Neuroevolution-based efficient field effect transistor compact device models

Ya-Wen Ho1, Tejender Singh Rawat1, Zheng-Kai Yang1, Sparsh Pratik1, Guan-Wen Lai1, Yen-Liang Tu1, and Albert Lin1,2

1Institute of Electronics Engineering, National Yang-Ming Chiao-Tung University, Taiwan
2Corresponding author: Albert Lin (e-mail: htdt5746@gmail.com).

This work was supported in part by the Ministry of Science and Technology (MOST), Taiwan under Grant 110-2221-E-A49-143.

ABSTRACT Artificial neural networks (ANN) and multilayer perceptrons (MLP) have proved to be efficient in terms of designing highly accurate semiconductor device compact models (CM). Their ability to update their weight and biases through the backpropagation method makes them highly useful in learning the task. To improve the learning, MLP usually requires large networks and thus a large number of model parameters, which significantly increases the simulation time in circuit simulation. Hence, optimizing the network architecture and topology is always a tedious yet important task. In this work, we tune the network topology using neuro-evolution (NE) to develop semiconductor device CMs. With input and output layers defined, we have allowed a genetic algorithm (GA), a gradient-free algorithm, to tune the network architecture in combination with Adam, a gradient-based backpropagation algorithm, for the network weight and bias optimization. In addition, we implemented the MLP model using a similar number of parameters as the baseline for comparison. It is observed that in most of the cases, the NE models exhibit a lower root mean square error (RMSE) and require fewer training epochs compared to the MLP baseline models. For instance, for patience number 10 with different number of model parameters, the RMSE for test dataset using NE and MLP in unit of log(ampere) are 0.1461, 0.0985, 0.1274, 0.0971, 0.0705, and 0.2254, 0.1423, 0.1429, 0.1425, 0.1391, respectively, for the 28nm technology node at foundry.

INDEX TERMS Metal oxide semiconductor (MOS), Machine learning, Neuroevolution, Semiconductor device compact model.

I. INTRODUCTION Metal Oxide Semiconductor (MOS) is the most widely used semiconductor device in integrated circuits (IC) due to its superior electrical properties. Owing to the rapid technological scaling, designing a compact model (CM) becomes a very crucial component in the design cycle for describing new device behaviors. CM provides the tool of incorporating the newly fabricated devices into the circuit simulation first before the physical hardware can be implemented. In addition, well-designed CMs, especially machine learning-based CMs, allow users to evaluate the device behaviors in different process conditions and the corresponding effects in circuit applications.

Mostly, device compact modeling is achieved by following complex physics and equations, which highly depend on each material layer and its interfaces, traps, and doping profile [1-3]. While conventional physical CMs with parameter fitting prevails in the past thirty years, the recent trend is that people are looking after using machine learning to expand or explore the capability of CMs. Many works have recognized machine learning and artificial neural networks (ANN) as a powerful tool for its accurate approximation, modeling, and optimization problem [4-13]. In most cases, standard multilayer perceptron (MLP) is used in constructing semiconductor device CMs. In this standard scenario using MLPs, the backpropagation (BP) [14] method is used to update the weights and reduce the overall error of the network. Stochastic gradient descent (SGD) or other gradient-based algorithms [15-17] have proved to be most effective in the optimization of large-scale deep neural network-based supervised learning models to update the network weights and biases.

Most of the previous works in semiconductor device compact models are based on dense connections using MLP[4, 6, 7]. Neuroevolution-based architecture for semiconductor CMs has not been seen to our best knowledge. One important earlier work related to neuroevolution is the physics-inspired network architecture.
by H. G. Xing et al. [7]. In Xing’s effort, the device physics has proved to be effective in tuning the network topology to improve the subthreshold current fitting without increasing the network size. Here we use Neuroevolution (NE) instead of physics inspiration to tune the network topologies because, for new devices, the physics models are not known. As a result, physics inspiration may not be possible. In addition, NE can explore a larger search space in network structures compared to physics inspiration. In [7], it is also shown that the MLP structure is not suitable for fitting device IV. The multiple orders of magnitude nature in semiconductor device current-voltage (IV) leads to difficulty in regression. Therefore, the adjustment in the standard dense connected layers is inevitable. Using MLP can lead to more neurons/weights required for the same accuracy or minimal improvement with increased network sizes. While MLP has achieved some success in semiconductor CMs [4, 6, 7], it is desired to further reduce the model size or increase the fitting accuracy under the same model size. The CMs are used mainly for intelligent semiconductor device parameter analyzer was used. The measurement upper and lower bounds in voltage can be determined based on applications, and all predicted data are observed. After multiplying the scalar function, the output and inputs are all scaled to the range (0, 1) for model training. The whole dataset will be split into train dataset and test dataset, with a 9:1(train: test) ratio. The test set offers the advantage of fitting all device IV data, ensuring better fitting and benefiting the subsequent SPICE simulations. Retaining the test set offers the advantages of detecting overfitting. In this work, we still retain a test set and calculate the test set scores.

II. METHODOLOGY

A. DATA PREPARATION AND MEASUREMENT

Various dimensions of MOSFET devices are used to train the machine-learning-based compact model. The 28nm NMOS with different gate lengths (L): 30nm, 40nm, 50nm, 60nm, 300 nm, 500 nm, and gate width (W): 400 nm, 1000nm, 1500nm, 3000nm, were fabricated and obtained from the foundry. Totally 11 devices are used. I_d-V_g_s curves with V_d_s step of 25mV curves at various V_d_s at 0.2V spacing are measured. V_g_s sweeping is from 0V to 1.5V, and V_d_s sweeping is from 0V to 1.6V. For the current-voltage (IV) measurement, Agilent 4156C semiconductor device parameter analyzer was used. The dataset was arranged according to the device gate length (L), gate width (W), gate voltage (V_g), drain voltage (V_d), and drain current (I_d). Our machine learning-based model will predict drain current (I_d) based on the four input parameters: W, L, V_g, V_d. To improve the subthreshold region modeling, we use logarithm transformation [42]:

\[ I' = \log_{10}(I) \]  

where \( I \) is the real current values, \( I' \) is transformed current values. The absolute values are taken for some data points at \( V_d=0 \) where residual negative current values are observed. After multiplying the scalar function, the output and inputs are all scaled to the range (0, 1) for model training. The whole dataset will be split into train dataset and test dataset, with a 9:1(train: test) ratio. The test set may not be required since fitting the semiconductor device IV is a pure interpolation problem [7, 43]. The measurement upper and lower bounds in voltage can be determined based on applications, and all predicted data points will fall within the bounds. Removing the test set offers the advantage of fitting all device IV data, ensuring better fitting and benefiting the subsequent SPICE simulations. Retaining the test set offers the advantages of detecting overfitting. In this work, we still retain a test set and calculate the test set scores.

B. MACHINE LEARNING ALGORITHM

Neuroevolution (NE) [18-21], a highly effective evolutionary algorithm (EA), uses GA for the evolution of network architecture or topology. NEs are highly beneficial in learning neural network building blocks (such as activation functions), hyperparameters, topologies, which are typically unavailable in gradient-based approaches [18]. In contrast to supervised learning, the NE algorithm does not require supervision [19], and this makes them best suited to reinforcement learning (RL) tasks. In RL, where the model learns by trial and error to try different action sequences to maximize the reward [22], the ability of NE to evolve the neural network topology along with weights has made them outperform baseline RL benchmark tasks [23, 24]. Along with this, NE has successfully outperformed many tasks such as pole balancing [19], evolving neural network controllers for robots [25, 26], video games [27, 28], and an automobile crash warning system [29]. It is worth mentioning that evolutionary algorithms (EA) [30] have been gaining popularity for their ability to alleviate the shortcoming of BP [31]. EA-based algorithms are bio-inspired from the principles of Darwin’s evolution theory to stochastically optimize the problem to build an efficient, robust, and adaptive system. EA comprises of four landmark evolutionary algorithms under its hood, namely: genetic algorithms (GA) [32, 33], evolution strategies (ES) [34], evolutionary programming (EP) [35], and genetic programming (GP) [36].

NE-based neural networks for designing compact models can be very promising. Using NE for the compact model would allow our network to deform, adapt, and shrink or expand according to the requirement. Nevertheless, NE is inefficient when it comes to optimizing a large number of weights and hence is generally limited to the relatively small network. On the other hand, hybrid methods, i.e., complementarity between stochastic gradient descent (SGD) and EA, where SGD optimizes the objective function and gradient-free EA evolves the neural structure [37-40], make it possible to utilize much larger networks. In this work, we have also developed a hybrid method for our purpose of compact device modeling. In our hybrid algorithm, we have used GA for selection, crossover, and mutation to evolve the network topologies and SGD-based algorithm Adam [15] to update the weights with the error BP algorithm. Some of the results in this work can be found in a student thesis [41].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3130254, IEEE Access

Author Name: Preparation of Papers for IEEE Access (February 2017)
We have implemented our models using Python 3.7.10 [44], Numpy 1.20.2 [45], Scipy 1.6.2 [46], Tensor-flow 2.3.0 with Keras API [47], Scikit-Learn 0.24.2 [48], Matlab, Pandas 1.2.5 [49]. In this work, we have considered the multi-layer perceptron (MLP) model as the baseline model with which we can compare our model. The reason for choosing MLP as a baseline model is due to its simple architecture and wide usage in semiconductor device CMs.

1) MULTI-LAYER PERCEPTRON (MLP)
MLP model is the most commonly used, fully connected feed-forward network, shown in Fig. 1.B MLP can have any number of hidden layers between the input and output layers. Each neuron within the hidden layers can have multiple inputs \(x_i\) which are multiplied with the corresponding weights \(w_i\). In addition to this, bias is added to every input. At last, the output signal is formed after operated with the activation function, as shown in equation 2. The activation function contributes to the nonlinear characteristic of an artificial neural network. Here sigmoid activation is used for MLP models. In MLP, manually optimizing the numbers of hidden layers and the number of neurons in each layer is a tedious yet important task.

\[
y = f_{activation}(\sum_{i=1}^{n} x_i w_i + bias) \tag{2}
\]

2) NE USING GA
In our proposed model, we have used GA to evolve the neural network architecture by itself. Initially, we will start with a partially connected feedforward network, and the neurons in a block of each layer are randomly connected to the neurons of the blocks in the previous layer, as shown in Fig. 1.B As the epochs advance, our model will evolve within the defined boundaries in terms of the number of hidden layers and the number of blocks in each layer. Here, we introduce a block instead of a neuron as the basic unit to construct the topology-optimized neural networks. Instead of neurons, every hidden-layers will have multiple blocks, and every block can have one to multiple neurons. Therefore, each hidden layer is composed of several blocks, and there will be at least one neuron in each block. To evolve the network using GA, we will tune (i) network-depth, i.e., the numbers of hidden-layers \(N_{layers}\), (ii) network-width, i.e., the number of blocks \(N_{blocks}\) in each layer and the numbers of neurons \(N_{neurons}\) in each block, and (iii) the activation function of each block in a hidden-layer. The partially connected model will then undergo evolution using GA and evolve to the best-fitted compact device model. Circuit simulators, in general, require a small compact model size. Based on our test, <20 neurons per hidden layer and <5 hidden layers lead to more reasonable CPU time in HSPICE. The total parameters in a CM should be kept <1000. The use of mixed tanh and sigmoid as activation functions in different blocks leads to improved performance.
performance, reference to single activations. Therefore, we allow mixed activations in neuro-evolved networks. The mixture of activation functions is also used in [7]. The details of the evolution scheme are discussed in the next section.

III. EVOLUTION OF NEURAL NETWORK ARCHITECTURE

GA, in general, follows the concept of propagating the genes of the fittest individual for successive generations to find the optimum solution. Initially, GA starts with a population of individuals, and each individual represents a potential solution to a specific problem. These individuals, over multiple generations, are evaluated using a fitness function and are assigned a fitness value or score. An individual with a better or poor-performing solution will be assigned a higher or lower fitness value or score. The fittest individuals, selected based on these scores for reproduction, will pass their genes to the next generation. This process of passing genetic features or genes from parents to their offspring occasionally results in better scoring solutions. The flowchart of the genetic algorithm is shown in Fig. 1.C

A. ENCODING STRATEGY

Genome representation plays an important role in the whole evolution process since evolutionary operations are performed by manipulating the genomes. A proper encoding scheme makes crossover and mutation easier and more efficient. In a typical genetic algorithm, a potential solution for a problem is usually encoded into a fixed-length genome. For example, binary encoding encodes the solutions into a string of 1 or 0 [50]. In our work, an optimal number of layers and neurons of a neural network is unknown, and thus we have to use a variable-length genome encoding method to represent our neural network.

**Genome (Genotype)**

| Gene ID | Gene (1,1) | Gene (1,2) | Gene (2,1) | Gene (2,2) | Gene (3,1) | Output Gene (0,0) |
|---------|------------|------------|------------|------------|------------|------------------|
| Neuron number: |
| Activation function: |
| Connected ids: |
| 2 | Tanh | 3 | Sigmoid | 3 | Sigmoid | 4 | Tanh | 2 | Sigmoid | 1 | None |

**Network (Phenotype)**

![Figure 2. The encoding strategy of network architecture. Each network architecture is represented as genome and its respective phenotype, i.e., physical representation of the genome.](image)

Encoding strategy involves the representation of the genome of each individual. In our work, the genome represents the network architecture and can have a variable length in each generation, i.e., metameric variable-length genome [51]. Each gene in our genome represents the block. Each gene is assigned a set of identification (ID) numbers \((m, n)\), where \(m\) represents the number of the hidden layer, and \(n\) represents the block number, as shown in Fig. 2. In Fig. 2, there are two blocks in the first hidden layer, represented by ID \((1,1)\) and \((1,2)\). Gene ID not only helps to define network structure but makes model building with Tensorflow Keras functional API much easier.

Network architecture or topology information will be represented by its genome. The genome represented by a list of genes incorporates the information of model parameters such as neuron number, activation function, and input connection, i.e., connection id. Neuron number and activation function describe the number of neurons and the
activation function defines for that particular block, respectively. Connection id defines the gene ID of the previous layer with which that particular block or gene has to establish the connections. For example, as shown in Fig. 2, gene ID (1,1) has to establish the connection with the input layer neuron, defined as (-1,1) and (-1,2). Therefore, the connection id directs each block to establish the connection to the blocks of the previous layer. It is worth mentioning that there can be more than one block in a single hidden layer and more than one single neuron in one block. Relative to MLP, the advantage of having different blocks in a hidden layer is that we can assign different activation functions within the same hidden layer. This increases the search space in GA. In addition, while one block can contain only one neuron, using blocks as the basic unit in GA does not diminish the search space and does not decrease the flexibility in network topology evolution.

B. POPULATION INITIALIZATION

During the initialization process, we first define the random network topologies, as discussed earlier. A population of these N individuals is initialized randomly within the constraints of hyper-parameter. The algorithm for initializing the population is described in Table I.

When GA started, i.e., in first-generation, all individuals in the population have random hidden layers and random blocks within each hidden-layers. For subsequent generations, the architecture will evolve within the defined hyper-parameters that define the network architectures, as shown in Table II. Here, we limit the minimum number of hidden layers (N_{layers}) to two layers, since in our previous experience of model fitting, it requires at least two hidden layers to get satisfactory fitting accuracy. The lower limit of block in each hidden layer (N_{block}) and the number of neurons (N_{neuron}) in each block are both set to one. Each block in each layer is given a corresponding gene id, as mentioned above.

Once the network architectures and ids are defined, the connection is then assigned randomly between layers to form a randomly connected network. The rule of connection is described below:
1. First hidden layer: All neurons of each block in the first hidden layer should connect at least three inputs.
2. Other hidden layers: Blocks only connect to the blocks in the previous layer. Blocks in the same layer are not allowed to connect with themselves.
3. Output layer: Connect at least half of the number of blocks in the last hidden layer.

| Table I. Algorithm for initializing the population. |
|---------------------------------------------------|
| **Algorithm 1: Population initialization** |
| **Input:** The population size N_{pop}, the maximal number of hidden layers N_{layers}, the maximal number of blocks in each hidden layer N_{block}, and the maximal number of neurons in each block N_{neuron}. |
| **Output:** Initialized population P_{0} |

| Table II. List of hyperparameter that defines neural network architecture. |
|--------------------------------------------------------------------------|
| **Hyper parameter** | **Type** | **Scope** |
|---------------------|----------|-----------|
| N_{layers} | Integer | [2, 5] |
| N_{block} | Integer | [1, -] |
| N_{neuron} | Integer | [1, -] |
| Activation function | Categorical | [Sigmoid, Tanh] |

C. FITNESS EVALUATION

Each individual in generations is evaluated and given a fitness score as a measure of parent solution for reproduction. Because model fitting is a regression problem, mean squared error is the best way to evaluate fitness scores. Each model decoded from the genome is trained on the training set using backpropagation, and the fitness score is estimated on the validation set reserved from the training set. We have used the concept of early stopping to efficiently train our model. Keras early stopping callback is implemented to monitor the training process. Once the loss on the validation set no longer decreases within a number of epochs, defined as patience...
number, the training will terminate. Here we set the patience number as 10 in GA, and the maximum epoch is set to 1000 epochs, the training process usually concludes before reaching the maximum epoch, and the mean squared error of validation set in the last epoch is assigned to be the fitness score of individuals. This approach not only prevents overfitting but also saves significant training and evolution time.

D. SELECTION METHOD
The selection process helps in identifying the fittest eligible parents from the population to reproduce offspring. There is plenty of selection methods used in genetic algorithms, such as roulette wheel selection, rank selection, and tournament selection is considered because of its ability to provide a better selection strategy in terms of convergence rate and time complexity [53]. Binary tournament selection is made by randomly choosing two individuals from the population, and these two individuals compete with each other in terms of their fitness scores. The one with a better fitness score, namely, a smaller mean squared error, is selected as one of the parents. The other parent will be selected by repeating the selection process again. These two parents will be added to the mating pool, which contains different pairs of parents for the reproduction of the next generation. This procedure is illustrated in Fig. 3. The size of the mating pool is set to be the population size.

**Figure 3.** Two parents are selected based on fitness scores using the binary tournament selection method. Spatial crossover and mutation on parent’s genomes to produce fit and diverse offspring.

E. OFFSPRING REPRODUCTION (CROSSOVER AND MUTATION)
Individual genomes in our population can have variable lengths and are therefore expressed by a metameric variable-length genome [51]. Splitting such genes from random points can lead to loss of information during the crossover, and the offspring may not be similar to their parents. To maintain the phenotype space of the network, we used the spatial recombination [51, 54, 55] operator for crossover. Using this method, we consider the structure or the spatial distribution of genomes, and then the crossover point can only be at the boundary of layers. This preserves the localized relationships between the child and parent genomes. In simple words, the spatial recombination operator will split the genome randomly between hidden
layers instead of blocks or neurons, therefore maintaining all the blocks in a hidden layer intact, as shown in Fig. 3.

Two-parent genomes will create two offspring genomes by doing a crossover. Due to variable-length genome representation in our problem, the length of offspring genomes may be different from the length of parent genomes. The crossover layer does not need to match in the two-parent genomes, and this can lead to the mismatch of information, such as inconsistent input IDs and repeated genes, as shown in Fig. 3.

In this work, a mutation operation is performed to correct the defects in the genomes after crossover, i.e., inconsistent gene IDs and repeated genes. The mutation operation can fix the inconsistent connections ids by randomly reassigning them from the gene ID of the previous layer after crossover. Similarly, for repeated genes, the mutation process replaces the redundant genes from the genome and corrects the connection id by reassigning them, as shown in Fig. 3.

**F. ELIMINATION**

During the elimination process, the individuals in the population, including the parents and the children, are all evaluated and ranked according to the fitness score. Those who have better performance are allowed to survive and have chances to evolve, while the others will be eliminated. This approach helps preserve the best candidate from being discarded in this process. Here we make the population size the same by eliminating half of the individuals after crossover. After the evolution process, the neuro-evolution algorithm will return the best individual who has the best fitness score in the last generation. The fittest individual will then compare to the baseline model in the next section.

**IV. RESULTS AND DISCUSSION**

In Table III, we have conducted five NE runs using patience=10. We can see that at 5 NE runs, all neuro-evolution cases exceed the performance of MLP models. The patience=10 is used for these five NE runs. The patience=10 is a reasonable value and also the default value in Scikit-Learn library [48]. Other patience values can be used to run the NE. Instead of using different patience values to re-run the NE, we use the trained model at patience=10 and continue the training at higher patience. This practice can lead to non-optimal topologies at the specific patience values, but it can give some insight into the effectiveness of NE and the models optimized at patience=10. It can be seen from Table III, the NE model with continued training at patience=50, 100 still shows lower RMSE values in reference to MLP baselines. It is worth mentioning that for all programs in this work, we uniformly set the random seeds to 1 at the beginning of the program execution. Therefore, the NE run does not optimize the random seeds for individuals. The training of an individual can potentially start from any seeds, and there is no guarantee that selected individuals will receive the same random seed in the next evaluation. We use this practice since we believe the optimized network topologies should be superior independent of seeds. Since the random seed is not optimized in NE, the baseline model of MLP is evaluated using random seed=1 for a fair comparison. In addition, to show the strength of NE, we actually rotate the MLP random seed from 1 to 30 and record the lowest RMSE values, labeled as RMSE\text{min} in Table III. It can be seen that the RMSE\text{min} is still greater than the RMSE of NE optimized individuals in this somehow unfair comparison. This indicates the effectiveness of NE using GA here.

In this work, the connections from the block in i\text{th} layer are only to the (i-1)\text{th} layer, preventing the possibility of connecting to two layers preceding the i\text{th} layer. This scheme reduces the searching space in the NE. Nonetheless, based on our numerical test, allowing the connections to more than the blocks in the (i-1)\text{th} layer can lead to difficulty in locating the optimized topologies for neural networks. The potential reason for the observed difficulty can lie in the too complex topologies in the neural networks, which leads to a vast searching space and poorer convergence in training. Regarding partially connected networks, instead of dense connections, Xing et al. also demonstrate their advantages[7]. Essentially, if we want to shrink the model size, we need to use a partial connection to maximize the critical usage of the neurons and connections. A fully connected structure with the same neuron and hidden layer number encompasses the subsets of partially connected networks but requires more training time and is larger in size.
On the other hand, allowing the connection only to the previous layer during evolution gives a smaller search space but faster locating the optimized topologies and faster convergence in network training. Fig. 4 shows the fitness statistics of GA during evolution, including the best individuals and the averages in one generation. Fig. 5 shows the scatter plots of the test dataset with patience=100 and with a similar number of parameters for NE and MLP, respectively. It can be observed that the RMSE values of neuro-evolution cases are reduced significantly in reference to MLP baseline models.

Table III. Performance indices of NE and MLP models. The RMSE and $R^2$ is calculated after min-max un-scaling but before inverse log transformation.

| Initial setup | Number of parameters | Patience | Training epochs | RMSE$_{test}$ | RMSE$_{min}$ | $R^2$-score |
|---------------|----------------------|----------|-----------------|---------------|-------------|-------------|
| (N$_{layers}$, N$_{blocks}$, N$_{neurons}$) | NE | MLP | NE | MLP | NE | MLP | NE | MLP | NE | MLP |
| 5, 3, 4 | 1026 | 592 | 0.1791 | 0.2570 | 0.1922 | 0.2650 | 0.1868 | 0.9931 | 0.9857 | 0.9920 | 0.9847 |
| 5, 4, 6 | 10398 | 585 | 0.1295 | 0.1913 | 0.1688 | 0.2441 | 0.1696 | 0.9946 | 0.9886 | 0.9938 | 0.9870 |
| 5, 6, 8 | 10265 | 687 | 0.1163 | 0.1589 | 0.1610 | 0.1774 | 0.1596 | 0.9971 | 0.9945 | 0.9965 | 0.9932 |
| 5, 6, 10 | 10365 | 687 | 0.1183 | 0.1793 | 0.1688 | 0.1891 | 0.1696 | 0.9964 | 0.9931 | 0.9954 | 0.9922 |
| 5, 6, 8 | 10365 | 325 | 0.0810 | 0.1608 | 0.0973 | 0.1660 | 0.0920 | 0.9870 | 0.9944 | 0.9979 | 0.9940 |
| 5, 6, 10 | 10365 | 677 | 0.0792 | 0.1653 | 0.0920 | 0.1657 | 0.0920 | 0.9870 | 0.9944 | 0.9979 | 0.9940 |

*RMSE$_{test}$: Random seed value = 1. All seeds are set to 1 in this work except the calculation of RMSE$_{min}$.
*RMSE$_{min}$: The minimum RMSE of MLP model trained with 30 random seeds.

It is worth to mention that the effectiveness of NE can be diminished as the network size grows. This means that when the parameter number in a network is allowed to grow, the MLP can also be effective, and the relative enhancement of NE over MLP can become insignificant. This is as expected since, for a fixed dataset size, the fitting becomes easier and more manageable when the network size increases in terms of neuron numbers and the number of hidden layers. In this work, we count the parameter number in our networks and compare the performance of

It is worth to mention that the effectiveness of NE can be diminished as the network size grows. This means that when the parameter number in a network is allowed to grow, the MLP can also be effective, and the relative enhancement of NE over MLP can become insignificant. This is as expected since, for a fixed dataset size, the fitting becomes easier and more manageable when the network size increases in terms of neuron numbers and the number of hidden layers. In this work, we count the parameter number in our networks and compare the performance of...
the neuro-evolution networks to the baseline MLP models with similar parameter numbers. It should be noted that although the effectiveness of neuro-evolution networks can be more pronounced at small network sizes and becomes less pronounced at larger network sizes, the method of topology optimization in network architecture is still of very high value for compact device modeling. This is because we need a small network for CMs in order to manage the circuit simulation time. While there are hundreds to thousands of transistors in one circuit, using a large network for one device as its CM is not practically feasible. Therefore, with a limited number of neurons and parameters in one network for CMs, we need to ensure that the network topology can be optimized to promote the fitting to device IVs and the accuracy in circuit simulation. The importance of neuro-evolution becomes indispensable in this regard.

The genome representation and the crossover scheme are other vital aspects of neuro-evolution. A conventional way to do the GA is to encode the individual in the optimization problem in a real-value or binary array, similar to the biological genomes. In this basic method, every parameter is represented by a bit in the genome. In our case, the parameters are the neuron number, activation functions in each block, and the connections between the blocks in each hidden layer. This basic encoding scheme is simple but can be ineffective in some cases. To improve the effectiveness of GA, we use a more sophisticated representation of genomes, as described in Fig. 2 in the previous sections. With this genome representation in hand, we have to proceed to decide the crossover scheme. When the genome has some data structure, we need to consider how to effectively conduct recombination to boost the individual score. Arbitrary crossover is not practical because the genome structures are destroyed, and the crossover scheme should lead to new individuals that can provide lower MSE values. In our case, we want to ensure that the evolution does not lead to too complex and too strange connections and maintain some basic features of parent individuals. This ensures the desired features from parents are retained. For this purpose, we formulate a one-point crossover scheme where the crossover point is only between the hidden layers. The crossover point cannot be within a block or a layer. In literature, the most pronounced case where careful consideration needs to be carried out in crossover is the so-called spatial recombination [54, 55]. In their case, similar to the consideration here, the sensor array represented by a genome cannot be recombined arbitrarily in order to maintain the spatial distribution and the similarity between the parents and the offspring.

In standard neuroevolution, the GA is used to optimize the topology and tune the network weighting and biases. This is similar to reinforcement learning, where the reward at each path keeps increasing during the procedure. In this work, we do not use pure GA or a reinforcement-learning scheme. Instead, we use a GA and ADAM hybridization method. We think using a conventional gradient-based training scheme can be more effective in tuning the network weights and the biases in topology-optimized networks. On the other hand, using GA to tune the network architecture, including connections between neurons or blocks, can be effective. Essentially, the gradient-based optimization method cannot adjust the network architecture since the slope is difficult to evaluate for various network topologies. Fig. 6 shows the device I-V with fitting, and Fig. 7 draws the optimized network topologies for the case of 677 and 101 parameters, respectively. In this effort, the neuron number in each block can vary from 1 to 4-10, and thus essentially, every single neuron can have its own unique connections.

As for the question of why neuro-evolution is effective and needed in semiconductor device CMs, we think this lies in the nature of the fitting problem and the requirement of the small network size for semiconductor device CMs. In general, if the mapping between the input and output variables is highly nonlinear or complex, we need to increase the size of the network to fit the relation. It is also likely that the input and output relation is so complex that increasing the network size becomes ineffective.

![Figure 5](image-url) Figure 5. Test set scatter plot of NE and MLP models with patience=100 and with similar number of parameters.
Figure 6. The I-V response of CM using NE and MPL models for different W and L. NE and MPL models with patience=100 have a similar number of parameters of 677 and 715, respectively.

Figure 7. Graph visualization of optimized network architecture using neuro-evolution with (a) 677 and (b) 101 parameters.

This means a more complex network connection/structure needs to be used, e.g., convolution, recurrence, pooling, dropout, concatenation, etc. The pure dense connection between adjacent layers becomes less effective to fit a very complex mapping between input and output variables. Therefore, it is not surprising that we can have better fitting using optimized network topologies since an MLP, i.e., a simple dense connection, does not encompass the entire search space in network topologies. The view on the enhancement in this work can be explained as that we can explore many more topologies when using the same number of parameters. Finally, there are undoubtedly other possible, extended ways to enhance the fitting beyond the scheme used in this work. Specifically, in this effort, we do not allow connections beyond adjacent layers and do not allow recurrent/convolutional connections. While a more complex evolution scheme can further boost the efficiency of NE, fortunately, our tuning already shows pronounced
effectiveness by the comparison between NE-based architecture and MLP at a similar parameter number.

IV. CONCLUSION

This work presents an efficient way to tune the network topologies to develop an accurate and topology-optimized device CM for the MOS transistor. Using a genetic algorithm (GA) and neuro-evolution (NE) method, we have allowed the network topology to learn and adapt within the defined parameters of \( N_{\text{layers}} \), \( N_{\text{blocks}} \), \( N_{\text{neurons}} \). Moreover, while in most cases, GA is solely used in NE to implement reinforcement-learning-like optimization, here, we have used GA and ADAM together in a supervised learning setting to develop CMs for MOSFET devices. During evolution, by limiting the connections of each block in a layer to its previous layer only, we have minimized the search space of GA, which resulted in a faster locating of the optimized topologies and faster convergence in network training. Furthermore, compared to the baseline MLP model with a similar parameter number, our NE model, at patience=10 for the test dataset, has reduced RMSE of 0.1461, 0.0985, 0.1274, 0.0971, 0.0705 for 5 GA runs. We believe topology optimization is an important task for future ML-based CMs to increase the fitting accuracy in a reduced network and minimize the circuit simulation time.

REFERENCES

[1] J. Sellier, J. Fonseca, T. Kubis, M. Povolotskiy, Y. He, H. Itakihameneh, et al., “NanoE, a parallel, multiscale, multiphysics nanoelectronics modeling tool,” 2012.
[2] S. Sinha, G. Yeric, V. Chandra, B. Cline, and Y. Cao, “Exploring sub-20nm FinFET design with predictive technology models,” presented at the Proceedings of the 49th Annual Design Automation Conference, San Francisco, California, 2012.
[3] A. KhakiFrooz, O. M. Nayeb, and D. Antoniadis, “A Simple Semisempirical Short-Channel MOSFET Current–Voltage Model Continuous Across All Regions of Operation and Employing Only Physical Parameters,” IEEE Transactions on Electron Devices, vol. 56, pp. 1674-1680, 2009.
[4] H. W. Chou, H. Yun, J. S. Yoon, and R. H. Baek, “Neural Approach for Modeling and Optimizing Si-MOSFET Manufacturing,” IEEE Trans. on VLSI Syst., vol. 8, pp. 159351-159370, 2020.
[5] L. Zhang and M. Chan, “Artificial neural network design for compact modeling of generic transistors,” Journal of Computational Electronics, vol. 16, 09/01 2017.
[6] H. Hammouda, M. Mhi, Z. Gafsi, and K. Besbes, “Neural-Based Models of Semiconductor Devices for SPICE Simulator,” American Journal of Applied Sciences, vol. 5, 04/01 2008.
[7] M. Li, O. Irsoy, C. Carde, and H. G. Xing, “Physics-Inspired Neural Networks for Efficient Device Compact Modeling,” IEEE Journal on Exploratory Solid-State Computational Devices and Circuits, vol. 2, pp. 44-49, 2016.
[8] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” Neural Networks, vol. 4, pp. 251-257, 1991/01/01. 1991.
[9] L. Ding, L. Huang, S. Li, H. Gao, H. Deng, Y. Li, et al., "Definition and Application of Variable Resistance Coefficient for Wheeled Mobile Robots on Deformable Terrain," IEEE Transactions on Robotics, vol. 36, pp. 894-909, 2020.
[10] Z. Lv, D. Chen, and Q. Wang, "Diversified Technologies in Internet of Vehicles Under Intelligent Edge Computing," IEEE Transactions on Intelligent Transportation Systems, vol. 22, pp. 2048-2059, 2021.
[11] Z. Lv, R. Lou, and V. K. Singh, "AI Empowered Communication Systems for Intelligent Transportation Systems," IEEE Transactions on Intelligent Transportation Systems, vol. 22, pp. 4579-4587, 2021.
[12] G. Sun, Y. Cong, J. Dong, Y. Liu, Z. Ding, and H. Yu, “What and How: Generalized Lifelong Spectral Clustering via Dual Memory,” IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1-1, 2021.
[13] G. Sun, Y. Cong, Q. Wang, B. Zhong, and Y. Fu, “Representative Task Self-Selection for Flexible Clustered Lifelong Learning,” IEEE Transactions on Neural Networks and Learning Systems, pp. 1-15, 2020.
[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations, ed: MIT Press, 1986, pp. 318-362.
[15] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” CoRR, vol. abs/1412.6980, 2015.
[16] J. Duch, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” Journal of Machine Learning Research, vol. 12, pp. 2121-2159, 07/01 2011.
[17] Y. Nesterov, “A method for unconstrained convex minimization problem with the rate of convergence of O(1/k^2),” 1983.
[18] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, “Designing neural networks through neuroevolution,” Nature Machine Intelligence, vol. 1, pp. 23-35, 2019/01. 2019.
[19] K. O. Stanley and R. Miikkulainen, “Efficient evolution of neural network topologies,” in Proceedings of the 2002 Congress on Evolutionary Computation. CEC’02 (Cat. No.02TH8680), 2002, pp. 1757-1762 vol.2.
[20] J. Bohrer, B. Grisi, and M. Dorn, “Neuroevolution of Neural Network Architectures Using CoDeepNEAT and Keras,” ArXiv, vol. abs/2002.04634, 2020.
[21] K. Ryczko, P. Darancet, and I. Tamblyn, “Inverse Design of a Graphene-Based Quantum Transducer via Neuroevolution,” The Journal of Physical Chemistry C, vol. 124, pp. 26117-26123, 2020/12/03 2020.
[22] R. S. Sutton, R. S. A. A. G. Barto, A. G. Barto, C. D. A. L. L. A. G. Barto, B. Fach, and M. Press, Reinforcement Learning: An Introduction: MIT Press, 1998.
[23] D. Moriarty and R. Miikkulainen, “Efficient Reinforcement Learning Through Symbiotic Evolution,” Machine Learning, vol. 22, 01/05 1995.
[24] D. I. Moriarty, “Symbiotic Evolution of Neural Networks in Sequential Decision Tasks,” 1997.
[25] D. Floreano, P. Husbands, and S. Nolfi, “Evolutionary Robotics,” ed, 2008, pp. 1423-1451.
[26] K. O. Stanley and R. Miikkulainen, “Competitive coevolution through evolutionary complexification,” J. Artif. Int. Res., vol. 21, pp. 63–100, 2004.
[27] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, “Real-time neuroevolution in the NERO video game,” IEEE Transactions on Evolutionary Computation, vol. 9, pp. 653-668, 2005.
[28] S. Such, V. Madlivan, E. Conti, J. Lehman, K. Stanley, and J. Clune, “Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning,” 12/18 2017.
[29] K. Stanley, N. Kohli, R. Sherony, and R. Miikkulainen, Neuroevolution of an automobile crash warning system, 2005.
[30] T. Back, Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms: Oxford University Press, Inc., 1996.
[31] P. J. Angeline, G. M. Saunders, and J. B. Pollack, “An evolutionary algorithm that constructs recurrent neural networks,” IEEE Transactions on Neural Networks, vol. 5, pp. 54-65, 1994.
[32] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning: Addison-Wesley Longman Publishing Co., Inc., 1989.
[33] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence: The MIT Press, 1992.
[34] I. Rechenberg, “Evolution Strategy: Nature’s Way of Optimization,” Berlin, Heidelberg, 1994, pp. 106-126.
[35] R. J. David, “Artificial Intelligence through Simulated Evolution,” in Evolutionary Computation: The fossil Record, ed: IEEE, 1998, pp. 227-296.
[36] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, pp. 87-112, 1994/06/01 1994.

[37] N. Y. Nikolaev and H. Iba, "Learning polynomial feedforward neural networks by genetic programming and backpropagation," *IEEE Transactions on Neural Networks*, vol. 14, pp. 337-350, 2003.

[38] P. Robb, A. Soper, and K. Rennolls, "Use of Genetic Algorithms for Optimal Topology Determination in Back Propagation Neural Networks," 1993.

[39] X. Cui, W. Zhang, Z. Tüiske, and M. Picheny, "Evolutionary stochastic gradient descent for optimization of deep neural networks," presented at the Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2018.

[40] R. Chandra and C. Omlin, *Hybrid Evolutionary One-Step Gradient Descent for Training Recurrent Neural Networks*, 2008.

[41] Y.-W. Ho, Master of Science, Institute of Electronics Engineering, National Yang Ming Chiao Tung University, 2021.

[42] Y. Lei, X. Huo, and B. Yan, "Deep Neural Network for Device Modeling," presented at the Electron Devices Technology and Manufacturing (EDTM) Conference, Kobe, Japan, 2018.

[43] H. Habal, D. Tsonve, and M. Schweikardt, "Compact Models for Initial MOSFET Sizing Based on Higher-order Artificial Neural Networks," presented at the MLCAD '20, Virtual Event, Iceland, 2020.

[44] V. Rossum, Guido, and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[45] C. R. Harris, K. J. Millman, S. J. v. d. Walt, R. Gommers, P. Virtanen, D. Cournapeau, et al., "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2000.

[46] E. Jones, T. Oliphant, P. Peterson, and others, "SciPy: Open Source Scientific Tools for Python," 2001.

[47] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al., "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[49] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, 2010.

[50] D. Dasgupta and D. McGregor, "SGA: A Structured Genetic Algorithm," 1992.

[51] M. L. Ryerkerk, R. C. Averill, K. Deb, and E. D. Goodman, "Solving metameric variable-length optimization problems using genetic algorithms," *Genetic Programming and Evolvable Machines*, vol. 18, pp. 247-277, 2017/06/01 2017.

[52] D. E. Goldberg and K. Deb, "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," in *Foundations of Genetic Algorithms*. vol. 1, G. J. E. Rawlins, Ed., ed: Elsevier, 1991, pp. 69-93.

[53] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, 2015, pp. 515-519.

[54] D. M. Cherba and W. Punch, "Crossover gene selection by spatial location," presented at the Proceedings of the 8th annual conference on Genetic and evolutionary computation, Seattle, Washington, USA, 2006.

[55] S.-Y. Lin and H.-F. Tsai, "Micro genetic algorithm with spatial crossover and correction schemes for constrained three-dimensional reader network planning," *Expert Systems with Applications*, vol. 44, pp. 344-353, 2016/02/01/ 2016.