Deep Learning for Logic Optimization

Winston Haaswijk[†*], Edo Collins[‡*], Benoit Seguin[§*],
Mathias Soeken[†], Frédéric Kaplan[§], Sabine Süsstrunk[‡], Giovanni De Micheli[†]

[†]Integrated Systems Laboratory, EPFL, Lausanne, VD, Switzerland
[‡]Image and Visual Representation Lab, EPFL, Lausanne, VD, Switzerland
[§]Digital Humanities Laboratory, EPFL, Lausanne, VD, Switzerland

[*]These authors contributed equally to this work

***Abstract*—The slowing down of Moore's law and the emergence of new technologies puts an increasing pressure on the field of EDA in general and the logic synthesis and optimization community in particular. There is a constant need to improve optimization heuristics. However, finding and implementing such heuristics is a difficult task, especially with the novel logic primitives and potentially unconventional requirements of emerging technologies. In this paper, we show how logic optimization may be recast as a game. We then take advantage of recent advances in deep reinforcement learning to build a system that learns how to play this game. Our design has a number of desirable properties. It is autonomous because it learns automatically, and does not require handcrafted heuristics or other human intervention. It generalizes to large multi-output Boolean functions after training on small examples. Additionally, it natively supports both single- and multi-output functions, without the need to handle special cases. Finally, it is generic because the same algorithm can be used to achieve different optimization objectives, e.g., size and depth.**

I. Introduction

Over the last decades, advances in logic synthesis and optimization have been key factors in enabling technological progress. The progress of Moore's law has been aided by the data structures and algorithms developed by the logic synthesis community. In recent years, this progress has slowed down, due to limitations of CMOS technology. At the same time, emerging technologies [1]–[3] provide new opportunities but may have different characteristics (e.g., correspond to different logic primitives) than conventional approaches. Seeking alternatives, and exploiting the potential of emerging technologies, puts additional pressure on the tools of logic synthesis to adapt and improve. Methods that take advantage of the potential of new technologies can unlock significant improvements [4], [5].

Many different logic synthesis techniques have been successfully developed and applied. They range from exact to heuristic, and from Boolean to algebraic [6]–[12]. Exact methods do not scale well, meaning that heuristics are applied by all practical optimization methods. Heuristics are, almost by definition, incomplete in the sense that they do not guarantee optimal results. In fact, this was recently shown for algorithms that use state-of-the-art heuristics [13]. Moreover, discovering and implementing good heuristics is difficult and will only become more difficult with the novel logic primitives and unconventional requirements of emerging technologies. Further, different design goals, such as size and depth, require different heuristics [10], [14]. Finally, it is difficult to combine heuristics in optimization scripts. For example, in general it is not obvious in which order they should be applied. In summary, the discovery of good heuristics is vital to the continued success of logic synthesis and optimization.

This paper shows for the first time how heuristics for logic optimization can be discovered automatically through the use of machine learning. Moving away from handcrafted heuristics avoids the difficulties facing optimization algorithms described above. The heuristics found by machine learning can improve over time, by using more time and training data. Finally, as we demonstrate in this paper, the same machine learning algorithm can be used to learn heuristics corresponding to different optimization objectives.

In recent years the advance of deep learning has revolutionized machine learning. Deep learning [15] is a machine learning approach based on neural networks. Contrary to conventional neural networks, deep neural networks (DNNs) stack many hidden layers together, allowing for more complex processing of training data [16], [17]. Some of the key features of DNNs are their capability to automatically determine relevant features and build hierarchical representations given a particular problem domain. This ability of DNNs removes the need for handcrafted features. We take advantage of this ability by using DNNs to automatically find features that are useful in logic optimization.

Deep learning has been particularly successful in the context of *reinforcement learning* [18], [19]. Essentially, in reinforcement learning an agent learns how to behave with respect to some environment by taking actions and being rewarded or punished by the environment depending on how good the chosen actions are [20]. In other words, the agent learns how to interact with an environment without explicitly being told how to do so. Recently, deep reinforcement learning was used to build the AlphaGo system, which became the first computer program to defeat a human professional player in the full-sized game of Go [19].

In this paper, we show how the problem of logic optimization can be recast as a game. We use the tools of deep reinforcement learning to train a computer program to perform logic optimization. The program learns how to perform these optimizations completely autonomously, without any handcrafted heuristics.

We show how our algorithm is able to find optimum representations for all 3-input functions, reaching 100% of potential improvement. It is also able to reach 83% of potential improvement in size optimization of 4-input functions.

Additionally, we show that our model generalizes to larger functions. After training on 4-input functions, it is able to find significant optimizations in larger 6-input *disjoint subset decomposable* (DSD) functions, reaching 89.5% of potential improvement as compared to the state-of-the-art. Moreover, after preprocessing the DSD functions, we are able to improve on the state-of-the-art for 12% of DSD functions. A case study of an MCNC benchmark shows that it can also be applied to realistic circuits as it attains 86% of improvement compared to the state-of-the-art. Finally, our algorithm is a generic optimization method. We show that it is capable of performing depth optimization, obtaining 92.6% of potential improvement in depth optimization of 3-input functions. Further, the MCNC case study shows that we are able to unlock significant depth improvements over the academic state-of-the-art, ranging from 12.5% to 47.4%,.

II. Background

This paper relies on concepts from different fields. Our goal in this section is to provide the conceptual context to understand the rest of the paper. Pointers to relevant literature are provided for the interested reader. We do assume some familiarity with concepts from logic synthesis, such as the notion of a logic network.

A. Majority Algebra and MIGs

The Boolean majority operator

$$\langle xyz \rangle = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$$

has many interesting properties. Knuth calls it "probably the most important operator in the universe," [21]. One property is that it can be used to define a **sound** and **complete** Boolean algebra as follows (due to [14]):

$$\Omega = \begin{cases} \textbf{Commutativity} - \Omega.C \\ \langle xyz \rangle = \langle yxz \rangle = \langle zyx \rangle \\ \textbf{Majority} - \Omega.M \\ \begin{cases} \langle xxz \rangle = x \\ \langle x\bar{x}z \rangle = z \end{cases} \\ \textbf{Associativity} - \Omega.A \\ \langle xu\langle yuz \rangle \rangle = \langle zu\langle yux \rangle \rangle \\ \textbf{Distributivity} - \Omega.D \\ \langle xy\langle uvz \rangle \rangle = \langle \langle xyu \rangle \langle xyv \rangle z \rangle \\ \textbf{Inverter Propagation} - \Omega.I \\ \overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle \end{cases}$$

Soundness means that, given an expression e from the majority algebra, the application of any of the rules that apply to that expression results in an equivalent expression e'. For example, suppose that $e = \langle x_1 \bar{x}_1 x_2 \rangle$. We can apply the majority axiom to obtain the equivalent expression $e' = x_2$. Completeness means that, given any two equivalent expressions e and e', there exists some sequence of rules from the algebra that can be used to transform e into e', or vice versa.

The Majority Inverter Graph (MIG) is a data structure that corresponds closely to the majority algebra [14], [22]. An MIG is a *directed acyclic graph* (DAG) in which every node (vertex) corresponds to a majority operator. Edges have an optional complementation attribute that indicates inversion. Because the majority operator includes the AND and OR operators, MIGs include data structures such as *And-Inverter Graphs* (AIGs). Due to this inclusion and the addition of the expressive majority operator, MIGs are a more compact logic representation than such conventional structures. Fig. 1 shows an example of an MIG, comparing it to an equivalent AIG.

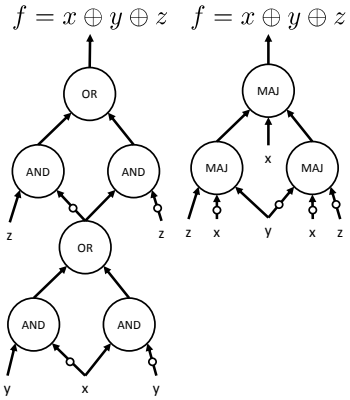

Fig. 1: A side-by-side comparison of an equivalent AIG and MIG. The AIG on the left, and the MIG on the right both compute the function $f = x \oplus y \oplus z$. Inversion is indicated by bubbles on the edges. The MIG represents the function more compactly due to the expressiveness of the majority operator.

MIGs correspond closely to the majority algebra. We can manipulate them using the rules of this algebra, providing a way of operating on MIGs that is sound and complete. This property is desirable for logic optimization. When we use MIGs as our logic representation we can reach any point in the design space using the rules from the algebra. This makes them a convenient representation for logic networks in the context of logic optimization. Algebraic and Boolean manipulation of MIGs has been shown to unlock significant improvements over the state-of-the-art in logic optimization [14], [22].

B. Deep Learning

With ever growing data sets and increasing computational power, the last decade has seen a dramatic rise in the popularity of deep learning approaches. These methods achieve state-of-the-art results on benchmarks from various fields, such as computer vision, natural language processing, speech recognition, genomics and many others [15].

The success of these models lies in their ability to learn powerful hierarchical representations directly from the raw data. These representations naturally encode domain-specific invariants. Thus, such transformations effectively linearize complex problems such as classification.

Of particular interest is the class of convolutional neural networks (CNNs) [16] for image inputs. At every layer, these models use shift-invariant filters to perform convolution, followed by a non-linear operation such as rectified-linearity (ReLU) [17]. The weight sharing allowed by convolution, as

well as the gradient properties of the ReLU, allow for very deep models to be trained with gradient descent.

In this paper we use a generalization of CNNs aimed at processing graph input, as described in Section IV-B.

C. Reinforcement Learning

Neural networks are usually trained under the paradigm of supervised learning, i.e., on input-output pairs from some ground-truth data set. A different paradigm is that of reinforcement learning (RL), where an agent is not told what action it should take, but instead receives a reward or penalty for actions. Rewards and penalties are dictated by an external environment. The goal of RL is for the agent to learn a policy (strategy) that maximizes its reward. Although it has many applications, reinforcement learning is often applied in the context of game playing, going back as far as 1959 [23].

Recent advances in deep learning have had a substantial impact on RL, resulting in the emerging sub-field of deep RL. In this context DNNs are used to approximate functions which assign a score to each possible action. The agent uses these scores to select which actions to take.

A recent example of this ability has been demonstrated in [18], where the authors trained a DNN to play 49 different Atari games from pixel input alone. With this technique they achieved comparable performance with human experts across all games. Building on this success, a deep RL agent was recently shown to have attained superhuman performance in the challenging game of Go [19].

III. Logic Optimization as a Game

In logic optimization we are given an input logic network N, and are asked to produce an equivalent logic network N' that is better than the input network. Which networks are considered better depends on the optimization objective.

We can cast the process of optimization as a single-player game as follows. The game consists of states and moves that transition from one state to the next. States in the logic optimization game correspond to logic networks. Moves in the game correspond to operations on these networks. We say that a move is *legal* in a state s if applying it to s results in an state s', where s and s' are logic networks with equivalent I/O behavior. As states s and s' correspond to logic networks, they are equivalent *if and only if* the logic networks s and s' are equivalent. We define $\mathrm{moves}(s)$ to be the set of moves that are legal in a state s: $\mathrm{moves}(s) = \{a \mid a \text{ is legal in } s\}$. Which moves are legal depends on the type of logic network and the type of operations we want to enable.

We write $s \xrightarrow{a} s'$ to mean that applying move $a \in \mathrm{moves}(s)$ to state s results in state s'. For every state, we define a special move a_ϵ that corresponds to the identity function, such that, $s \xrightarrow{a_\epsilon} s$. This means that the set of legal moves is never empty: we can always apply the identity move. The game can now be played by applying legal moves to states. A game can then be characterized by an initial state s_0, an output state s_n, and a sequence of n moves

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots s_{n-1} \xrightarrow{a_n} s_n$$

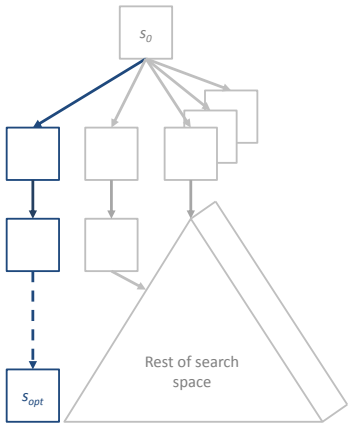

Fig. 2: An optimization game decision tree of height n and breadth b, rooted at initial state s_0. The highlighted path indicates an optimum sequence of moves.

Given a sequence $\mathrm{seq} = (a_1, \ldots, a_n)$ we write

$$s_0 \overset{\mathrm{seq}}{\Longrightarrow} s_n \Leftrightarrow s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$$

We use $|\,\mathrm{seq}\,|$ to denote the length of a sequence.

Finally, we have a function $\mathrm{score}(s)$ that indicates how good a state s is with respect to the optimization criterion. With these definitions, we are now ready to propose the following game: given an input state s_0, find a sequence of moves seq such that $|\,\mathrm{seq}\,| = n$, and for all sequences seq', $|\,\mathrm{seq}'\,| = n$:

$$s_0 \overset{seq}{\Longrightarrow} s_n \wedge s_0 \overset{seq'}{\Longrightarrow} s'_n \Rightarrow \mathrm{score}(s_n) \geq \mathrm{score}(s'_n) \quad (1)$$

We call these kinds of games, defined for fixed n, *optimization games*. Such games correspond to generic optimization procedures. For example, suppose we would like to perform size optimization. In that case, $\mathrm{score}(s)$ could return the reciprocal of the size of a logic network. This corresponds to the objective of finding the minimum size network achievable within n moves. Other optimization objectives can be defined analogously. We say that an optimization game is *solved* or *won* when we have found a sequence of n moves that results in a maximum score.

An n-step optimization game is a game of perfect information. It gives rise to a decision tree, rooted at the initial state s_0. Fig. 2 shows what such a decision tree looks like. It contains approximately b^n sequences of moves, where b is the game's *breadth* (approximate number of legal moves per state) [19].

A. MIGs as Game States

MIGs are a natural state representation for logic optimization through game playing. They can represent arbitrary combinational logic networks. Moreover, we can compute a set of legal moves for an MIG directly from the majority algebra Ω. Finally, we know that the majority algebra is sound and complete. Therefore, the sets of legal moves are also sound and complete. Any transformation between two equivalent MIGs that is achievable by a sequence of n applications of the majority axioms can be achieved by playing a game of n moves on MIGs.

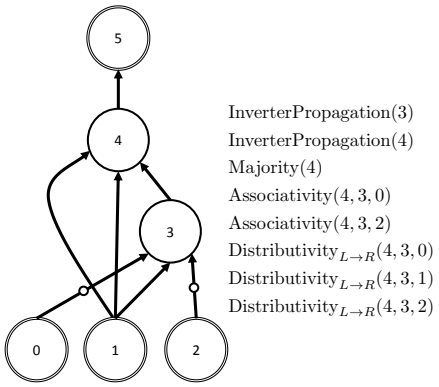

Fig. 3: An MIG and its associated set of legal moves. The inputs are indicated by double-lined circles, as is the output node which is explicitly shown here.

An axiom that applies to an MIG induces a legal move. For example, suppose that an MIG M contains a node v with 2 equal fanins. The majority axiom applies at that node. Hence, we can say that the move "Apply the majority axiom at node v" is legal. This notion can be expressed as $\mathrm{Majority}(v) \in \mathrm{moves}(M)$. Thus, moves can be represented by simple structures that refer to axioms and nodes. To compute the complete set of moves, we then traverse M and collect for each node the moves induced by the axioms that apply at that node. As the majority axioms are equalities, moves may be applied in two directions: the left-hand side of an equality may be rewritten to the right-hand side, and vice versa. For example, the distributivity axiom induces two move types: $\mathrm{Distributivity}_{L\to R}$, and $\mathrm{Distributivity}_{R\to L}$. These correspond to rewriting majority algebra equations from left to right and from right to left, respectively.

Fig. 3 shows an example of an MIG corresponding to a state of the game, as well as the set of moves that are legal in that state. Moves such as $\mathrm{Majority}$ are easy to detect. Other moves, such as $\mathrm{Associativity}$ may require closer inspection of the graph.

In our experiments, we added a number of derived moves from the majority algebra axioms. Adding these moves ($\mathrm{Relevance}$, $\mathrm{Complementary\_Associativity}$, and $\mathrm{Substitution}$) is helpful because it allows us to find shorter sequences to optimal MIGs. Their derivation can be found in [14].

B. Playing Optimization Games

So far, we have seen how logic optimization may be viewed as a perfect information optimization game. These games can be used to optimize a wide variety of objectives. Moreover, they may always be won through exhaustive search. Obviously, such a strategy is in general infeasible, as the size of the search space grows exponentially with the number of moves. Hence, we need heuristics to prune the search tree. In other words, we want to develop strategies (heuristics) for playing the game that are less computationally expensive than exhaustive search, but that still lead to good results.

The generic nature of optimization games is a desirable property. It makes specifying potentially exotic optimization objectives straightforward. A drawback is that developing heuristics for optimization in this setting is a nontrivial task, as we wish to perform well on a wide variety of games. One way to approach this is to define new heuristics for different score functions. However, this means that we have to develop new strategies every time our objectives change. Another approach might be to develop very generic heuristics. This is a difficult task, and has the added drawback that specialized strategies are likely to outperform a generic strategy in their specific domains. Therefore, we propose the use of machine learning to discover these heuristics in a flexible and automated way.

C. Global Optimality

As a consequence of the soundness and completeness of the majority algebra, there always exists a sequence seq, such that $s_0 \overset{\mathrm{seq}}{\Longrightarrow} s_{\mathrm{opt}}$, where s_{opt} is some globally optimum state according to $\mathrm{score}(s_{\mathrm{opt}})$. However, because $|\,\mathrm{seq}\,|$ may be much larger than n, even if we play the optimization game optimally, we are not guaranteed to find s_{opt}. Moreover, depending on the particular score function it may be computationally expensive, or impossible, to know whether any state is a global optimum. For example, we may want to know if an MIG is a global size optimum. This involves solving the *Minimum Circuit Size Problem* (MCSP), which is conjectured to be intractable [24].

IV. Applying Deep Reinforcement Learning

Given an MIG state at step t, s_t, a move a_t leads to a new state s_{t+1}. This transition is associated with a *reward* function $r(s_t, a_t)$, which is defined with respect to the optimization objective. We can define this reward with respect to the score function as $r(s_t, a_t) = \mathrm{score}(s_{t+1}) - \mathrm{score}(s_t)$. As shorthand we write $r_t = r(s_t, a_t)$.

Summing the rewards obtained during a sequence results in a telescoping sum:

$$\begin{aligned}\sum_{t=0}^{n} r_t &= \sum_{t}^{n-1} \mathrm{score}(s_{t+1}) - \mathrm{score}(s_t) \\ &= \mathrm{score}(s_n) - \mathrm{score}(s_0)\end{aligned}$$

Since the score of the initial state s_0 is constant for all sequences starting from it, we see that satisfying the optimality criterion in (1) corresponds to maximizing the expectation of the above sum for every initial state.

To capture the notion that immediate rewards are generally preferred to later ones, especially given a budget of n moves, we follow the practice of discounting rewards over time. We do this by introducing a *time discount factor*, $0 < \lambda \leq 1$. Our optimization objective becomes:

$$\underset{\pi}{\mathrm{argmax}}\, \mathbb{E}_\pi \left[\sum_{k=t}^{n} \lambda^k r_k \right] \tag{2}$$

The expression being optimized in known as the *expected return*, and the expectation is taken with respect to π, known as the *policy*. The policy is a function which given the state, assigns every possible move a probability, and is parameterized

by θ. In other words, $\pi_\theta(a_t|s_t)$ is the probability of applying move a_t when the state s_t is observed. Consequently, finding an optimal policy π translates to finding an optimal set of parameters θ.

A. Learning Strategy

While some methods learn the policy indirectly, in this work we chose to learn the distribution directly with the technique of *Policy Gradient* (PG) [25]. The PG parameter update is given by:

$$\theta_{t+1} = \theta_t + \alpha \left(r_t \nabla \pi_\theta(a_t|s_t) \right) \tag{3}$$

where $\alpha \in \mathbb{R}$ is the learning rate ($\alpha > 0$). This update embodies the intuition that a positively-rewarded move should be positively reinforced—by following the gradient of the move's log-probability under the current policy—and inversely negatively-rewarded moves should be discouraged.

B. Deep Neural Network Model

We model our policy function as a deep neural network, based on Graph Convolution Networks (GCN) [26]. For every node in the MIG we would like to derive node features, based on a local neighborhood, that capture information that is useful towards predicting which move is beneficial to apply. We do derive these features with a series of L transformation, corresponding to as many graph convolution layers.

The ℓth layer in our network maintains an $p \times d_\ell$ matrix of node features. Every row of the matrix represents one of the p nodes in the MIG as a d-dimensional feature vector. The ℓth layer transforms its input feature matrix as follows:

$$F^{(\ell+1)} = \sigma(AF^{(\ell)}W^{(\ell)}) \tag{4}$$

where A is the graph adjacency matrix and σ is an element-wise nonlinearity, in our case ReLU. Using the adjacency matrix here results propagation of information between neighboring nodes, meaning that concatenating L layers has the effect of expanding the neighborhood from which the final node features are derived to include Lth degree neighbors. We call L the size of the *receptive field*.

We initialize the first feature matrix with p-dimensional one-hot vector representations for every node in the graph, i.e., $d_0 = p$. This yields the identity matrix, i.e., $F^{(0)} = I$.

After propagating through all CGN layers, we use two move-dependent fully-connected layers with a final normalization layer to obtain probabilities for every available move.

C. Training Procedure

The set of policy parameters θ corresponds to the parameters of the DNN described above. Initially θ is set to random values, and it is through playing the game that the parameters are updated towards optimality. Training thus proceeds by performing sequences of n moves, called *rollouts*. For each move in a rollout we calculate its base reward with the reward function, and its associated discounted return.

We sample initial states from which to begin rollouts from a designated set of initial states. This set is manually initialized with a seed of MIGs, e.g. MIGs corresponding to the Shannon-decomposition of all 3-input Boolean functions.

At every iteration, we augment the set of initial states by randomly sampling MIGs from the rollout history, such that future rollouts may start at a newly-derived state.

V. Experiments

We start our experiments in Section V-A by training a model to perform size optimization of small functions. In Section V-B and Section V-D we show the potential for our algorithm to generalize and scale by applying it to a set of DSD functions and a circuit from the MCNC benchmark suite, respectively. Finally, we demonstrate the generic nature of our algorithm by showing that it can also be used for depth optimization in Section V-C.

We perform all experiments using a single neural network architecture, consisting of 4 GCN layers followed by 2 fully-connected layers. The results of the experiments are summarized in Table I and Table II.

A. Size Optimization

The data set $D = \{(x_1, y_1), \ldots, (x_{256}, y_{256})\}$ consists of 256 tuples (x_i, y_i), where x_i is the MIG corresponding to the Shannon decomposition of the i-th 3-input function, and y_i is the optimum MIG representation of that function. We generate the size optimum MIGs using the CirKit logic synthesis package [27], [28]. We initialize the training set $X = \{x_1, \ldots, x_{256}\}$, so that we start training on the Shannon decompositions. The optima $Y = \{y_1, \ldots, y_{256}\}$ are used *only for evaluation purposes*. We set $n = 5$ and perform 100 iterations over the training set, augmenting it in every iteration as described in Section IV-C. After training, we use the model to perform 20 steps of inference on each of the x_i, obtaining the MIG $\hat{y}_i$, and compare the results to the optima y_i. We find that in every case the model is able to achieve optimum size, i.e., $\mathrm{score}(\hat{y}_i) = \mathrm{score}(y_i)$ for all i $(1 \leq i \leq 256)$. This confirms that the model and training procedure are able to learn the strategy required to perform size optimization on MIGs.

We perform inference using an NVIDIA GeForce GTX TITAN X GPU. The run time of one inference step on a 3-input graph is approximately 5 ms, making total inference run time $20 \times 5 = 100$ms. In practice we perform inference in parallel on batches of 50 graphs at a time, making the average total inference time 2ms per graph. Inference run times of the experiments below are similar, scaled polynomially with the size of the input graphs.

Note that the model finds all optima, despite never being presented with them, by simply playing the optimization game. Interestingly, it finds all optima within at most 8 moves from the initial Shannon decomposition. Fig. 4 shows an example of such a model inference, demonstrating how the model moves from an initial decomposition to a size optimum MIG representation.

At this point it becomes useful to introduce the notion of *potential improvement*. For any pair $(x, y) \in D$, the maximum potential size improvement that can be made from a Shannon decomposition x to an MIG optimum is $\mathrm{score}(x) - \mathrm{score}(y)$. The *total potential improvement* for D is $\sum_i (\mathrm{score}(x_i) - \mathrm{score}(y_i))$. Since the model finds the optima for all 3-input functions, we say that it reaches 100% of potential improvement.

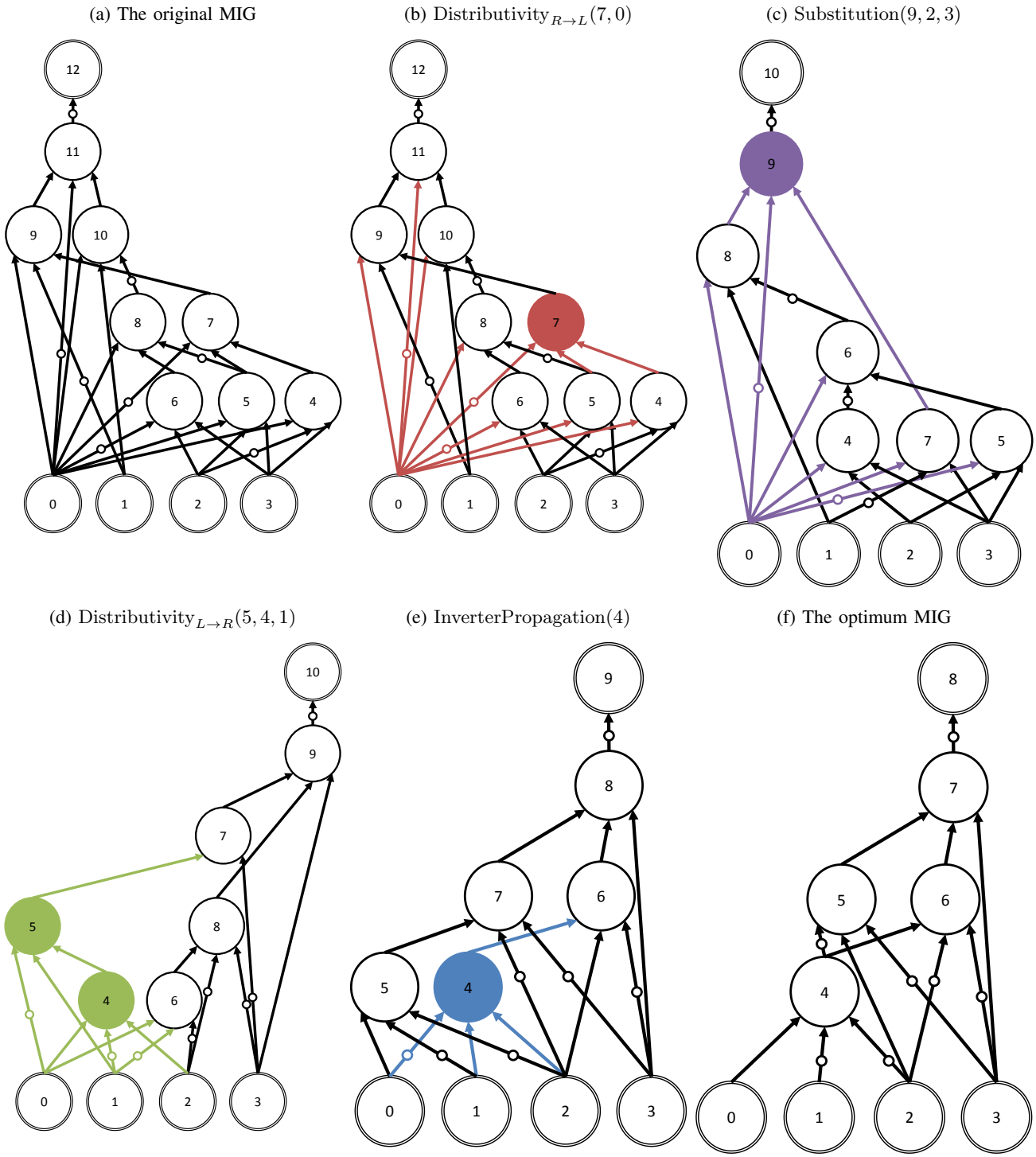

Fig. 4: From top-to-bottom, left-to-right, a sequence of 4 moves found by our model that optimizes an MIG from a Shannon decomposition to its optimum MIG representation.

TABLE I: A summary of the experimental results. The conventional run time column refers to the run time of the optimization algorithm to which we compare for each benchmark (ie. exact synthesis run time or resyn2 run time in the case of C1355). Inference run time shows the corresponding inference run time used by our neural network model obtain its results. In the (PP) experiment we apply our algorithm after pre-processing with resyn2. In one case our model achieves potential improvement above 100%. This means that it finds additional improvement as compared to resyn2.

Experiment	Optimization Objective	Potential Improvement %	Conventional Synthesis Run Time (s)	Inference Run Time (s)
3-input functions	SIZE	100.00%	0.047	0.010
3-input functions	DEPTH	92.60%	0.047	0.010
4-input functions	SIZE	83.00%	1.002	0.026
6-input DSD functions	SIZE	89.50%	0.053	0.090
6-input DSD functions (PP)	SIZE	101.60%	0.053	0.090

Next, we run the same experiment for all 4-input functions. The data set now consists of all 65536 4-input functions, i.e., $D = \{(x_1, y_1), \ldots, (x_{65536}, y_{65536})\}$. We run the training procedure to convergence. We find that the model is able to find the global size optima for 24% of the functions. For other functions it does not reach the full optima within 20 inference steps. However, it reaches 83% of total potential improvement. This implies that for most functions the model is nearly optimal. Future work will explore the effect of different network architectures and more sophisticated RL training algorithms.

B. Generalization

The experiments in Section V-A show that our model and training procedure perform well on size optimization. However, in those experiments evaluation is done on the training set, i.e., the model is trained to optimize 3- and 4-input functions and its size optimization performance is evaluated the same functions. In this section we present an experiment to test the capacity of our model to generalize. Intuitively, this capacity is important because the space of possible MIGs is infinite, even for the small number of 3-input functions. As such, training cannot possibly explore all possible states that the model may encounter. Therefore, the model should be able to pick good moves even when confronted with previously unseen states. There is another useful aspect to generalization. In theory, we could always train a model specifically on the benchmarks that we wish to optimize. However, training is a potentially expensive operation. Thus, in order to save time, we may want to reuse previously trained models.

In this experiment we use the trained 4-input model from Section V-A to perform inference on a large set containing 40,195 6-input (DSD) functions [29]. We now have $D = \{(x_1, y'_1), \ldots, (x_{40,195}, y'_{40,195})\}$. Due to the hardness of the MCSP, for this experiment it becomes computationally infeasible to find the optimum y_i. Therefore, we use the resyn2 command from the state-of-the-art ABC logic synthesis package to obtain heuristic optima [30]. This resyn2 command in ABC is an efficient high-effort optimization script that combines several nontrivial logic optimization heuristics and algorithms. We now compute potential improvement with respect to these strong heuristic optima.

The average size of the Shannon decompositions is 25.636 MIG nodes. Using the model trained on 4-input functions, we perform 100 inference steps to reach an average size of 13.826 nodes, improving average MIG size by approximately 46%. The average size of the resyn2 heuristic optima is 12.442. This means that by generalizing the models to previously unseen MIGs, we are still able to obtain 89.5% of potential improvement. Interestingly, our model is able to improve beyond the optima found by resyn2 for 5% of the graphs.

Next, we examine if our model is able to go further beyond the heuristic resyn2 results, by applying inference directly to those results instead of the Shannon decompositions. In this case, our average size improves to 12.243 nodes. The model is able to improve on resyn2 for approximately 12% of functions. These improvements range between 1 and 9 nodes of improvement.

C. Depth Optimization

This experiment explores the generic nature of our optimization algorithm, focusing on depth- instead of size optimization. In this experiment, we again look at all 3-input functions. We have $D = \{(x_1, y_1), \ldots, (x_{256}, y_{256})\}$, where the y_i now correspond to depth-optimum MIGs. Training proceeds analogously to the 3-input size optimization experiment of Section V-A. After training, we perform 20 inference steps of inference on all x_i. Within 20 steps the model is able to find depth optima for 87.5% of the functions and obtains 92.6% of the total potential improvement.

For this experiment it is crucial to note that depth is a global feature of MIGs. In this experiment our node representations consist of features derived from a local receptive field. As such, it is likely that they do not contain global depth information. In Section V-D we add critical path features in order to perform depth optimization for a number of circuits from the MCNC benchmark suite.

D. MCNC Case Study

The previous experiments focus on the optimization of relatively small functions. The experiments in this section explore the potential of our approach to scale to more realistic benchmarks. We select a subset of circuits from the MCNC benchmark suite and train our model to optimize them. We select this subset due to memory limitations of current implementation, preventing us from working on some of larger MCNC functions. However, this is strictly a technicality of our current implementation, and not a fundamental limitation of our approach.

TABLE II: Depth optimization on three circuits of the MCNC benchmark suite. Our trained neural networks model (DNN) outperforms resyn2 (ABC) in all of these cases.

Benchmark	I/O	Initial depth	ABC	DNN	Improvement
b9	41/21	10	8	7	12.5%
count	35/16	20	19	10	47.4%
my_adder	33/17	49	49	42	14.3%

We experiment with both size and depth optimization. In the first experiment we perform size optimization on the C1355 benchmark. We train the model for 250 iterations on this single circuit. Using 200 inference steps the model is able to reduce the size of the circuit by 98 nodes, reducing the circuit from its original size of 504 nodes down to 406 nodes. Analogous to Section V-B, we use resyn2 as the heuristic optimal. It reduces the size of C1355 to 390 nodes. In other words, our model is able to obtain 86% of potential improvement as compared to the state-of-the-art.

In the second experiment we select 3 other benchmarks and train a DNN model to perform depth optimization on them. As noted in Section V-C, depth optimization requires non-local features. We therefore add critical path features to the node representations. The one-hot representation of each node is augmented with a single binary feature indicating presence on the critical path. After training, we use 50 inference steps to optimize depth. The results are summarized in Table II. We obtain significant improvements compared to resyn2, ranging from 12.5% to 47.5%, with an average of 24.7%.

VI. Conclusions and Future Work

Our algorithm achieves 100% of potential size improvement for 3-input functions, 83% for 4-input functions, and 89.9% for a large set of DSD functions. After preprocessing, it is able to improve on the state-of-the-art for 12% of DSD functions, removing between 1 and 9 additional nodes. A case study of an MCNC benchmark shows that we are able to obtain 86% of potential improvement, demonstrating its application to realistic circuits. Finally, we show that our optimization algorithm is generic by applying it to depth optimization. We find 89% of the depth-optimum MIGs for all 3-input functions within 20 steps of inference, and obtain 92.6% of potential improvement. Applying depth optimization to the MCNC benchmark circuits in our case study, we improve upon the state-of-the-art, with improvements ranging from 12.5% to 47.4%.

This paper shows for the first time the applicability of machine learning to logic synthesis. We demonstrate that it is possible to create an automatic and generic optimization procedure that obtains results that are close to—and even surpass—those of heavily specialized state-of-the-art algorithms.

References

[1] I. Amlani, A. O. Orlov, G. Toth, G. H. Bernstein, C. S. Lent, and G. L. Snider, "Digital logic gate using quantum-dot cellular automata," *Science*, vol. 284, no. 5412, pp. 289–291, 1999.

[2] T. Schneider, A. A. Serga, B. Leven, B. Hillebrands, R. L. Stamps, and M. P. Kostylev, "Realization of spin-wave logic gates," *Applied Physics Letters*, vol. 92, no. 2, p. 022505, 2008.

[3] O. Y. Loh and H. D. Espinosa, "Nanoelectromechanical contact switches," *Nature Nanotechnology*, vol. 7, no. 5, pp. 283–295, 2012.

[4] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Biconditional BDD: A Novel Canonical BDD for Logic Synthesis Targeting XOR-rich Circuits," in *DATE*, 2013, pp. 1014–1017.

[5] W. Haaswijk, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "NEM Relay Design with Biconditional Binary Decision Diagrams," in *NANOARCH*, 2015.

[6] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Dag-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[7] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[8] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *Int'l Conf. on Computer Design*, 2011, pp. 429–430.

[9] W. Yang, L. Wang, and A. Mishchenko, "Lazy Man's Logic Synthesis," in *IWLS*, 2012.

[10] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A Novel Basis for Logic Rewriting," in *ASPDAC*, 2017.

[11] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[13] L. Amarú, M. Soeken, W. Haaswijk, E. Testa, P. Vuillod, J. Luo, P.-E. Gaillardon, and G. De Micheli, "Multi-level logic benchmarks: An exactness study," in *ASPDAC*, 2017.

[14] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[15] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[16] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

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

[19] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[20] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[21] D. E. Knuth, *The Art of Computer Programming*. Upper Saddle River, New Jersey: Addison-Wesley, 2011, vol. 4A.

[22] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Boolean logic optimization in majority-inverter graphs," in *Design Automation Conference*, 2015, pp. 1–6.

[23] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[24] C. D. Murray and R. R. Williams, "On the (non) NP-hardness of computing circuit complexity," in *Conference on Computational Complexity*, 2015, pp. 365–380.

[25] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPS*, vol. 99, 1999, pp. 1057–1063.

[26] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[27] M. Soeken, "CirKit," https://github.com/msoeken/cirkit.

[28] M. Soeken, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017, accepted for publication.

[29] A. Mishchenko, "An Approach to Disjoint-Support Decomposition of Logic Functions," Portland State University, Tech. Rep., 2001.

[30] A. Mishchenko, "ABC," https://bitbucket.org/alanmi/abc.