicator function $\mathbb{1}_{\{\cdot\}}$ and softened by the sigmoid$(\cdot)$ function. $N^{\mathrm{R}}_{\mathrm{soft}}$, in the forward pass, is expressed by

$$\sum_i \left\{ [\mathbb{1}_{\{g_1>0\}}, \mathbb{1}_{\{g_2>0\}}, ..., \mathbb{1}_{\{g_i>0\}}, ..., \mathbb{1}_{\{g_b>0\}}, \mathbb{1}_{\{g_d>0\}}] \right\},$$

in the backpropagation, is calculated by

$$\sum_i \{[\mathrm{sigmoid}(g_1), \mathrm{sigmoid}(g_2), ..., \mathrm{sigmoid}(g_i), ..., \mathrm{sigmoid}(g_d), \mathrm{sigmoid}(g_b)]\}.$$

$N^{\mathrm{N}}_{\mathrm{soft}}$, in the forward pass, is expressed by

$$\underset{i}{max} \left\{ [1 - \mathbb{1}_{\{\theta_1 \geq 0\}}, 1 - \mathbb{1}_{\{\theta_2 \geq 0\}}, ..., 1 - \mathbb{1}_{\{\theta_i \geq 0\}}, ...] \right\},$$

in the backpropagation, is calculated by

$$\underset{i}{max} \{[1 - \mathrm{sigmoid}(\theta_1), 1 - \mathrm{sigmoid}(\theta_2), ..., 1 - \mathrm{sigmoid}(\theta_i), ...]\},$$

As a result, the area estimator for gradient-based methods is formulated as follows:

$$A_{\mathrm{soft}} = N^{\mathrm{R}}_{\mathrm{soft}} A^{\mathrm{R}} + N^{\mathrm{N}}_{\mathrm{soft}} A^{\mathrm{N}} + N^{\mathrm{A}}_{\mathrm{soft}} A^{\mathrm{A}}. \tag{4.3}$$

In this chapter, we first established an area model for circuits. Subsequently, we delved into the key components of our proposed method. Towards the end of the chapter, we compared our approach with the gradient-based pruning network, which served as the baseline of this work.

5. Experiment

The preceding chapter offered a description of the method we have proposed. To assess the effectiveness of our proposed EA-based approach, we implemented the algorithm using PyTorch [28] and neat-python[1] module and then conducted experiments on 11 benchmark datasets.

5.1 Experiment Setup

5.1.1 Dataset

Dataset	**#Input**	**#Output**	**#Data**
Acute Inflammation	6	2	120
Balance Scale	4	3	625
Breast Cancer Wisconsin	9	2	699
Energy (y1)	8	3	768
Energy (y2)	8	3	768
Iris	4	3	150
Mammographic Mass	5	2	961
Seeds	7	3	210
Tic-Tac-Toe Endgame	9	2	958
Vertebral Column (2 cl.)	6	2	310
Vertebral Column (3 cl.)	6	3	310

Table 5.1: Information of 11 benchmark datasets.

In this work, we conducted experiments on a set of 11 benchmark datasets. These datasets have been utilized in other state-of-the-art studies related to printed neuromorphic circuits [43, 45]. Moreover, these benchmark datasets exhibit complexity and scenarios that align with the intended application domains of PE. We randomly split each dataset into training set (60%), validation set (20%), and test set(20%) . The detailed information about the datasets can be found in Table 5.1. For these datasets, the inputs consist of features, while the outputs are represented by labels encoded as 0, 1, 2, and so on. Each of these datasets pertains to classification tasks.

[1]The module is available at https://neat-python.readthedocs.io

5.1.2 Initial topology

In our proposed method, the topologies for all datasets are initialized as unconnected networks, including only #output noodes. Additionally, in our approach, both the activation circuits and the negative circuits remain constant and unaltered. The corresponding auxiliary parameters are $\boldsymbol{\eta}^{\mathrm{A}} = [0.290, 0.710, -0.017, 20]$ and $\boldsymbol{\eta}^{\mathrm{N}} = [-0.006, 1.024, 0.016, 1.006]$, respectively.

As baseline, the pruning networks for all datasets use a consistent topology(#inputs-3-4-#outputs). Apart from the input and output layers, there are 2 hidden layer containing 4 nodes and 3 nodes.

5.1.3 Objective function

For training ANNs, loss functions are generally utilized to guide the optimization process and reflect the performance of the ANNs. For classification tasks we adopt a typical loss function, namely cross-entropy(CE), as metric for classification accuracy. It measure the difference between the model's predicted probability distribution and the actual distribution of the labels. It heavily penalizes confident yet incorrect predictions, making it effective in ensuring that models are not just accurate, but also confident in their predictions. Being differentiable, it's also suitable for gradient-based optimization. Additionally, it often leads to quicker training convergence compared to other loss functions.

Therefore, to jointly optimize both classification accuracy and circuit area, the training objective is given by

$$\mathcal{L}(\boldsymbol{\theta}) = (1-\gamma) \cdot \mathbf{CE}(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y}) + \gamma \cdot \frac{A}{A'}, \tag{5.1}$$

where $\boldsymbol{\theta}$ collects all learnable conductance introduced in Equation 2.3, γ denotes the trade-off factor between accuracy and circuit area, vector $\mathbf{x}$ and vector $\mathbf{y}$ are training examples provided by the target dataset, and A' serves as a constant multiplier to harmonize the magnitude of the area term with that of the accuracy term. Through few initial trials, for each dataset A' is empirically set to different constant value, is shown in Table 5.2. Moreover, To explore the trade-off between accuracy and area, we methodically choose 50 values for γ within the range $[0, 1]$. Each value of γ undergoes training 10 times, with varying random seeds from 1 to 10, ensuring a robust and optimal solution for the stochastic process.

Additionally, we employed an early stopping strategy [30] with a patience of 30, meaning the training will halt if there's no performance improvement on the validation set for 30 consecutive epochs.

Analogously, the objective function for the baseline is

$$\mathcal{L}_{\mathrm{b}}(\boldsymbol{\theta}) = (1-\gamma) \cdot \mathbf{CE}(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y}) + \gamma \cdot \frac{A_{\mathrm{soft}}}{A'},$$

where A_{soft} is obtained by Equation 4.3, since it is a gradient-based training.

Dataset	$\boldsymbol{A'}(\text{mm}^2)$
Acute Inflammation	10840
Balance Scale	10360
Breast Cancer Wisconsin	12860
Energy (y1)	6250
Energy (y2)	6190
Iris	10660
Mammographic Mass	2192
Seeds	6090
Tic-Tac-Toe Endgame	61400
Vertebral Column (2 cl.)	2176
Vertebral Column (3 cl.)	28850

Table 5.2: Constant multiplier A' for each dataset.

5.1.4 Hyperparameter

For the baseline, we choose the optimizer Adam [20] with default parameters. The learning rate is set to 0.01.

In the EA-based method, the hyperparameters delineate how the topology evolves. Table 5.3 provides a detailed list of the specific hyperparameters we employed in our experiments.

Hyperparameter	**Value**
pop_size	300
mutate_rate	0.7
replace_rate	0.1
compatibility_disjoint_coefficient	1
compatibility_weight_coefficient	0.5
conn_add_prob	0.6
conn_delete_prob	0.4
enabled_default	True
enabled_mutate_rate	0.01
node_add_prob	0.3
node_delete_prob	0.2
compatibility_threshold	3
max_stagnation	20
species_elitism	2
elitism	2
survival_threshold	0.2

Table 5.3: Hyperparameters for EA-based method.

Below is an explanation for each hyperparameter.

Pop Size: The number of individuals in each generation.

Mutate Rate: The probability that mutation will change the learnable parameters of a gene by adding a random value.

Replace Rate: The probability that mutation will replace the learnable parameters of a gene with a newly chosen random value (as if it were a new gene).

Compatibility Disjoint Coefficient: The coefficient for the disjoint and excess gene counts' contribution to the genomic distance.

Compatibility Weight Coefficient: The coefficient for each learnable parameter difference's contribution to the genomic distance (for homologous nodes or connections).

Conn Add Prob: The probability that mutation will add a connection between existing nodes.

Conn Delete Prob: The probability that mutation will delete an existing connection.

Enabled Default: The default enabled attribute of newly created connections.

Enabled Mutate Rate: The probability that mutation will replace (50/50 chance of True or False) the enabled status of a connection.

Node Add Prob: The probability that mutation will add a new node (essentially replacing an existing connection, the enabled status of which will be set to False).

Node Delete Prob: The probability that mutation will delete an existing node (and all connections to it).

Compatibility Threshold: Individuals whose genomic distance is less than this threshold are considered to be in the same species.

Species Elitism: The number of species that will be protected from stagnation; mainly intended to prevent total extinctions caused by all species becoming stagnant before new species arise. For example, a 'species elitism' setting of 2 will prevent the 2 species with the highest species fitness from being removed for stagnation regardless of the amount of time they have not shown improvement.

Elitism: The number of most-fit individuals in each species that will be preserved as-is from one generation to the next.

Survival Threshold: The fraction for each species allowed to reproduce each generation.

In summary, this chapter systematically details the experimental setups undertaken to validate the proposed method. It provides a comprehensive overview of the datasets, initial topology, objective functions, and hyperparameters, ensuring a transparent and rigorous account of the entire validation process.

5.2 Result

After training, we evaluate the trained printed neuromorphic networks on the test sets. Table 5.4 and 5.5 report the accuracies and areas with $\gamma \in \{0, 0.25, 0.5, 0.75, 1\}$. The former results are derived from pruning, while the latter are obtained through the EA approach.

Dataset	$\gamma = 0$		$\gamma = 0.25$		$\gamma = 0.5$		$\gamma = 0.75$		$\gamma = 1$	
	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)
Acute Inflammat	1.000 ± 0.000	583.7 ± 7.1	1.000 ± 0.000	354.9 ± 51.4	1.000 ± 0.000	305.4 ± 86.7	0.920 ± 0.214	187.1 ± 62.6	0.320 ± 0.000	60.6 ± 0.0
Balance Scale	0.907 ± 0.015	559.7 ± 11.7	0.878 ± 0.039	350.8 ± 56.9	0.706 ± 0.208	207.6 ± 122.3	0.550 ± 0.172	125.3 ± 72.5	0.111 ± 0.000	90.9 ± 0.0
Breast Cancer Wiscon	0.966 ± 0.006	550.5 ± 72.4	0.970 ± 0.003	193.4 ± 31.6	0.971 ± 0.002	135.0 ± 7.1	0.959 ± 0.005	60.6 ± 0.0	0.693 ± 0.000	60.6 ± 0.0
Energy Efficiency (y1)	0.902 ± 0.030	650.8 ± 24.8	0.877 ± 0.029	327.1 ± 21.2	0.858 ± 0.022	297.0 ± 19.1	0.773 ± 0.095	199.7 ± 62.8	0.500 ± 0.000	90.9 ± 0.0
Energy Efficiency (y2)	0.910 ± 0.011	654.1 ± 16.0	0.858 ± 0.028	379.8 ± 38.1	0.827 ± 0.056	331.0 ± 47.5	0.612 ± 0.101	114.3 ± 33.1	0.526 ± 0.000	90.9 ± 0.0
Iris	0.942 ± 0.025	555.1 ± 14.6	0.968 ± 0.000	311.3 ± 34.0	0.968 ± 0.000	260.8 ± 25.9	0.771 ± 0.113	141.1 ± 51.2	0.323 ± 0.000	90.9 ± 0.0
Mammographic Mass	0.828 ± 0.011	543.0 ± 9.6	0.798 ± 0.012	255.8 ± 49.7	0.787 ± 0.007	127.0 ± 23.2	0.705 ± 0.089	85.4 ± 27.5	0.539 ± 0.000	60.6 ± 0.0
Seeds	0.912 ± 0.029	635.3 ± 12.4	0.874 ± 0.052	318.0 ± 75.1	0.867 ± 0.027	208.6 ± 23.6	0.812 ± 0.141	119.6 ± 23.6	0.256 ± 0.000	90.9 ± 0.0
Tic-Tac-Toe Endgame	0.990 ± 0.004	591.6 ± 26.3	0.996 ± 0.004	212.9 ± 26.2	1.000 ± 0.000	182.2 ± 9.9	0.962 ± 0.117	114.7 ± 47.7	0.370 ± 0.000	60.6 ± 0.0
Vertebral Column (2 cl.)	0.852 ± 0.015	558.8 ± 41.1	0.840 ± 0.017	343.5 ± 24.1	0.635 ± 0.000	60.6 ± 0.0	0.635 ± 0.000	60.6 ± 0.0	0.635 ± 0.000	60.6 ± 0.0
Vertebral Column (3 cl.)	0.822 ± 0.020	573.1 ± 28.5	0.821 ± 0.026	357.0 ± 43.3	0.757 ± 0.049	223.9 ± 67.1	0.570 ± 0.070	91.8 ± 2.7	0.159 ± 0.000	90.9 ± 0.0
Average	0.912 ± 0.015	586.9 ± 24.0	0.898 ± 0.019	309.5 ± 41.1	0.852 ± 0.034	212.6 ± 39.3	0.752 ± 0.102	118.2 ± 34.9	0.403 ± 0.000	77.1 ± 0.0

Table 5.4: Result of the baseline-experiment on 11 benchmark datasets.

Dataset	$\gamma = 0$		$\gamma = 0.25$		$\gamma = 0.5$		$\gamma = 0.75$		$\gamma = 1$	
	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)	Accuracy	Area(mm^2)
Acute Inflammat	1 ± 0	200.3 ± 73.3	1 ± 0	109.2 ± 10.8	1 ± 0	97.1 ± 14.3	1 ± 0	96.3 ± 14.8	0.32 ± 0	60.6 ± 0
Balance Scale	0.877 ± 0.017	233.2 ± 67.2	0.884 ± 0.004	159.1 ± 38.4	0.883 ± 0.003	136.3 ± 14	0.886 ± 0.004	121.1 ± 12.1	0.111 ± 0	90.9 ± 0
Breast Cancer Wiscon	0.962 ± 0.006	142.7 ± 64.7	0.958 ± 0.004	105.9 ± 24	0.958 ± 0.004	96.6 ± 20.1	0.957 ± 0.005	90.4 ± 12.2	0.693 ± 0	60.6 ± 0
Energy Efficiency (y1)	0.86 ± 0.033	183.6 ± 71	0.855 ± 0.018	127.7 ± 18.5	0.858 ± 0.027	120.8 ± 12.2	0.844 ± 0.011	114.5 ± 0.3	0.5 ± 0	90.9 ± 0
Energy Efficiency (y2)	0.877 ± 0	163.4 ± 49.7	0.877 ± 0	129.9 ± 15.2	0.877 ± 0.017	114.5 ± 0.2	0.877 ± 0	114.4 ± 0.2	0.526 ± 0	90.9 ± 0
Iris	0.887 ± 0.105	196.9 ± 71	0.813 ± 0.137	203 ± 55.7	0.794 ± 0.131	138.8 ± 22.8	0.726 ± 0.123	120.3 ± 12.4	0.323 ± 0	90.9 ± 0
Mammographic Mass	0.792 ± 0.005	160.5 ± 68.7	0.797 ± 0.002	83.8 ± 0.1	0.793 ± 0.007	83.8 ± 0.1	0.798 ± 0.003	83.7 ± 0.1	0.539 ± 0	60.6 ± 0
Seeds	0.87 ± 0.021	169.7 ± 32.9	0.851 ± 0.032	139.3 ± 47	0.877 ± 0.028	129.9 ± 28.2	0.853 ± 0.031	114.4 ± 0.2	0.256 ± 0	90.9 ± 0
Tic-Tac-Toe Endgame	0.716 ± 0.02	96.5 ± 20.3	0.717 ± 0.017	118.1 ± 42.1	0.726 ± 0.02	105.9 ± 23.8	0.716 ± 0.019	96.7 ± 20.3	0.37 ± 0	60.6 ± 0
Vertebral Column (2 cl.)	0.806 ± 0.012	160.4 ± 34.2	0.802 ± 0.023	111.4 ± 16.5	0.689 ± 0.066	105 ± 13.9	0.757 ± 0.045	83.7 ± 0.1	0.635 ± 0	60.6 ± 0
Vertebral Column (3 cl.)	0.781 ± 0.04	184.8 ± 23.7	0.781 ± 0.016	141.7 ± 9.1	0.783 ± 0.007	147.6 ± 16.5	0.727 ± 0.066	117.2 ± 9	0.159 ± 0	90.9 ± 0
Average	0.857 ± 0.024	172.0 ± 52.4	0.849 ± 0.023	129.9 ± 25.2	0.840 ± 0.028	116.0 ± 15.1	0.831 ± 0.028	104.8 ± 7.4	0.403 ± 0.000	77.1 ± 0.0

Table 5.5: Result of the EA-experiment on 11 benchmark datasets.

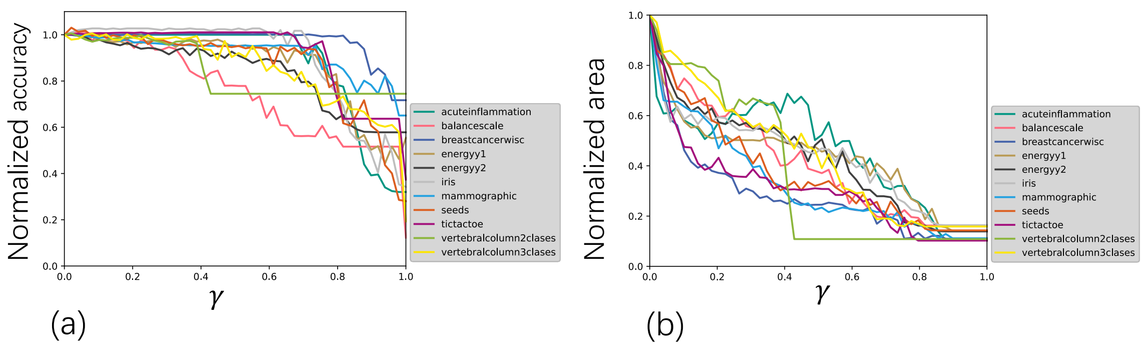


Figure 5.1: Results of Pruning with different γ values. (a) Normalized accuracies of 11 tasks. Each task indicated by a different color. (b) Normalized area of 11 circuits for the corresponding tasks.

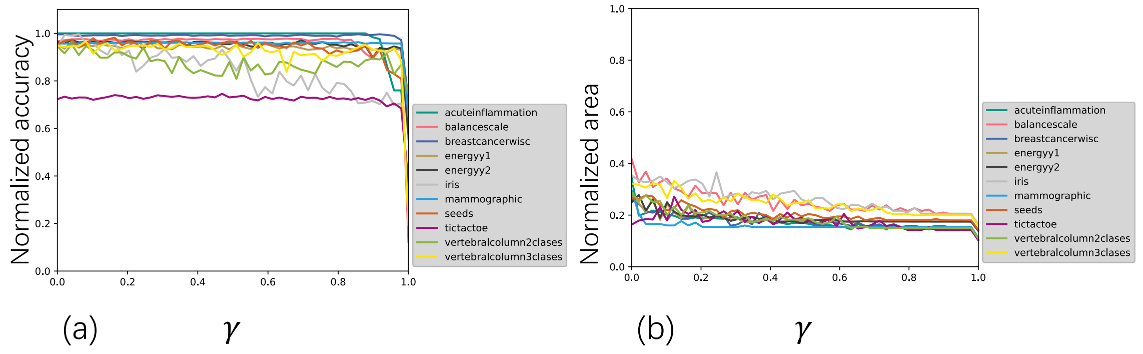


Figure 5.2: Results of EA approach with different γ values. (a) Normalized accuracies of 11 tasks. Each task indicated by a different color. (b) Normalized area of 11 circuits for the corresponding tasks.

In pruning experiment (as baseline), to analyze the impact of γ more clearly and to eliminate the disparate difficulties among different tasks, we normalize the accuracy by the value ($\gamma = 0$), which refers to the area-unaware training, and should theoretically achieve the best accuracy. It's important to recognize that, given the intricate nature of the non-convex optimization challenge akin to neural network training, this isn't always the case in practical scenarios. The resulting curves are displayed in Figure 5.1(a). Analogously, the circuit's area is also normalized by the value taking no consideration of area($\gamma = 0$). Because compared to the exact values, the relative reduction of accuracy and area serves as a more informative metric. The normalized area are visualized in Figure 5.1(b).

To investigate the effectiveness of the area-aware training of pruning within a comprehensive and generic scenarios, we calculate the averaged normalized accuracy across all tasks. The statistical result (w.r.t. 10 random seeds) of the averaged normalized accuracy (blue curve) and area (red curve) are summarized in Figure 5.3(a).

For the experimental results derived from the EA method, we conducted similar processing as mentioned above. It's worth noting that the accuracy and area for the EA method should be normalized by the values when $\gamma = 0$ in the pruning results, rather than its own values at $\gamma = 0$. Only in this way can we compare the results of the two methods.

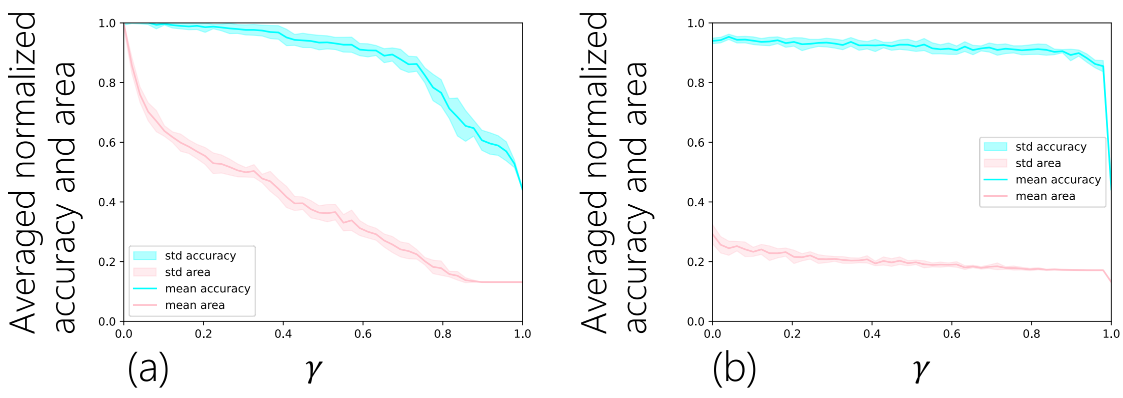


Figure 5.3: Averaged normalized accuracy and area, the curves and area denote the mean and standard deviation w.r.t. random seeds. (a) Result of Pruning. (b) Result of our EA method.

The curves showing the normalized accuracy and normalized area for 11 datasets, obtained using the EA method and varying with γ, are presented in the Figure 5.2.

The statistical results, w.r.t. 10 random seeds, for the average normalized accuracy (depicted by the blue curve) and area (represented by the red curve) of experiments conducted using the EA method are encapsulated in Figure Figure 5.3(b).

In order to obtain the Pareto front, we plot the entirety of normalized areas versus their respective normalized accuracies for all runs (random seeds) and all values of γ by the points in Figure 5.4. The blue dots represent the results from the EA method, while the green dots indicate the results from the pruning method. Subsequently, we can delineate the Pareto front of EA approach by the blue curve and that of pruning by the green curve respectively.

This section presents the results obtained from the experiments, along with tables and figures based on these findings.

5.3 Discussion

Based on the aforementioned experimental outcomes, we observed, as expected, as γ progressively increase to 1, both averaged normalized accuracy and area derived from pruning and EA approach in Figure 5.3 decline.

However, for the EA method, even at $\gamma = 0$, its normalized area has approximately reduced to 0.3. This observation is also corroborated when comparing the results from Table 5.4 and Table 5.5. This signifies that the EA led to a 3.3x reduction in circuit area while only incurring a 5% loss in accuracy.

In Figure 5.4, the pareto front of EA approach located above that of pruning. At the same level of accuracy, EA approach can achieve a smaller area than pruning. Table 5.6 extracted several trade-off points from Figure 5.4. It is evident that the baseline method yields a great area reduction at the accuracy level of 95%, namely from 100% to 36%. This indicates that gradient-based pruning methods have already achieved commendable results in area optimization. Even though, our proposed EA

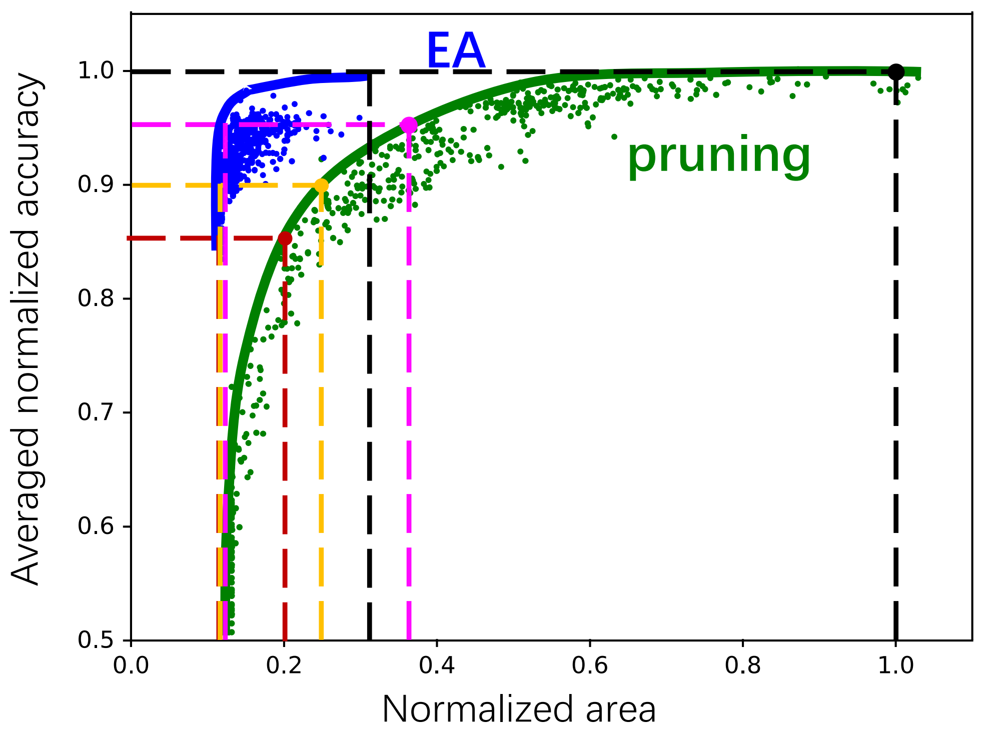


Figure 5.4: Scatter plot of normalized accuracy vs. area. Green and blue color denote pruning and EA respectively. The curves represent the Pareto front and the dash lines illustrate various potential trade-offs consistent with Pareto optimality.

method outperforms. The circuit area can be further reduced by 2.6×. At other level of accuracy , we can get the similar benefits, the specific reduction in area can also be gleaned from the Table 5.6. This clearly demonstrates the superiority of our approach.

In summary, the experiments validated the effectiveness of our proposed method in compact circuit design. Compared to the baseline, our approach achieves a more compact topology and reduced circuit area while maintaining comparable accuracy.

Normalized Accuracy (100%)	**Pruning Area (100%)**	**EA approach Area (100%)**	
100	100	32	↓3.1x
95	36	14	↓2.6x
90	25	13	↓1.9x
85	20	12	↓1.6x

Table 5.6: Comparison of accuracy-area trade-off from both methods.

6. Conclusion

In this work, we specifically focus on the compact design of printed analog neuromorphic circuits by explicitly establishing the area model. This model allows for the integration of circuit area into the design objective of printed neuromorphic circuits. Additionally, to facilitate the optimization of circuit topologies, we proposed an EA centric approach. This approach enable the simultaneous optimization of circuit topology and its parameters. We validated the effectiveness of the proposed method by comparing it with the gradient-based pruning method in our experiments. Our approach can achieve a smaller topology and reduced circuit area while maintaining equivalent accuracy. For instance, our approach can achieve a circuit area that is $3.1\times$ smaller than the pruning method, without any loss in accuracy. Our methodology offers a compelling strategy for realizing highly compact and resource-efficient neuromorphic circuits. Furthermore, our methodology can be seamlessly applied on other research, such as power-aware or variation-aware design of printed neuromorphic circuits that require topology optimization.

A. Appealdix: UML class diagram for neat-python

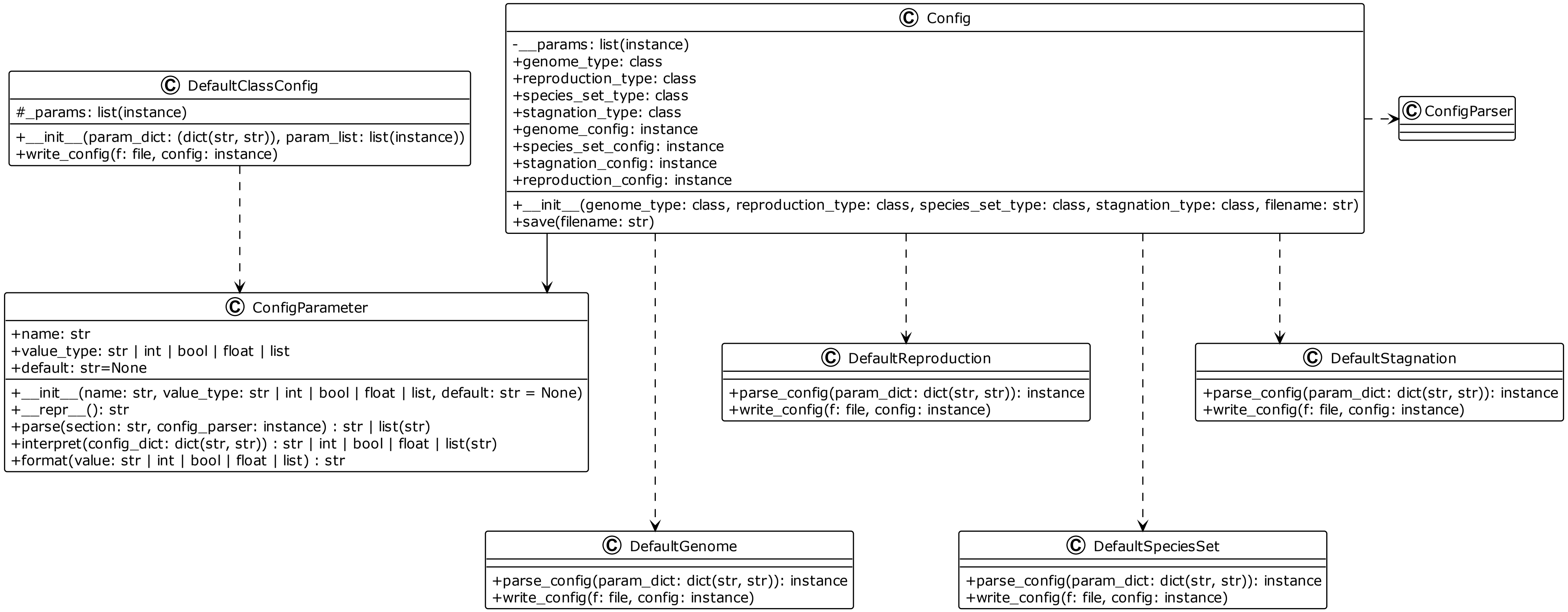
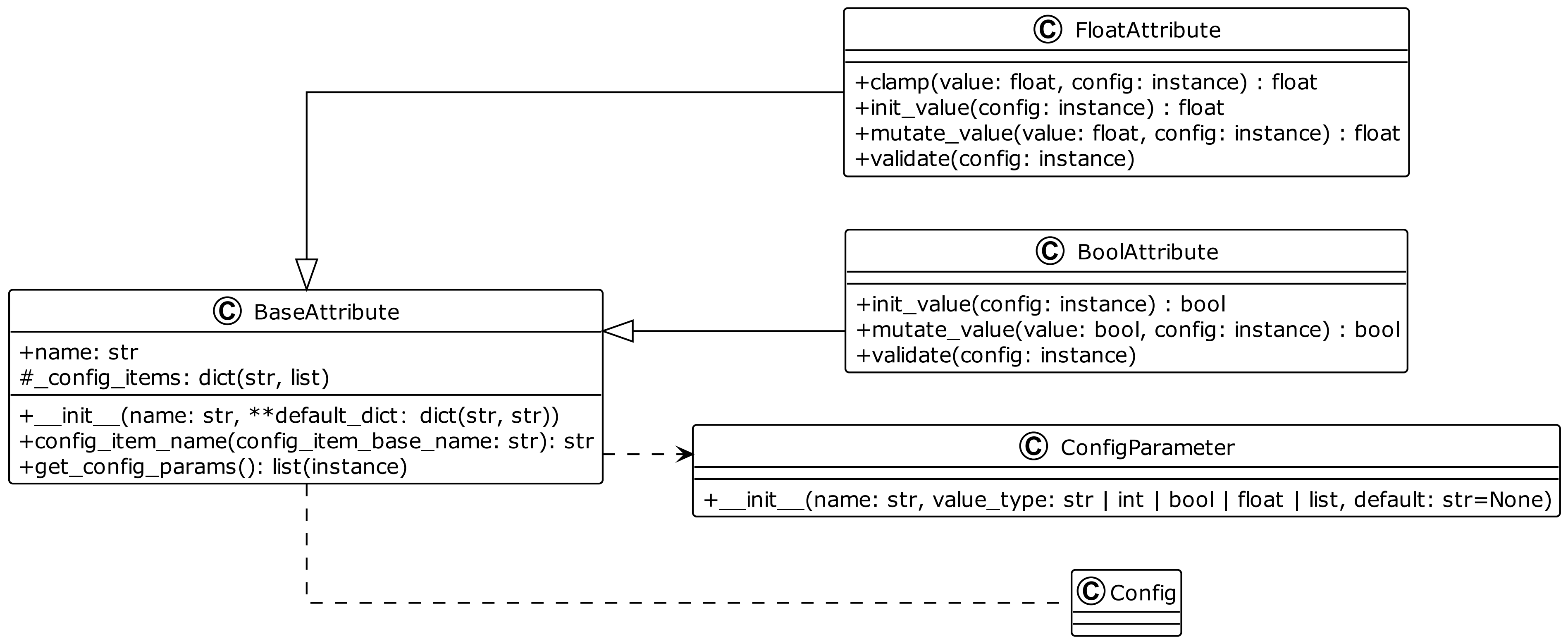

DefaultClassConfig
#_params: list(instance)
+__init__(param_dict: (dict(str, str)), param_list: list(instance))
+write_config(f: file, config: instance)
Config
-__params: list(instance)
+genome_type: class
+reproduction_type: class
+species_set_type: class
+stagnation_type: class
+genome_config: instance
+species_set_config: instance
+stagnation_config: instance
+reproduction_config: instance
+__init__(genome_type: class, reproduction_type: class, species_set_type: class, stagnation_type: class, filename: str)
+save(filename: str)
ConfigParser
ConfigParameter
+name: str
+value_type: str | int | bool | float | list
+default: str=None
+__init__(name: str, value_type: str | int | bool | float | list, default: str = None)
+__repr__(): str
+parse(section: str, config_parser: instance) : str | list(str)
+interpret(config_dict: dict(str, str)) : str | int | bool | float | list(str)
+format(value: str | int | bool | float | list) : str
DefaultReproduction
+parse_config(param_dict: dict(str, str)): instance
+write_config(f: file, config: instance)
DefaultStagnation
+parse_config(param_dict: dict(str, str)): instance
+write_config(f: file, config: instance)
DefaultGenome
+parse_config(param_dict: dict(str, str)): instance
+write_config(f: file, config: instance)
DefaultSpeciesSet
+parse_config(param_dict: dict(str, str)): instance
+write_config(f: file, config: instance)

Figure A.1: UML class diagram for neat.config: Does general configuration parsing; used by other classes for their configuration.



BaseAttribute
+name: str
#_config_items: dict(str, list)
+__init__(name: str, **default_dict: dict(str, str))
+config_item_name(config_item_base_name: str): str
+get_config_params(): list(instance)
FloatAttribute
+clamp(value: float, config: instance) : float
+init_value(config: instance) : float
+mutate_value(value: float, config: instance) : float
+validate(config: instance)
BoolAttribute
+init_value(config: instance) : bool
+mutate_value(value: bool, config: instance) : bool
+validate(config: instance)
ConfigParameter
+__init__(name: str, value_type: str | int | bool | float | list, default: str=None)
Config

Figure A.2: UML class diagram for neat.attributes: Deals with attributes used by genes.

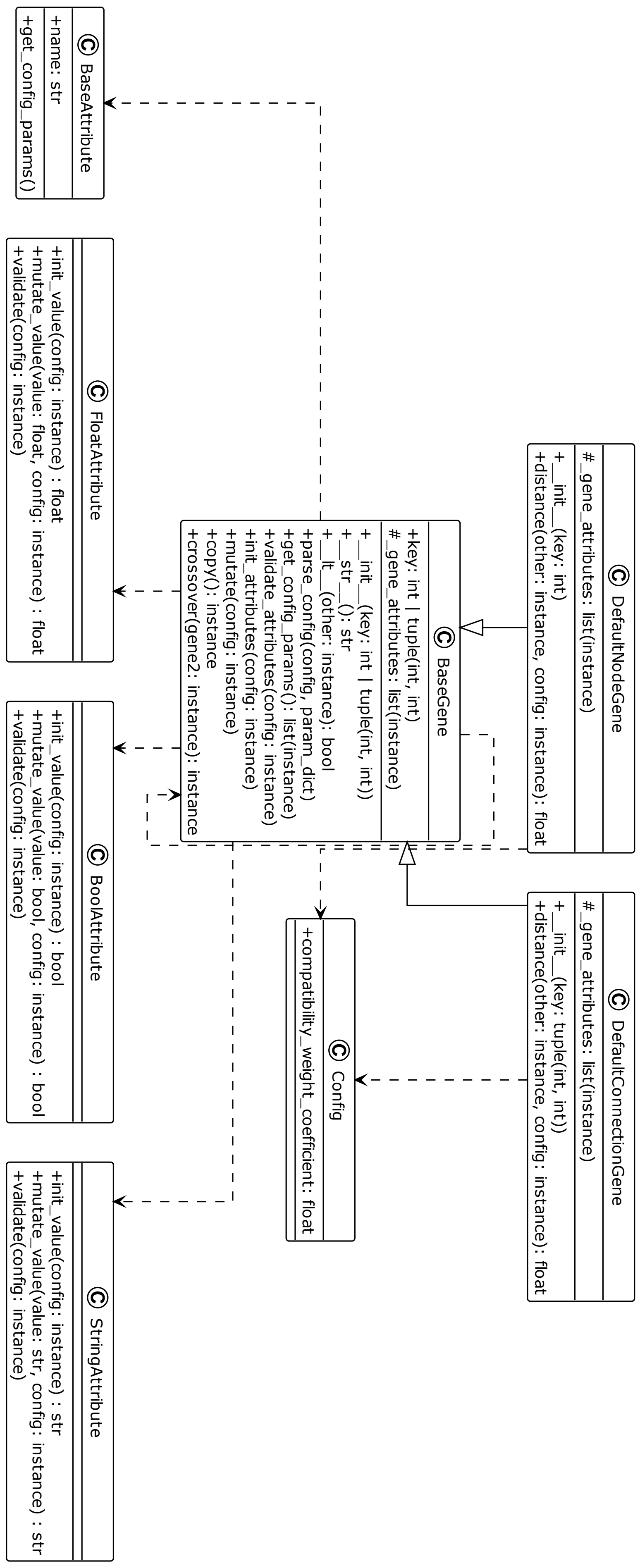

C DefaultNodeGene
#_gene_attributes: list(instance)
+__init__(key: int)
+distance(other: instance, config: instance): float
C DefaultConnectionGene
#_gene_attributes: list(instance)
+__init__(key: tuple(int, int))
+distance(other: instance, config: instance): float
C BaseGene
+key: int | tuple(int, int)
#_gene_attributes: list(instance)
+__init__(key: int | tuple(int, int))
+__str__(): str
+__lt__(other: instance): bool
+parse_config(config, param_dict)
+get_config_params(): list(instance)
+validate_attributes(config: instance)
+init_attributes(config: instance)
+mutate(config: instance)
+copy(): instance
+crossover(gene2: instance): instance
C Config
+compatibility_weight_coefficient: float
C BaseAttribute
+name: str
+get_config_params()
C FloatAttribute
+init_value(config: instance) : float
+mutate_value(value: float, config: instance) : float
+validate(config: instance)
C BoolAttribute
+init_value(config: instance) : bool
+mutate_value(value: bool, config: instance) : bool
+validate(config: instance)
C StringAttribute
+init_value(config: instance) : str
+mutate_value(value: str, config: instance) : str
+validate(config: instance)

Figure A.3: UML class diagram for neat.genes: Handles node and connection genes.

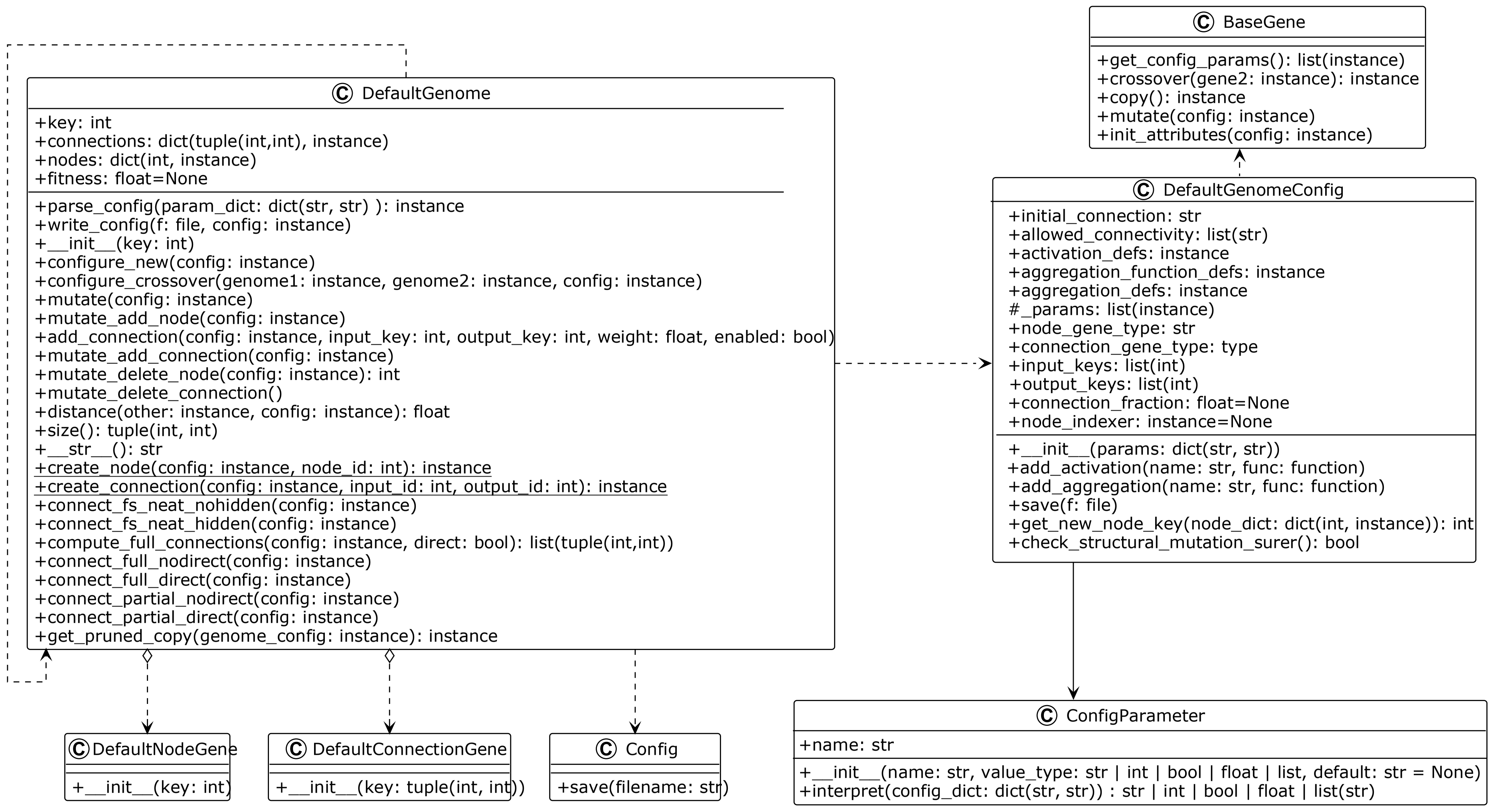

DefaultGenome
+key: int
+connections: dict(tuple(int,int), instance)
+nodes: dict(int, instance)
+fitness: float=None
+parse_config(param_dict: dict(str, str)): instance
+write_config(f: file, config: instance)
+__init__(key: int)
+configure_new(config: instance)
+configure_crossover(genome1: instance, genome2: instance, config: instance)
+mutate(config: instance)
+mutate_add_node(config: instance)
+add_connection(config: instance, input_key: int, output_key: int, weight: float, enabled: bool)
+mutate_add_connection(config: instance)
+mutate_delete_node(config: instance): int
+mutate_delete_connection()
+distance(other: instance, config: instance): float
+size(): tuple(int, int)
+__str__(): str
+create_node(config: instance, node_id: int): instance
+create_connection(config: instance, input_id: int, output_id: int): instance
+connect_fs_neat_nohidden(config: instance)
+connect_fs_neat_hidden(config: instance)
+compute_full_connections(config: instance, direct: bool): list(tuple(int,int))
+connect_full_nodirect(config: instance)
+connect_full_direct(config: instance)
+connect_partial_nodirect(config: instance)
+connect_partial_direct(config: instance)
+get_pruned_copy(genome_config: instance): instance
BaseGene
+get_config_params(): list(instance)
+crossover(gene2: instance): instance
+copy(): instance
+mutate(config: instance)
+init_attributes(config: instance)
DefaultGenomeConfig
+initial_connection: str
+allowed_connectivity: list(str)
+activation_defs: instance
+aggregation_function_defs: instance
+aggregation_defs: instance
#_params: list(instance)
+node_gene_type: str
+connection_gene_type: type
+input_keys: list(int)
+output_keys: list(int)
+connection_fraction: float=None
+node_indexer: instance=None
+__init__(params: dict(str, str))
+add_activation(name: str, func: function)
+add_aggregation(name: str, func: function)
+save(f: file)
+get_new_node_key(node_dict: dict(int, instance)): int
+check_structural_mutation_surer(): bool
DefaultNodeGene
+__init__(key: int)
DefaultConnectionGene
+__init__(key: tuple(int, int))
Config
+save(filename: str)
ConfigParameter
+name: str
+__init__(name: str, value_type: str | int | bool | float | list, default: str = None)
+interpret(config_dict: dict(str, str)) : str | int | bool | float | list(str)

Figure A.4: UML class diagram for neat.genome: Handles genomes (individuals in the population).

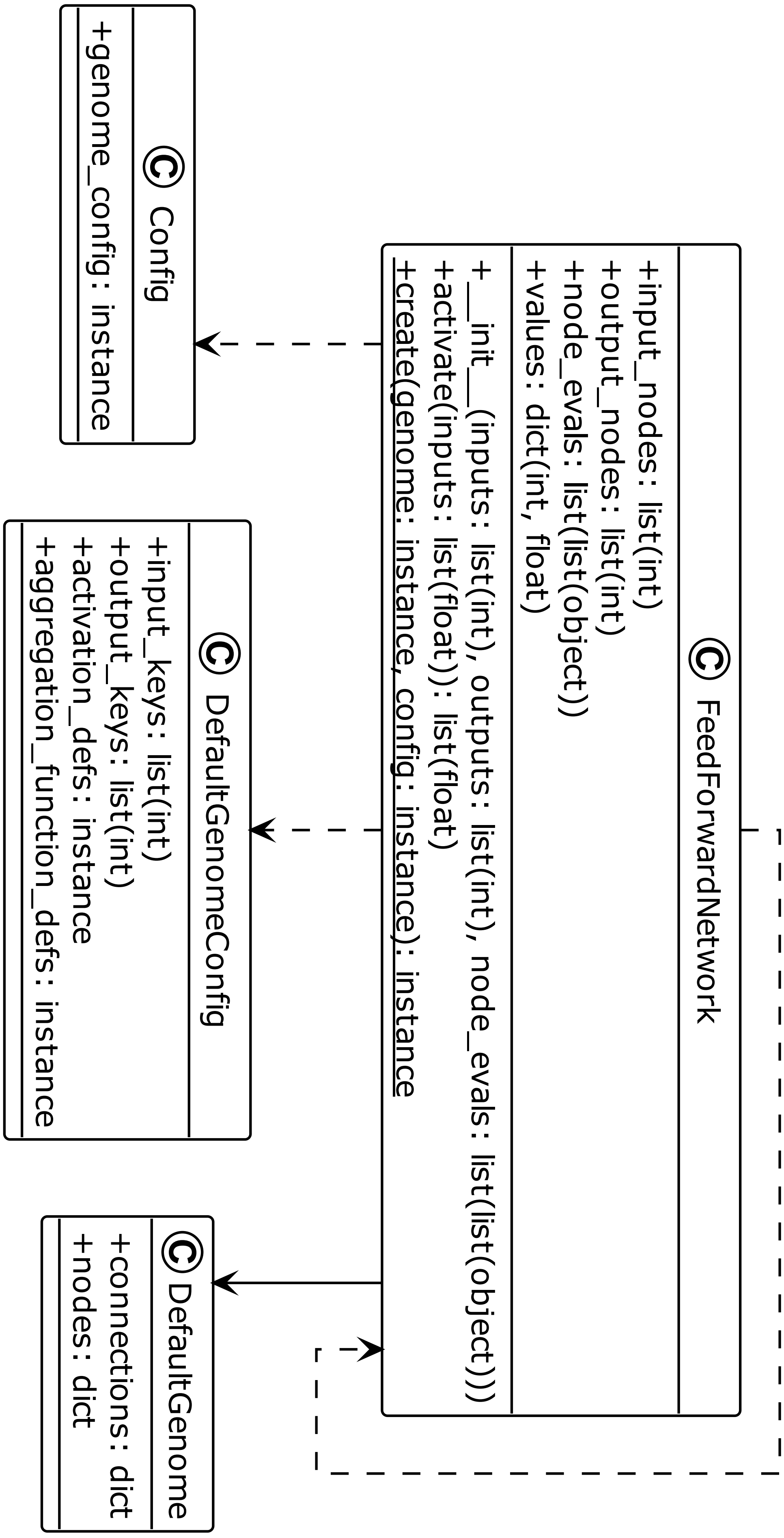

FeedForwardNetwork
+input_nodes: list(int)
+output_nodes: list(int)
+node_evals: list(list(object))
+values: dict(int, float)
+__init__(inputs: list(int), outputs: list(int), node_evals: list(list(object)))
+activate(inputs: list(float)): list(float)
+create(genome: instance, config: instance): instance
Config
+genome_config: instance
DefaultGenomeConfig
+input_keys: list(int)
+output_keys: list(int)
+activation_defs: instance
+aggregation_function_defs: instance
DefaultGenome
+connections: dict
+nodes: dict

Figure A.5: UML class diagram for neat.nn.feedforward: A straightforward feed-forward neural network NEAT implementation.

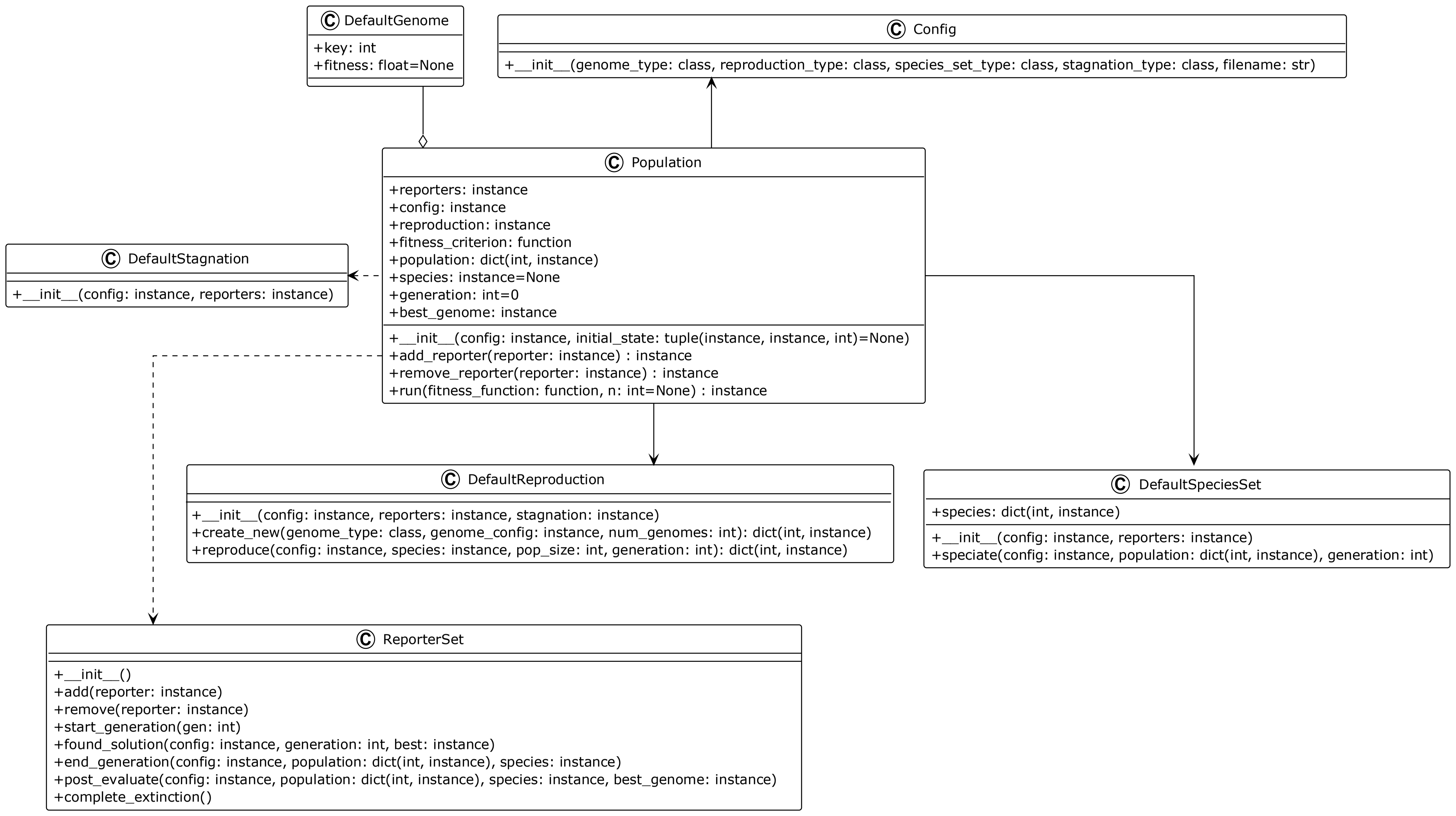

DefaultGenome
+key: int
+fitness: float=None
Config
+__init__(genome_type: class, reproduction_type: class, species_set_type: class, stagnation_type: class, filename: str)
Population
+reporters: instance
+config: instance
+reproduction: instance
+fitness_criterion: function
+population: dict(int, instance)
+species: instance=None
+generation: int=0
+best_genome: instance
+__init__(config: instance, initial_state: tuple(instance, instance, int)=None)
+add_reporter(reporter: instance) : instance
+remove_reporter(reporter: instance) : instance
+run(fitness_function: function, n: int=None) : instance
DefaultStagnation
+__init__(config: instance, reporters: instance)
DefaultReproduction
+__init__(config: instance, reporters: instance, stagnation: instance)
+create_new(genome_type: class, genome_config: instance, num_genomes: int): dict(int, instance)
+reproduce(config: instance, species: instance, pop_size: int, generation: int): dict(int, instance)
DefaultSpeciesSet
+species: dict(int, instance)
+__init__(config: instance, reporters: instance)
+speciate(config: instance, population: dict(int, instance), generation: int)
ReporterSet
+__init__()
+add(reporter: instance)
+remove(reporter: instance)
+start_generation(gen: int)
+found_solution(config: instance, generation: int, best: instance)
+end_generation(config: instance, population: dict(int, instance), species: instance)
+post_evaluate(config: instance, population: dict(int, instance), species: instance, best_genome: instance)
+complete_extinction()

Figure A.6: UML class diagram for neat.population: Implements the core evolution algorithm.

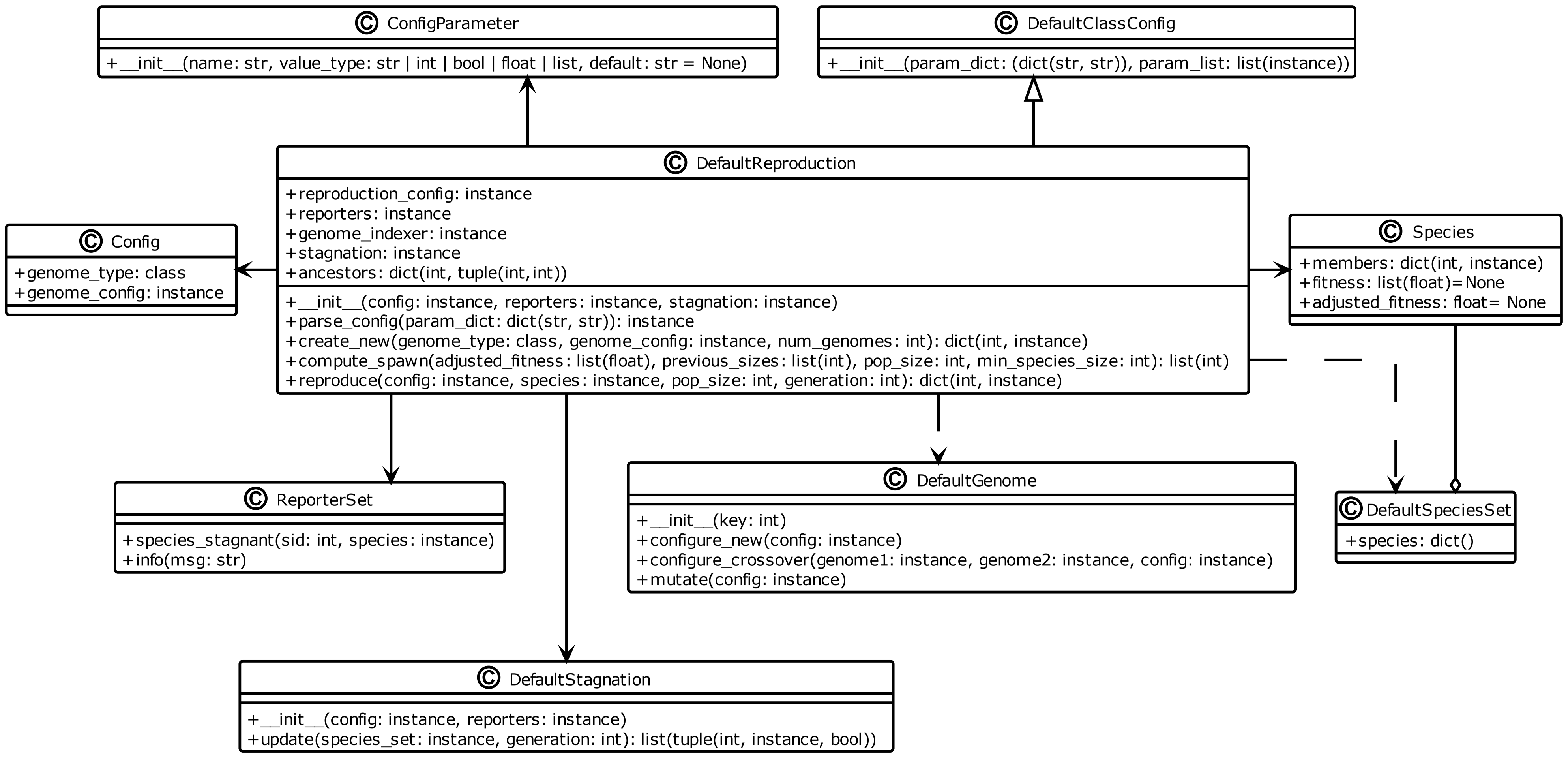

ConfigParameter
+__init__(name: str, value_type: str | int | bool | float | list, default: str = None)
DefaultClassConfig
+__init__(param_dict: (dict(str, str)), param_list: list(instance))
Config
+genome_type: class
+genome_config: instance
DefaultReproduction
+reproduction_config: instance
+reporters: instance
+genome_indexer: instance
+stagnation: instance
+ancestors: dict(int, tuple(int,int))
+__init__(config: instance, reporters: instance, stagnation: instance)
+parse_config(param_dict: dict(str, str)): instance
+create_new(genome_type: class, genome_config: instance, num_genomes: int): dict(int, instance)
+compute_spawn(adjusted_fitness: list(float), previous_sizes: list(int), pop_size: int, min_species_size: int): list(int)
+reproduce(config: instance, species: instance, pop_size: int, generation: int): dict(int, instance)
Species
+members: dict(int, instance)
+fitness: list(float)=None
+adjusted_fitness: float= None
ReporterSet
+species_stagnant(sid: int, species: instance)
+info(msg: str)
DefaultGenome
+__init__(key: int)
+configure_new(config: instance)
+configure_crossover(genome1: instance, genome2: instance, config: instance)
+mutate(config: instance)
DefaultSpeciesSet
+species: dict()
DefaultStagnation
+__init__(config: instance, reporters: instance)
+update(species_set: instance, generation: int): list(tuple(int, instance, bool))

Figure A.7: UML class diagram for neat.reproduction: Handles creation of genomes, either from scratch or by sexual or asexual reproduction from parents. Implements the default NEAT-python reproduction scheme: explicit fitness sharing with fixed-time species stagnation.

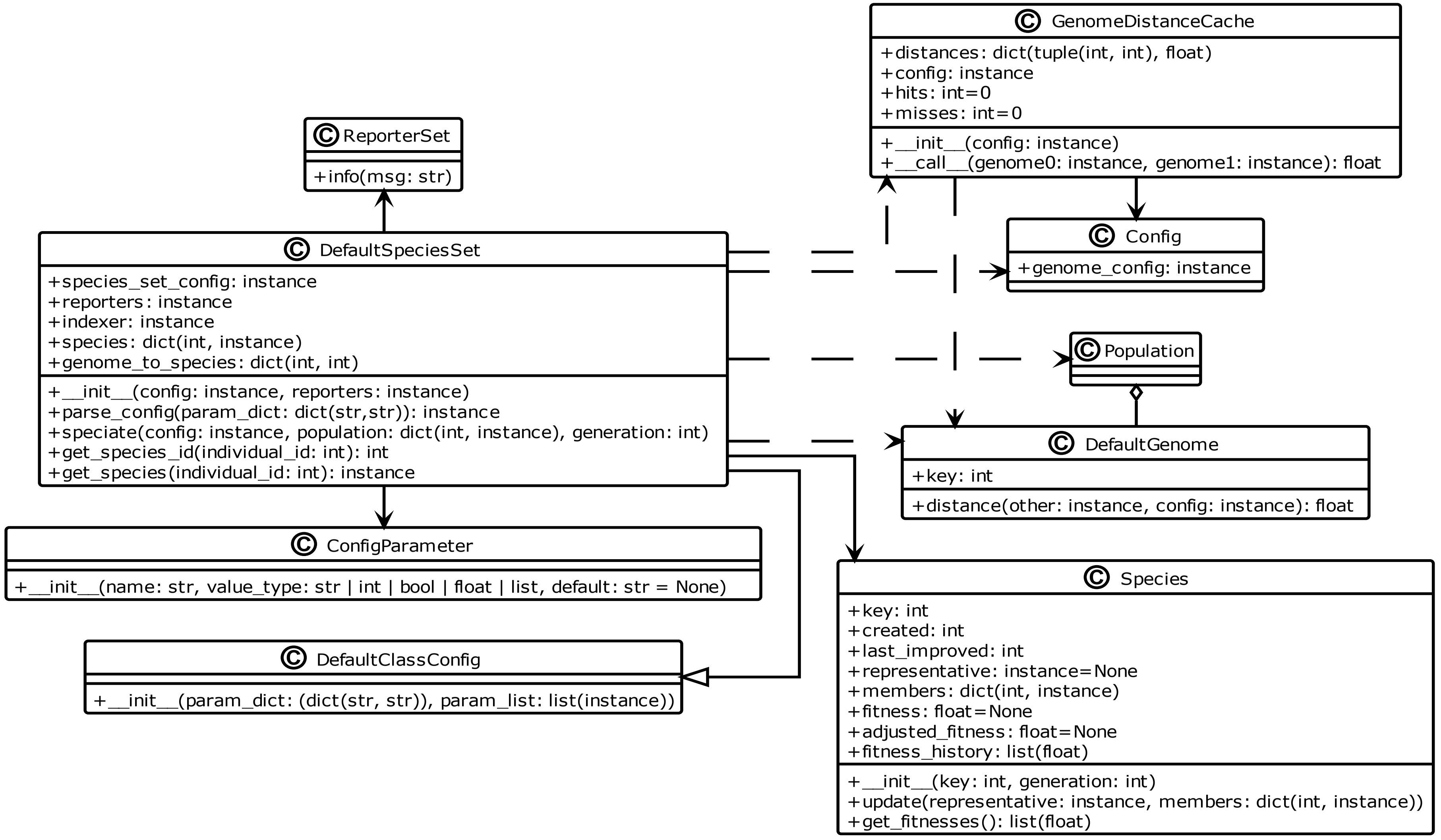

GenomeDistanceCache
+distances: dict(tuple(int, int), float)
+config: instance
+hits: int=0
+misses: int=0
+__init__(config: instance)
+__call__(genome0: instance, genome1: instance): float
ReporterSet
+info(msg: str)
DefaultSpeciesSet
+species_set_config: instance
+reporters: instance
+indexer: instance
+species: dict(int, instance)
+genome_to_species: dict(int, int)
+__init__(config: instance, reporters: instance)
+parse_config(param_dict: dict(str,str)): instance
+speciate(config: instance, population: dict(int, instance), generation: int)
+get_species_id(individual_id: int): int
+get_species(individual_id: int): instance
Config
+genome_config: instance
Population
DefaultGenome
+key: int
+distance(other: instance, config: instance): float
ConfigParameter
+__init__(name: str, value_type: str | int | bool | float | list, default: str = None)
DefaultClassConfig
+__init__(param_dict: (dict(str, str)), param_list: list(instance))
Species
+key: int
+created: int
+last_improved: int
+representative: instance=None
+members: dict(int, instance)
+fitness: float=None
+adjusted_fitness: float=None
+fitness_history: list(float)
+__init__(key: int, generation: int)
+update(representative: instance, members: dict(int, instance))
+get_fitnesses(): list(float)

Figure A.8: UML class diagram for neat.species: Divides the population into species based on genomic distances.

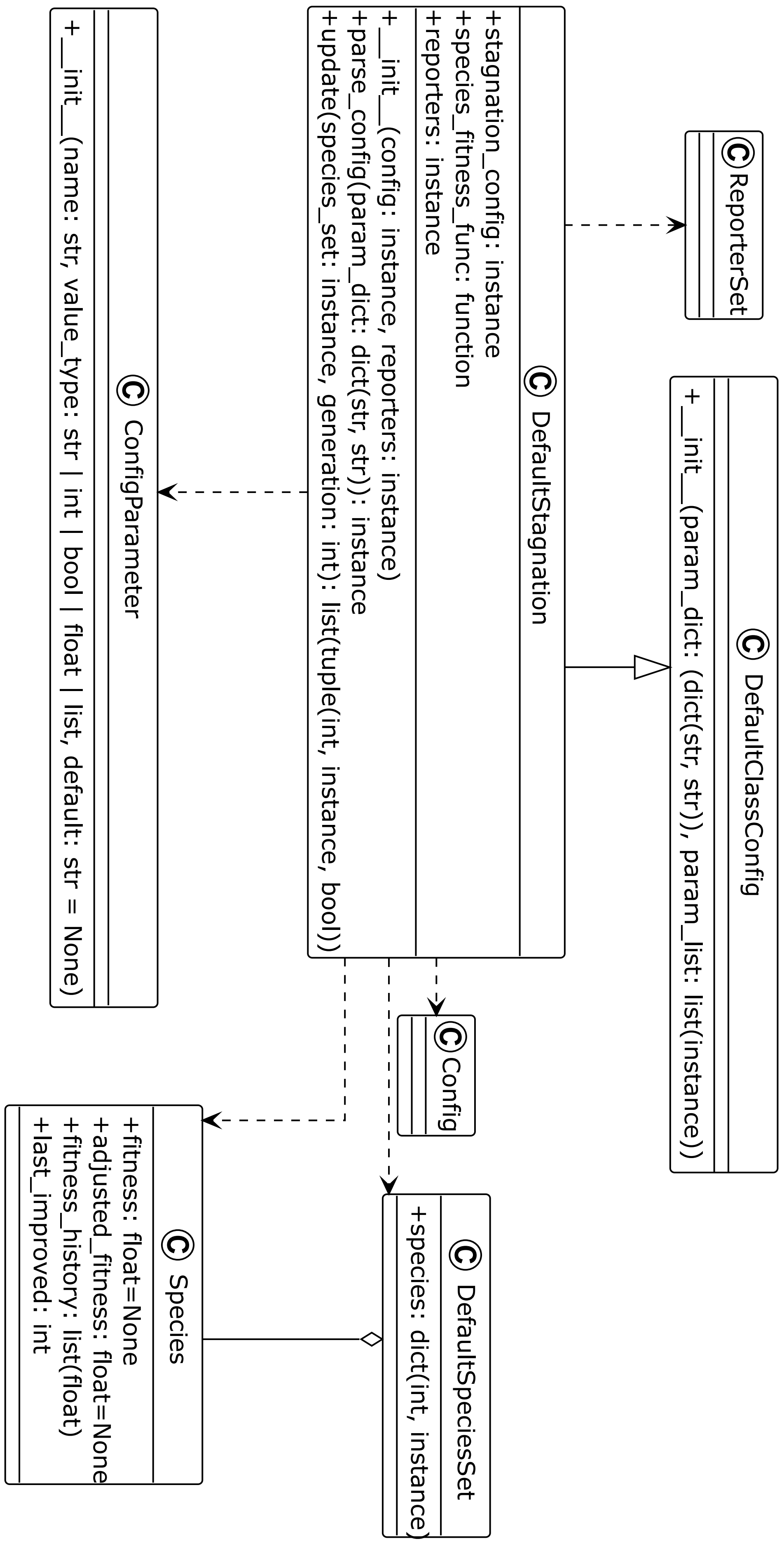

C ReporterSet
C DefaultClassConfig
+__init__(param_dict: (dict(str, str)), param_list: list(instance))
C DefaultStagnation
+stagnation_config: instance
+species_fitness_func: function
+reporters: instance
+__init__(config: instance, reporters: instance)
+parse_config(param_dict: dict(str, str)): instance
+update(species_set: instance, generation: int): list(tuple(int, instance, bool))
C Config
C ConfigParameter
+__init__(name: str, value_type: str | int | bool | float | list, default: str = None)
C DefaultSpeciesSet
+species: dict(int, instance)
C Species
+fitness: float=None
+adjusted_fitness: float=None
+fitness_history: list(float)
+last_improved: int

Figure A.9: UML class diagram for neat.stagnation: Keeps track of whether species are making progress and helps remove ones that are not.

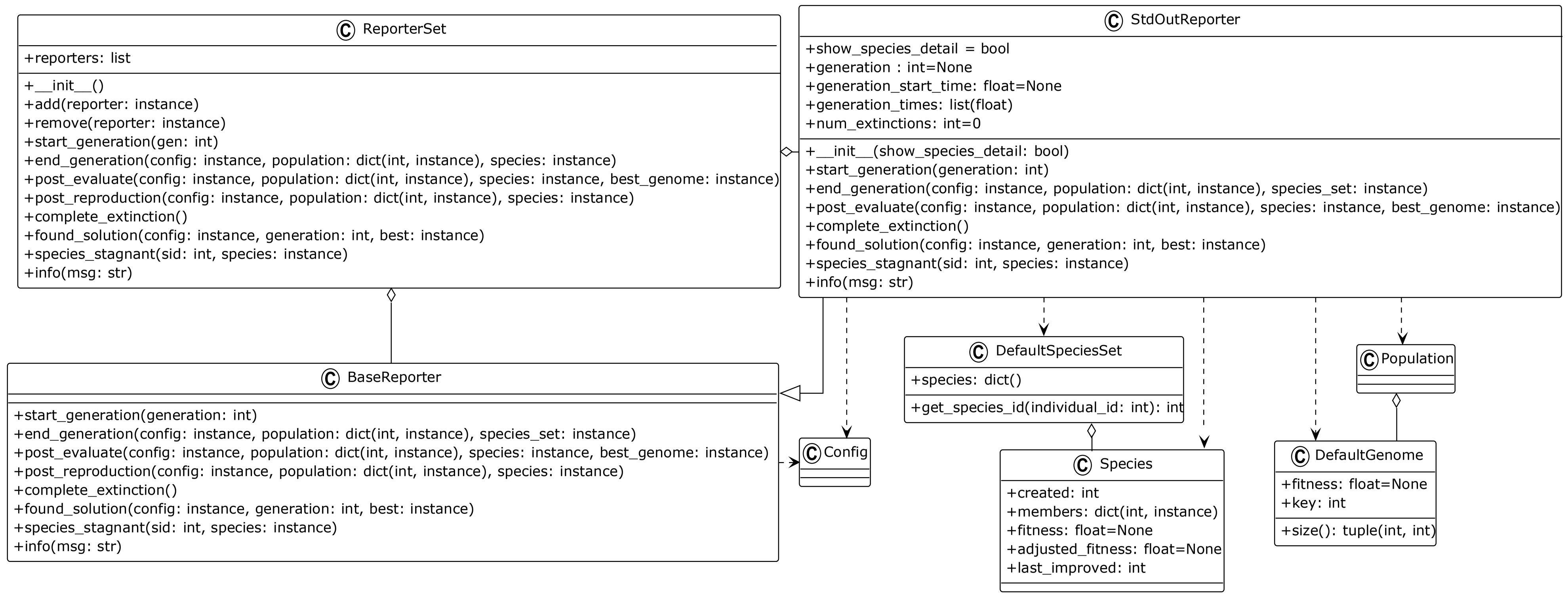

ReporterSet
+reporters: list
+__init__()
+add(reporter: instance)
+remove(reporter: instance)
+start_generation(gen: int)
+end_generation(config: instance, population: dict(int, instance), species: instance)
+post_evaluate(config: instance, population: dict(int, instance), species: instance, best_genome: instance)
+post_reproduction(config: instance, population: dict(int, instance), species: instance)
+complete_extinction()
+found_solution(config: instance, generation: int, best: instance)
+species_stagnant(sid: int, species: instance)
+info(msg: str)
StdOutReporter
+show_species_detail = bool
+generation : int=None
+generation_start_time: float=None
+generation_times: list(float)
+num_extinctions: int=0
+__init__(show_species_detail: bool)
+start_generation(generation: int)
+end_generation(config: instance, population: dict(int, instance), species_set: instance)
+post_evaluate(config: instance, population: dict(int, instance), species: instance, best_genome: instance)
+complete_extinction()
+found_solution(config: instance, generation: int, best: instance)
+species_stagnant(sid: int, species: instance)
+info(msg: str)
BaseReporter
+start_generation(generation: int)
+end_generation(config: instance, population: dict(int, instance), species_set: instance)
+post_evaluate(config: instance, population: dict(int, instance), species: instance, best_genome: instance)
+post_reproduction(config: instance, population: dict(int, instance), species: instance)
+complete_extinction()
+found_solution(config: instance, generation: int, best: instance)
+species_stagnant(sid: int, species: instance)
+info(msg: str)
Config
DefaultSpeciesSet
+species: dict()
+get_species_id(individual_id: int): int
Species
+created: int
+members: dict(int, instance)
+fitness: float=None
+adjusted_fitness: float=None
+last_improved: int
Population
DefaultGenome
+fitness: float=None
+key: int
+size(): tuple(int, int)

Figure A.10: UML class diagram for neat.reporting: Makes possible reporter classes, which are triggered on particular events and may provide information to the user, may do something else such as checkpointing, or may do both.

Bibliography

[1] Arif U Alam et al. “Fruit quality monitoring with smart packaging”. In: *Sensors* 21.4 (2021), p. 1509.

[2] Mohammad Ansari et al. “PHAX: Physical characteristics aware ex-situ training framework for inverter-based memristive neuromorphic circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2017), pp. 1602–1613.

[3] Giorgos Armeniakos et al. “Cross-layer approximation for printed machine learning circuits”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 190–195.

[4] Thomas Bäck and Hans-Paul Schwefel. “An overview of evolutionary algorithms for parameter optimization”. In: *Evolutionary computation* 1.1 (1993), pp. 1–23.

[5] Bowen Baker et al. “Designing neural network architectures using reinforcement learning”. In: *arXiv preprint arXiv:1611.02167* (2016).

[6] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. “Estimating or propagating gradients through stochastic neurons for conditional computation”. In: *arXiv preprint arXiv:1308.3432* (2013).

[7] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning.* Vol. 4. 4. Springer, 2006.

[8] Joseph Chang, Tong Ge, and Edgar Sanchez-Sinencio. “Challenges of printed electronics on flexible substrates”. In: *2012 IEEE 55th international midwest symposium on circuits and systems (MWSCAS)*. IEEE. 2012, pp. 582–585.

[9] Xin Chen et al. “Progressive differentiable architecture search: Bridging the depth gap between search and evaluation”. In: *Proceedings of the IEEE/CVF international conference on computer vision.* 2019, pp. 1294–1303.

[10] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[11] Xiangxiang Chu et al. “Fair darts: Eliminating unfair advantages in differentiable architecture search”. In: *European conference on computer vision.* Springer. 2020, pp. 465–480.

[12] Zheng Cui. *Printed electronics: materials, technologies and applications.* John Wiley & Sons, 2016.

[13] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural architecture search: A survey”. In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017.

[14] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[15] Xin He, Kaiyong Zhao, and Xiaowen Chu. “AutoML: A survey of the state-of-the-art”. In: *Knowledge-Based Systems* 212 (2021), p. 106622.

[16] Sepp Hochreiter. “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[17] Guohua Hu et al. “Functional inks and printing of two-dimensional materials”. In: *Chemical Society Reviews* 47.9 (2018), pp. 3265–3300.

[18] Joanna Izdebska-Podsiadły and Sabu Thomas. *Printing on polymers: fundamentals and applications.* William Andrew, 2015.

[19] Altynay Kaidarova et al. “Wearable multifunctional printed graphene sensors”. In: *NPJ Flexible Electronics* 3.1 (2019), p. 15.

[20] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[21] G Kirchhoff. “LXIV. On a deduction of Ohm’s laws, in connexion with the theory of electro-statics”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 37.252 (1850), pp. 463–468.

[22] Isidoro Ibanez Labiano and Akram Alomainy. “Flexible inkjet-printed graphene antenna on Kapton”. In: *Flexible and Printed Electronics* 6.2 (2021), p. 025010.

[23] Shujie Li et al. “Inkjet Printing of Perovskites for Breaking Performance–Temperature Tradeoffs in Fabric-Based Thermistors”. In: *Advanced Functional Materials* 31.1 (2021), p. 2006273.

[24] Chenxi Liu et al. “Progressive neural architecture search”. In: *Proceedings of the European conference on computer vision (ECCV).* 2018, pp. 19–34.

[25] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “Darts: Differentiable architecture search”. In: *arXiv preprint arXiv:1806.09055* (2018).

[26] Makoto Mizukami et al. “Flexible organic light-emitting diode displays driven by inkjet-printed high-mobility organic thin-film transistors”. In: *IEEE Electron Device Letters* 39.1 (2017), pp. 39–42.

[27] Robert A Nawrocki, Richard M Voyles, and Sean E Shaheen. “Neurons in polymer: Hardware neural units based on polymer memristive devices and polymer transistors”. In: *IEEE Transactions on Electron Devices* 61.10 (2014), pp. 3513–3519.

[28] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[29] Hieu Pham et al. “Efficient neural architecture search via parameters sharing”. In: *International conference on machine learning.* PMLR. 2018, pp. 4095–4104.

[30] Lutz Prechelt. "Automatic early stopping using cross validation: quantifying the criteria". In: *Neural networks* 11.4 (1998), pp. 761–767.

[31] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4780–4789.

[32] EH Rijnbeek. "Approximate Computing-Reconsidering the Analog Computer". In: (2019).

[33] Hecht-Nielsen Robert et al. "Theory of the backpropagation neural network". In: *Proc. 1989 IEEE IJCNN* 1 (1989), pp. 593–605.

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

[35] Abu Sebastian et al. "Memory devices and applications for in-memory computing". In: *Nature nanotechnology* 15.7 (2020), pp. 529–544.

[36] Linghao Song et al. "GraphR: Accelerating graph processing using ReRAM". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 531–543.

[37] Kenneth O Stanley and Risto Miikkulainen. "Efficient evolution of neural network topologies". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*. Vol. 2. IEEE. 2002, pp. 1757–1762.

[38] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[39] Kenneth O Stanley et al. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1 (2019), pp. 24–35.

[40] Jürgen Steimle. "On-skin computing". In: *Communications of the ACM* 65.4 (2022), pp. 38–39.

[41] Qizeng Sun et al. "Smart band-aid: Multifunctional and wearable electronic device for self-powered motion monitoring and human-machine interaction". In: *Nano Energy* 92 (2022), p. 106840.

[42] Dennis D Weller et al. "Programmable neuromorphic circuit based on printed electrolyte-gated transistors". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2020, pp. 446–451.

[43] Dennis D Weller et al. "Realization and training of an inverter-based printed neuromorphic computing system". In: *Scientific reports* 11.1 (2021), p. 9554.

[44] Zheqi Yu et al. "An overview of neuromorphic computing for artificial intelligence enabled hardware-based hopfield neural network". In: *IEEE Access* 8 (2020), pp. 67085–67099.

[45] Haibin Zhao et al. "Aging-aware training for printed neuromorphic circuits". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–9.

[46] Haibin Zhao et al. "Power-Aware Training for Energy-Efficient Printed Neuromorphic Circuits". In: *42nd IEEE/ACM International Conference on Computer-Aided Design*. 2023.

[47] Haibin Zhao et al. “Printed electrodermal activity sensor with optimized filter for stress detection”. In: *Proceedings of the 2022 ACM International Symposium on Wearable Computers*. 2022, pp. 112–114.

[48] Haibin Zhao et al. “Split Additive Manufacturing for Printed Neuromorphic Circuits”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2023, pp. 1–6.

[49] Feichi Zhou and Yang Chai. “Near-sensor and in-sensor computing”. In: *Nature Electronics* 3.11 (2020), pp. 664–671.

[50] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).

[51] Barret Zoph et al. “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.