Deep Learning for Logic Optimization Algorithms

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. There is a constant need to improve optimization algorithms. However, finding and implementing such algorithms is a difficult task, especially with the novel logic primitives and potentially unconventional requirements of emerging technologies. In this paper, we cast logic optimization as a deterministic *Markov decision process* (MDP). We then take advantage of recent advances in deep reinforcement learning to build a system that learns how to navigate this process. Our design has a number of desirable properties. It is autonomous because it learns automatically and does not require human intervention. It generalizes to large functions after training on small examples. Additionally, it intrinsically 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

In this paper we show how logic optimization algorithms can be discovered automatically through the use of deep learning. Deep learning is a machine learning approach based on neural networks [1], [2]. In recent years the advance of deep learning has revolutionized machine learning. Contrary to conventional neural networks, deep neural networks (DNNs) stack many hidden layers together, allowing for more complex processing of training data [3]–[5]. 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. Recently, deep learning has been particularly successful in the context of *reinforcement learning* [6], [7].

Our deep reinforcement learning 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 unlock significant depth improvements over the academic state-of-the-art, ranging from 12.5% to 47.4%.

II. Background

A. 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 [2].

Of particular interest is the class of convolutional neural networks (CNNs) [3] 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) [5]. 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.

B. 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.

Recent advances in deep learning have had a substantial impact on RL, resulting in the emerging sub-field of deep RL [6], [7]. 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.

III. Logic Optimization as an MDP

We can cast the process of logic network optimization as a deterministic Markov decision process (MDP). Such a process can also be thought of as a single-player game of perfect information. The game consists of states and moves that

transition from one state to the next. Game states 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. We define $\text{moves}(s)$ to be the set of moves that are legal in a state s: $\text{moves}(s) = \{a \mid a \text{ is legal in } s\}$. Which moves are legal depends on the type of logic network and the type of operations we want to enable. In our work, we use the *majority inverter graphs* (MIGs) as logic network data structures [8]. MIGs correspond closely to a median (ternary majority) algebra, which allows us to define a sound and complete move set. Such a move set allows us to reach any point in the design space.

We write $s \xrightarrow{a} s'$ to mean that applying move $a \in \text{moves}(s)$ 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

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

Finally, we have a function $\text{score}(s)$ that indicates how good a state s is with respect to the optimization criterion. The procedure outlined here corresponds to a generic optimization procedure. For example, suppose we would like to perform size optimization. In that case, $\text{score}(s)$ could return the reciprocal of the size of a logic network. This corresponds to a game in which the objective is to find the minimum size network achievable within n moves. Other optimization objectives are defined analogously.

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 $r(s_t, a_t) = \text{score}(s_{t+1}) - \text{score}(s_t)$. As shorthand we write $r_t = r(s_t, a_t)$.

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

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

Since the score of the initial state s_0 is constant for all sequences starting from it, satisfying the optimality criterion 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:

$$\underset{\pi}{\text{argmax}}\, \mathbb{E}_\pi \left[\sum_{k=t}^{n} \lambda^k r_k \right] \tag{1}$$

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) [9]. The PG parameter update is given by:

$$\theta_{t+1} = \theta_t + \alpha \left(r_t \nabla \pi_\theta(a_t|s_t) \right) \tag{2}$$

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) [10]. 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 a $p \times d_\ell$ 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:

$$F^{(\ell+1)} = \sigma(AF^{(\ell)}W^{(\ell)}) \tag{3}$$

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 show 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 [11], [12]. 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.

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.

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.

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 on the same functions. In this section we present an experiment to verify the capacity of our model to generalize.

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. 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 $\mathrm{resyn2}$ command from the state-of-the-art ABC logic synthesis package to obtain heuristic optima [13]. 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 $\mathrm{resyn2}$ 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 $\mathrm{resyn2}$ for 5% of the graphs.

Next, we examine if our model is able to go further beyond the heuristic $\mathrm{resyn2}$ 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 $\mathrm{resyn2}$ 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

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

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, that depth is a global feature of MIGs. Therefore, 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. The one-hot representation of each node is augmented with a single binary feature indicating presence on the critical path.

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.

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. 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%.

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%

VI. Conclusions

We show how machine learning can be applied to algebraic logic optimization, and the feasibility of creating an automatic and generic optimization procedure that obtains results close to—and even surpassing—those of heavily specialized state-of-the-art algorithms. Moore's law is slowing down, complex new technologies are emerging, and the pressure on EDA tools is increasing. Going forward, algorithms that quickly and autonomously learn new optimizations and heuristics will be invaluable tools to drive progress in electronics and EDA. In this paper, we take a first step in that direction.

Acknowledgments

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty).

References

[1] L. Chua and T. Roska, "The CNN Paradigm," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 40, no. 3, pp. 147–156, 1993.

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

[3] 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.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] M. Soeken, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on CAD*, 2017.

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