RL-Guided Runtime-Constrained Heuristic Exploration for Logic Synthesis

Yasasvi V. Peruvemba†, Shubham Rai‡, Kapil Ahuja†, Akash Kumar‡
†Computer Science and Engineering, IIT Indore, India ‡ Chair of Processor Design, TU Dresden, Germany
†{ee170002061, kahuja}@iiti.ac.in ‡{shubham.rai, akash.kumar}@tu-dresden.de

***Abstract*—Within logic synthesis, most optimization scripts are well-defined heuristics that generalize over a variety of Boolean circuits. These heuristic-based scripts comprise various optimization algorithms which are applied sequentially in a specific order over a logic graph representation of Boolean circuits (typically in the form of *And Inverter Graphs* (AIGs) or *Majority Inverter Graphs* (MIGs)). These heuristics, despite being well-defined generalizations, may not perform well over all kinds of circuits. In order to develop custom heuristics specific to a particular Boolean circuit that performs well, we propose a runtime-constrained reinforcement learning (RL) approach which is able to generate scripts to carry out logic synthesis flows. Within our approach, we incorporate a graph convolution network (GCN) in order to perform a holistic exploration of the search space. To carry out an extensive evaluation, we identify three different classes of environments consisting of different baseline optimization sequences. The experimental results reveal that our model outperforms the prevalent state-of-the-art work [24] and the best heuristic-based scripts of Berkeley-ABC [4]. Our evaluations show that our framework provides up to an average of 8.3% further reduction in level over the EPFL Benchmark Suite [8] as compared to the Berkeley-ABC scripts. Further, we develop a framework for the EPFL mockturtle [20] logic synthesis libraries and generate custom scripts using our RL-based approach.**

***Index Terms*—Reinforcement Learning, Graph Convolutional Network, Logic Synthesis, Majority-Inverter Graph**

I. Introduction

Logic synthesis deals with optimizing and minimizing Boolean circuits, with the major objective being either to reduce the overall size, depth or power of the circuit. Various logic representations such as *And-Inverter Graphs* (AIGs) [5] or *Majority-Inverter Graphs* (MIGs) [9] have been proposed which offer compact yet multi-level logic representations which enable better runtimes for logic synthesis flows [5, 13]. Using Boolean and algebraic methods, several algorithms such as *balancing, refactoring, resubstitution, rewriting* have been developed over these multi-level logic representations which can be sequentially applied over any given logic network to achieve size and depth reduction. Hence, several heuristic-based general purpose scripts like *resyn, compress2rs* exist within the state-of-the-art academic tool, Berkeley-ABC [4] which employs a fixed sequential set of optimizations that performs well over a variety of logic circuits.

However, logic synthesis being intractable [1], these powerful optimization scripts may or may not provide improvement over all circuits. For a given Boolean circuit, the final representational features, i.e the number of nodes and the depth of the logic network, may change vastly with the application of various permutations of such optimization algorithms.

Therefore, rather than trying to enumerate the vast possibilities of different permutations of optimization algorithms, we aim to use a reinforcement learning (RL) agent in order to determine an appropriate sequential order of algorithms to be applied over any logic network. Our approach enables a generic and user-defined environment which can be used to target specific optimization goals during the inference phase.

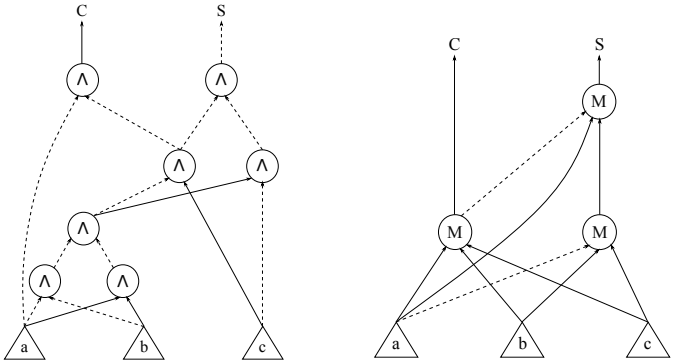
Various works in the literature have targeted logic synthesis using ML techniques. The authors in [14] formulate the logic synthesis and optimization step as a Markov decision process (MDP), and use a policy gradient approximation via graph neural networks (GNN) to explore the search space. Similarly, authors in [24] made use of a graph convolutional network (GCN) to incorporate the logic network. Works such as [10, 12, 24] show the ability of GCNs as powerful neural networks designed to work directly on graphs and leverage their structural information. Inspired by the state representation as used in [24], we formulate logic synthesis as an MDP and make use of GCNs to incorporate the structure of logic graphs. However, unlike [24], we model both the reward function and the exit condition for a given logic network using runtime as the primary indicator. The reason behind this consideration is that runtime is often a defining aspect for various logic optimization algorithms and synthesis flows. Hence, our work focuses on generating custom scripts that aim to perform better than the baseline heuristic-based optimization scripts and algorithms of Berkeley-ABC (*compress2rs, dch* and *dc2*) within a specific threshold of their runtime.

We also leverage the use of MIGs with EPFL mockturtle [20] as the logic network representation for a given Boolean network. Since mockturtle does not provide any specific heuristic-based scripts, we develop custom scripts that can be applied for a given network. We generate models to perform inference which outputs a custom sequence of optimization algorithms that perform well over the given logic network.

The major contributions of our work are as follows:

- Proposing a runtime-constrained reinforcement learning approach to develop custom optimization scripts
- Developing an efficient inference framework which can produce custom scripts quickly for any given Boolean circuit
- Employing the MIG logic representation for a Boolean network through the EPFL mockturtle [20] framework and generating custom heuristics for the same

After extensive experimentation the results show that our work performs better than the work [24] on average by 9.5% in reduction in level while giving similar reduction in the number of nodes. The scripts generated by our method also show a



(a) AIG representation (b) MIG representation

Fig. 1: The logic graphs used in Berkeley-ABC and EPFL mockturtle for a 1-bit *Full Adder*. The dotted lines indicate an inverted signal.

lesser degree of variation in runtimes (90%-160%) with respect to the baseline script as compared to their work (102%-215%). Our work also performs up to 8.3% better in average depth reduction and over 2% better in average node reduction than the heuristic-based scripts of Berkeley-ABC. The generated scripts by our RL agent on average run within 30% of the excess runtime of the baseline scripts. The results also produce an overall improvement of up to 27.9% in average depth reduction and 25.5% in average node reduction from the initial configurations.

II. Fundamentals

Within this section, we provide a background for AIGs, MIGs, optimization algorithms in logic synthesis and MDP.

A. And-Inverter Graph

An and-inverter graph (AIG) is a type of logic graph used to functionally represent any Boolean network, i.e the AIG representation is Boolean complete. It behaves as a directed acyclic graph (DAG) where each node is a 2-input AND function, and the edge weight corresponds to whether or not the given signal is inverted. Fig. 1a shows an AIG for a 1-bit *Full Adder*. An AIG is used as a common logic graph representation within logic synthesis frameworks. Berkeley-ABC [4] uses an AIG as the representation of Boolean netlists and provides various optimization techniques [3, 6, 13] that are used to reduce the size and depth of the logic graph.

B. Majority-Inverter Graph

A majority-inverter graph (MIG) [7, 9] is another variant of logic data structures that can fully functionally represent any given Boolean netlist. The MIG also behaves as a DAG, wherein each node is a 3-input majority gate, with the edge weight representing whether the input signal is inverted or not. Fig. 1b shows an MIG for a 1-bit *Full Adder*. EPFL's mockturtle [20] is a logic synthesis framework that provides MIGs as a logic graph representation of Boolean networks. Similar to AIGs, multiple graph based optimization algorithms [7, 18, 19, 23] can be applied over the MIG to exploit its structural and functional properties.

C. Markov Decision Process

MDP refers to a discrete-time stochastic control process. The mathematical framework of any MDP formulation is provided with actions, states, rewards and state transitions. The most fundamental property for an MDP is for any future state that can be reached via any action, to only be completely dependent upon the current state. The works [14, 24] formulate logic synthesis as an MDP and choose an appropriate state representation for AIGs in order for the framework to obey the markovian property. We utilise a similar state representation for the MIG logic graph representation as well.

III. Related Work and Motivation

A. Related Work

Recently, there have been several works [14, 16, 22, 24] that have aimed to employ learning techniques for script generation to perform logic graph minimization. The work [22] developed an advantage actor critic agent that minimises area of a given logic network under some delay constraint. Similarly, the work [16] proposed to use a convolutional neural network (CNN) to determine whether a generated script is useful or not. They randomly generated 4-repetition synthesis flows and pruned them using the trained CNN classifier. However, in their approach, they generate multiple 4-repetition scripts which makes their approach time-intensive. We note that the work [24] used a graph convolutional network (GCN) to provide an RL framework for logic synthesis and define their framework by generating scripts with the same sequence length of the heuristic baseline (*resyn2*). However, they apply this technique only to specific benchmarks which makes their approach time-intensive. Also, they do not generate an inference model which can be applied to any test circuit.

B. Motivation

Our work is motivated by the fact that runtime is a crucial component in defining logic synthesis flows. Hence, we propose a runtime-constrained reward function and threshold for our RL framework, while also designing an inference framework that can produce custom scripts for any benchmark in much lesser overall time. We reason that the sequence length of the baseline script which has been used by previous work [24] is not an appropriate parameter for the RL agent, because the action space consists of both slow and fast optimizations. This can lead to an uncertain runtime of the generated script for a given Boolean circuit. Hence, the baseline runtime is a more representative parameter as compared to the number of steps used by [24].

We aim to find custom sequences tailor-made for a specific benchmark (or type of benchmarks), by utilizing a reinforcement learning (RL) approach which will consume a runtime equivalent to a threshold of the runtime of the competing heuristic script. In reinforcement learning, the learner is a decision-making agent that takes actions in an environment and receives a reward for its actions in trying to solve a problem. After a set of trial and error runs, it learns the best policy, which is the sequence of actions that maximize the total reward. We formulate logic synthesis as an MDP, hence modelling the reinforcement problem to be solved by our agent as the minimization of the nodes and depth of the logic graph.

Drawing a similar parallel, we can describe the environment as the logic graph of the given Boolean circuit and the actions as the optimization algorithms made available to the agent. We further design extensive environments that make use of both the AIG and MIG logic graph representations.

IV. Methodology

Within this section, we introduce our framework for performing reinforcement learning, the revamped reward function, graph convolution networks, the overall neural network architecture, and the reinforcement learning technique used.

A. Reinforcement Learning Framework

1) MDP Formulation: Logic optimization within the paradigm of logic synthesis is inherently a sequential process. Each permutation of sequentially applied algorithms results in a different structural representation of the same Boolean network. The work [24] determines a formulation of state space similar to [14], by incorporating the current logic graph into the state. The state space is governed by the actions provided to the RL agent and we provide a framework for the user to choose the actions available to optimize a given logic network. It is important to note that each permutation of this *generic user-defined* action space results in a new environment, and hence different models for such environments need to be trained in order to provide inference. We therefore develop a *generic* and *user-defined action space*, and hence make use of a generic state space, each unique to its own environment upon which the RL agent proceeds to interact with. Further details regarding the action spaces is provided under section IV-A3.

2) State Space: The state space for the reinforcement learning framework is defined as follows -

- The number of nodes and depth of the current logic graph
- The number of nodes and depth of the previous state
- A normalised one hot vector of the last 3 actions (dynamically dependent on action space size)
- The scalar representing the current state, normalised by the runtime of the expected sequence
- The AIG or MIG Graph

We use the information of the current logic graph, and of the previous state. We then concatenate this information with the normalised one-hot vector representing the previous 3 actions. Finally we concatenate the scalar to represent the state space. The graph embeddings of the AIG or MIG are handled separately using a graph convolutional network explained in section IV-C below. As AIGs have 2 fanins, the different types of nodes can be evaluated as (a) Constant 1 (b) Primary Input (c) Primary Output (d) No inverted inputs (e) One inverted input, and (f) Two inverted inputs.

In case of MIGs which possess 3 fanins, their node type can be defined as (a) Constant 1 (b) Primary Input (c) Primary Output (d) No inverted inputs (e) One inverted input (f) Two inverted inputs, and (g) Three inverted inputs. We then model feature embeddings for either of the graphs by representing each node as a one-hot vector of its node type.

3) Action Space: Since the aim of our work is to compete with the best known heuristic-based optimization scripts of Berkeley-ABC within the threshold of their runtime, our action space is defined as discrete optimization algorithms that are used within *compress2rs*, along with *dch* and *dc2*. We developed a set of environments in order to compare the RL agents effectiveness over the various benchmarks on the different baselines. We model six different environments, each solving a particular issue or tackling a separate baseline, which will be discussed in section V. Five of the six environments use Berkeley-ABC and AIGs as the intermediary representation whereas the last environment uses EPFL mockturtle and MIG to represent the logic graph. Table I lists the environments and their corresponding baselines. Table II shows the various actions for the environments 2-4 based on Berkeley-ABC, along with the environments that they are a part of. The rest of the action spaces for environments are provided in section V. We further develop a generic system that allows a user to select the optimization algorithms that they wish to apply over the logic network, changing the allowed actions for the RL Agent to learn from while competing against the best heuristic-based scripts of Berkeley-ABC.

TABLE I: List of modelled environments

Env no.	Environment	Baseline used
1	Comparison Env	*resyn2; resyn2;*
2	Compress2rs Env	*compress2rs;*
3	Compress2rs Env without balance	*compress2rs;* without balance
4	Dch Env	*compress2rs; dch; balance -l;*
5	Reduced Dch Env	*compress2rs; dch; balance -l;*
6	Mockturtle Env	10 runs of *balance; rewrite;*

TABLE II: List of actions for environment 2,3,4

Action	Env no.
balance -l;	2, 4
rewrite -l;	2, 3, 4
rewrite -z -l;	2, 3, 4
refactor -l;	2, 3, 4
refactor -z -l;	2, 3, 4
resub -K 6 -l;	2, 3
resub -K 6 -N 2 -l;	2, 3
resub -K 8 -l;	2, 3, 4
resub -K 8 -N 2 -l;	2, 3, 4
resub -K 10 -l;	2, 3, 4
resub -K 10 -N 2 -l;	2, 3, 4
resub -K 12 -l;	2, 3, 4
resub -K 12 -N 2 -l;	2, 3, 4
resub -K 16 -l;	2, 3, 4
resub -K 16 -N 2 -l;	2, 3, 4
dch;	4
dc2;	4

4) Reward Function: As mentioned in sections above that runtime of any optimization is an important criterion within the domain of logic synthesis, we aim to devise a better overall optimization sequence within the limits of the runtime of the best heuristic-based scripts. Hence, we model our reward function with paying special emphasis to the runtime of any given optimization algorithm. We propose to find the marginal gain of improvement of a state s_k on taking an action a_k with a runtime of t_k, reaching a new state s_{k+1}. We define the term *betterment function* using a value function that provides a weighted sum of a given state s_k's normalised number of nodes and depth, with respect to the initial state. The value function is defined in equation 1, where c_1 and c_2, are configurable parameters that are fixed per experiment. $s_k.n$

and $s_k.l$ refer to the normalised number of nodes and depth of the k^{th} state with respect to the initial network configuration.

$$v(s_k) = \frac{c_1 * s_k.n + c_2 * s_k.l}{c_1 + c_2} \tag{1}$$

The *betterment function* between any two states is then defined as the difference of their value functions. We then finally define our reward function, utilising the marginal improvement of the state, along with using a $baseline$, in order to help with the convergence. The $baseline$ is defined by the *betterment* gained by applying the baseline optimization algorithm to the benchmark that we use for any given environment (mentioned in Table I), which is further normalised by the runtime of the algorithm. The reward function is defined by equation 2.

$$r(k) = \frac{(v(s_k) - v(s_{k+1}))}{t_k} - baseline \tag{2}$$

We can configure the optimization focus of the RL Agent to either target reduction of the size or the depth of the circuit. This can be carried out simply by altering the value of c_1 (size) and c_2 (depth) to handle the appropriate weight given to the size or the depth of the circuit in the value function.

B. Reinforcement Learning Algorithm

We make use of the REINFORCE with Baseline [2] method provided in algorithm 1 [15], which is a policy gradient method that aims to directly learn the policy $\pi(a|s, \theta)$. The action values gained from this policy are used to develop a probability distribution using softmax. The desired action is then sampled from this distribution, to aid the agent in exploration. In an episode, a trajectory is followed which performs a set of actions (as derived from the action spaces provided in Table II) until the termination condition is reached. We model the termination condition as a threshold on the overall runtime of the RL agents applied actions. Once these actions are accumulated, the method uses a discount factor γ to provide the policy to learn from experience. We set the value of γ to be 0.95.

The objective of this method is to maximise the expected total reward gained on a trajectory following π. In order to reduce the variance in the expected returns, a baseline is subtracted from it. The baseline is modelled as a differentiable state-value function that approximates the expected reward.

C. Graph Convolutional Network

Due to logic networks being represented as directed acyclic graphs (DAGs) in both the AIG and MIG intermediary representations, we use graph convolutional networks, in order to capture information related to the structure and property of the logic graph. As specified before, the initial node feature is defined by the node type to the logic graph that a given node belongs to. We define three layers of graph convolutions over the initial features, wherein each layer accumulates information based on node connectivity, and serves as the feature embedding for the next layer.

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}) \tag{3}$$

Algorithm 1 REINFORCE with Baseline for estimating π_θ

Input: A differentiable policy parameterization $\pi(a|s, \theta)$
Input: A differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Output: Final updated policy π_θ and $\hat{v}(s, \mathbf{w})$
  Initialize policy parameter θ
  for Number of Episodes **do**
    Generate a trajectory $s_0, a_0, R_1, ...s_{T-1}, a_{T-1}, R_T$ following $\pi(.|., \theta)$
    for each step of the trajectory t=0,1,2...$T-1$ **do**
      $G \leftarrow \Sigma_{k=t+1}^{T} \gamma^{k-t-1} R_k$
      $\delta \leftarrow G - \hat{v}(s_t, \mathbf{w})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(s_t, \mathbf{w})$
      $\theta \leftarrow \theta + \alpha^{\theta} \gamma^t \delta \nabla \ln \pi(a_t|s_t, \theta)$
    end for
  end for

Where $\mathcal{N}(i)$ is the neighbour set of node i, σ is the activation function and c_{ij} is equal to the product of the square root of node degrees: $\sqrt{|\mathcal{N}(i)|}\sqrt{|\mathcal{N}(j)|}$. We then take the average of all output features $h^{(v)}$ from the successive convolutions as the final graph feature,

$$h_{final} = \frac{1}{P} \sum_{i \in P} h_i^{(v)} \tag{4}$$

D. Neural Network Architecture

The network architecture for policy parameterization $\pi(a|s, \theta)$ is specified in Fig. 2. We use a simple fully connected feed forward neural network with graph embeddings from the GCN as the differentiable policy parameterization. The graph convolution network consists of 3 convolutional layers, of either (6, 12, 12, 4) or (7, 12, 12, 4) channels for inputs, hidden layers and outputs of the AIG or MIG logic graph respectively. The simple feed forward net consists of 3 fully connected layers with ReLU as the activation function. For the differentiable state-value function parameterization $\hat{v}(s, a)$, we use a 3 layered fully connected neural network with ReLU activations as well. The input vector is our state-space representation. It becomes a dynamic vector of length equivalent to *number of actions* + 5. The policy parameterization π_θ outputs a probability distribution of actions to be taken and the state-value parameterization returns a scalar for the expected reward for that given state.

E. Runtime Indicator and Inference Framework

We use runtime as our primary indicator in our modelling of the framework. The exit condition for an episode is defined as a threshold of the runtime of the best heuristic-based scripts of Berkeley-ABC. We aim to perform better than the heuristic-based methods in approximately a similar amount of processing time. To account for border cases wherein the generated trajectory currently utilises up to 80% of the runtime threshold or above, we set the final threshold upon the runtime of the trajectory at 20% over the exact runtime of the baseline heuristic-based script for each environment.

We also carry forward the Pytorch [17] models that are used to explore the state space from benchmark to benchmark

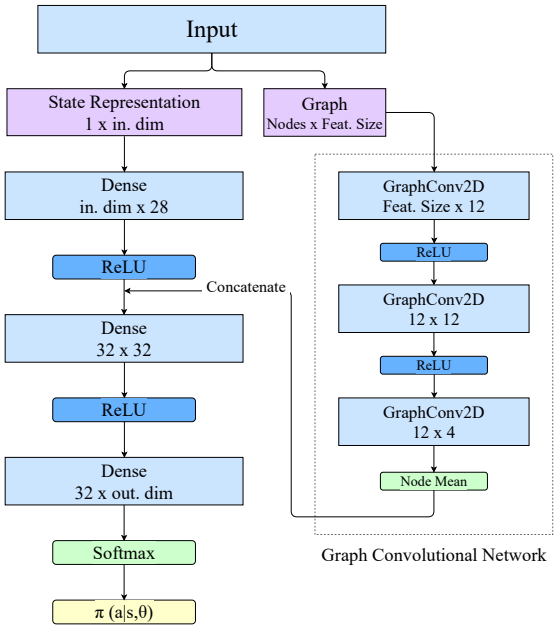


Fig. 2: Neural network architecture for policy parameterization π_θ

over our training set. Both the policy parameterization and state-value parameterization models are trained over a random order of selected EPFL benchmarks [8]. These models are then used within the designed inference engine, that directly produces a custom script that runs within the time threshold of the baseline scripts. For any given environment, we can modify its hyper parameters identified by c_1, c_2, in order to target more specific optimizations using our RL agent. We develop six different policy and state-value parameterizations for each environment, based on manipulating these hyper parameters. We then perform inference by running all the models over the input logic circuit and pick the best result by using the value function.

V. Experimentation

The Berkeley-ABC as well as the EPFL mockturtle python interface[1], were implemented using C++ and pybind11 [11]. The graph convolutional network is implemented using *Deep Graph Library* [21]. All the environments are trained on CPU, as the network size is relatively small. All our experiments are performed on a server cluster.

Since environment 1 is a comparison with the state-of-the-art [24], the same benchmarks in their work are used to perform training. For the rest of the environments, we extract a training set of benchmarks from the arithmetic and random_control EPFL Benchmark Suite [8]. For all the experiments, we make the entire action space available to the RL agent in order to make fair comparisons with the heuristic-based scripts of Berkeley-ABC. We run 200 episodes per benchmark as training, and accumulate the results by running inference on all the benchmarks. The experiments are split into 3 sections, one regarding comparisons with the state-of-the-art work [24], another tackling the stronger heuristic-based scripts of Berkeley-ABC and finally employing the MIG representation via the EPFL mockturtle framework.

[1]https://github.com/YasasviPeruvemba/mtlPy

A. Comparison with the state-of-the-art (Env 1)

To draw fair comparisons to the work of [24], we make use of the same benchmarks, baseline script (2 runs of *resyn2*) and action space. We make use of 6 different value pairs mentioned in Table IV for the reward function and pick the best script post inference. On replicating their work, we note that the runtimes of the generated scripts from their method is uncertain and vary from 102% to over 215% resulting in an average of 41% excess usage of the runtime of the baseline script. Our scripts however display a lesser degree of variation from 90% to 160% with an average use of 32% of excess runtime of the baseline script. We also note that the time taken to generate the scripts by our method by running inference on 6 value pairs (4hrs 35min) is around 25% of the overall time taken by their approach over a single reward setting (17hrs 40min). Table III shows the inference results of our work contrasted with the results of the 2 sets of flows reported by the work [24]. Both *RL-1* and *RL-2* are flows dedicated to node reduction and level reduction respectively as proposed by [24]. The results under column *RF* indicate the inference results of our framework. The suffixes *N* and *L* pertain to the nodes and level of the optimized circuit representation. We note that our scripts provide upto 30% reduction in number of nodes and up to 35.7% reduction in level from the initial circuit configuration. This outperforms *RL-1* in level reduction by 9.5%, while being within 1% of the reduction in nodes. With respect to *RL-2*, our work gives 1.1% better reduction in nodes and 2% betterment in levels.

TABLE III: Comparison with the state-of-the-art [24]

Benchmark	RL-1_N	RL-1_L	RL-2_N	RL-2_L	RF_N	RF_L
C1355	386.2	17.6	390.0	16.0	386.0	17.0
C6288	1870.0	88.0	1882.0	88.0	1870.0	88.0
C5315	1337.4	27.2	1364.7	25.4	1315.0	24.0
dalu	1039.8	33.2	1095.6	30.0	1085.0	29.0
k2	1128.4	19.8	1187.5	13.0	1137.0	13.0
mainpla	3438.4	25.0	3504.0	25.5	3461.0	23.0
apex1	1921.6	19.2	2004.7	17.0	1885.0	17.0
bc0	819.4	18.6	851.7	17.5	831.0	16.0
Improvement from initial circuit	30.9%	26.2%	28.9%	33.7%	30%	35.7%

B. Heuristic-based scripts of Berkeley-ABC (Env 2-5)

1) ***Env 2 (Baseline script: compress2rs)****:* For the action space of this environment, we simply enumerate the different types of optimizations utilised inherently by Berkeley-ABC within *compress2rs*. We also add 2 very large optimizations as resubstitutions with a maximum cut size of 16, along with a node replacement option of 2. We train 8 different models, with the value pairs of (c_1, c_2) as provided in Table IV.

We average the total sum of rewards for all benchmarks over each episode of a given value pair, and observe that for the value pair (0, 1), the rewards show erratic behaviour. We note that this occurs due to some optimizations being heavily biased towards reducing level (eg. *balance*), in a very short time, presenting a case of local minima for the RL agent.

TABLE IV: Value pairs of c_1, c_2 for each environment

c_1	c_2	Env. numbers
0	1	2, 3
1	0	1, 2, 3, 4, 5, 6
1	1	1, 2, 3, 4, 5, 6
2	1	1, 2, 3, 4, 5, 6
2	3	1, 2, 3, 4, 5, 6
2	5	2, 3
2	7	1, 2, 3, 4, 5, 6
2	9	1, 2, 3, 4, 5, 6

Note that this particular behaviour is not observed for cases where the value function holds importance to the number of nodes. Fig. 3a shows the reward plot for the value pair (2, 3) during training. To tackle the bias towards depth from the optimization algorithm *balance*, we create and perform experiments on environment 3.

2) ***Env 3 (Baseline script: compress2rs - without balance)****:* The algorithm *balance* is a very time inexpensive optimization script used to perform the algebraic balancing of the multi-input AND-gates contained within the logic graph. The balancing is applied in a topological order selecting the minimum delay tree-decomposition for each multi-input AND-gate. To perceive the effects of *balance* on our state space, we design another environment similar to environment 2 but devoid of this optimization. The baseline is a version of compress2rs without any balancing steps.

We observe a slight betterment in the quality of results (Fig. 3b) from a theoretical perspective as the cumulative rewards gained are relatively better, and the agent starts to outperform the baseline from 75 episodes onward. But the lack of *balance* makes the result worse with respect to the final depth as compared to that of environment 2. Therefore, we incorporate *balance* into the modelling of the rest of the environments, while keeping sure that no value pair is unduly biased toward the level. We therefore remove the option for value pair (0, 1) from all further experiments.

3) ***Env 4 (Baseline script: compress2rs; dch; balance -l)****:* With environment 4, we aim to push the threshold further by using stronger optimizations provided by Berkeley-ABC. We include both *dch* and *dc2* within our action space, while discarding the two relatively less used optimizations of resubstitutions of maximum cut size 6 with node replacement option 2. From Fig. 3c, we observe that adding more powerful optimizations into the action space result in the total rewards overtaking that of the baseline within 55 episodes. We investigate this performance gain by inspecting the value function at the end of every episode, averaging it over all the benchmarks in Fig. 4a. This gradually downward flowing graph provides us with 2 interesting observations – (i) when total sum of rewards start to perform much better than the baseline, the value function goes below the baseline script value, i.e with an average of 26% betterment from the initial circuit. We note spikes in value functions where there are dips in the reward graph, which is logically consistent (as an average decrease in reward pertains to lesser improvement from the baseline in terms of depth and size); (ii) despite gaining such improvements on performance, there are still circuits that the RL agent only performs slightly better than the baseline script in. In some cases where circuits are small (eg - *ctrl, int2float*), and already well optimized, there seems to be little room for the RL agent to minimize. Hence causing the gradual downward slope in the value graph.

TABLE V: Action space for environment 5 using cumulative algorithms as a single action

Actions
rewrite -z -l; balance -l;
dch; balance -l;
dc2;
resub -K 8 -l; refactor -z -l; resub -K 8 -N 2 -l;
resub -K 10 -l; refactor -z -l; resub -K 10 -N 2 -l;
resub -K 12 -l; refactor -z -l; resub -K 12 -N 2 -l;

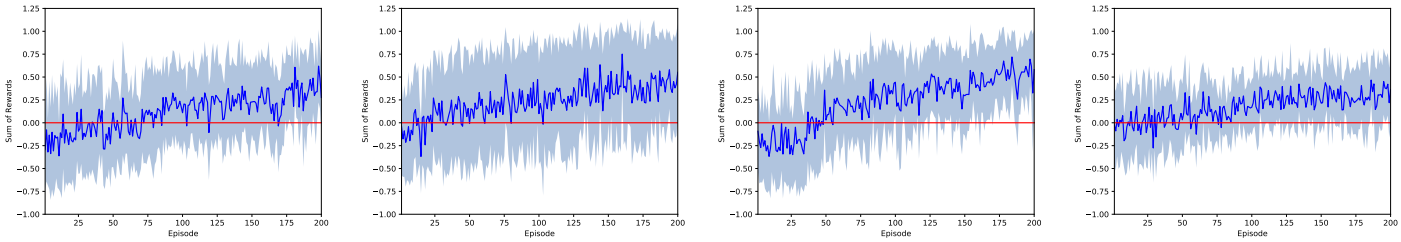
4) ***Env 5 (Baseline script: compress2rs; dch; balance -l)****:* For this particular environment we do not produce models to run inference. We desire that the RL agent specifically find the best script for a given logic network, without any bias from the other benchmarks. We analyse all the scripts generated by the environments 2 through 4 and aim to devise a smaller action space in order to – (i) reduce the vastly variant search space, focusing on a more thorougher exploration and exploitation over the training networks; (ii) normalise the run time of each action to avoid extra biasing.

On inspecting the script *compress2rs*, we notice a particular repeating structure wherein, 2 resubstitutions of the same maximum cut size are applied surrounding either a rewriting or refactoring algorithm. We use this structure with *refactor* to form a boilerplate action consisting of multiple algorithms and derive statistics over the custom algorithms given by the RL agent for env 2, 3 and 4 in order to produce the derived action space given in Table V. We discard resubstitutions for maximum cut size 16 from the action space, as they were the least used resubstitution algorithms. Both *rewrite -l* and *refactor -l* were also discarded due to them having significantly less usage compared to their zero cost replacement counterpart.

From the reward plots of environment 5 (Fig. 3d), we observe that having a smaller action space leads to better performance along with the total rewards being consistently above the baseline. We also note a decrease in variance of the rewards, hence indicating a better learning for the RL agent. This environment performs better than environment 4 in terms of value function reduction as well due to the RL agent being able to better navigate the search space. Fig. 4b shows the average value function plot over the 200 episodes. Despite a gradual slope, we notice that the RL agent performs better than the baseline, consistently from 130 episodes onward.

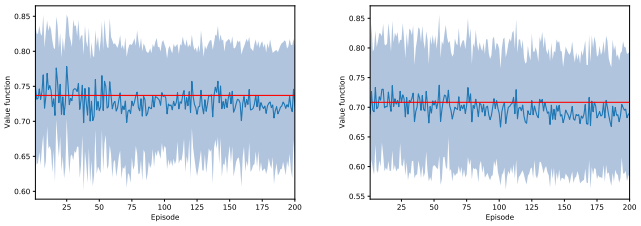
C. EPFL mockturtle framework : ***Env 6 (Baseline script: 10 runs of balance; rewrite)***

Environment 6 is based on EPFL mockturtle, wherein we represent the logic graph using an MIG. Mockturtle provides many configurable algorithms for performing logic optimizations on the MIG [7]. We design the action space for this environment using the most basic techniques that are provided, in order to develop an understanding of the MIG optimization algorithms. Table VI specifies the action space used within



(a) Env. 2 with value pair (2, 3) (b) Env. 3 with value pair (1, 1) (c) Env. 4 with value pair (1, 0) (d) Env. 5 with value pair (1, 1)

Fig. 3: The reward plots showing the performance of the RL agent on different environments for a specific value pair. The shadow describes the standard deviation. The red line indicates the baseline performance.



(a) Env. 4 with value pair (1, 0) (b) Env. 5 with value pair (1, 1)

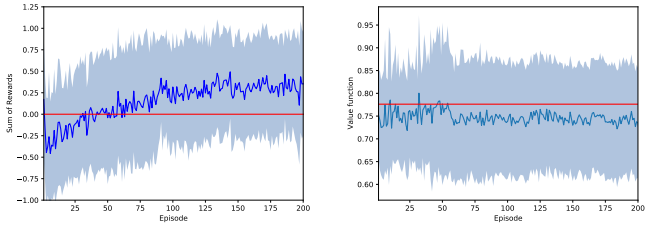
Fig. 4: Value function plots showing the betterment of the RL agent on different environments for a specific value pair. The shadow describes the standard deviation. The red line indicates betterment by the baseline script from the initial circuit.

TABLE VI: Action space for environment 6

Action	Description
rewrite;	Performs basic rewriting of MIG
rewrite -udc;	Rewriting while using don't cares
rewrite -azg;	Rewriting while allowing 0 gain substitutions
rewrite -udc -azg;	Rewriting using don't cares with 0 gain substitutions
balance;	Performs basic balancing of the MIG
balance -c;	Balancing only the critical path of MIG

this environment. We specifically choose a smaller set of actions similar to environment 5 for better performance. As more complex algorithms are developed and integrated into the mockturtle python interface, the action space space could be further modelled to provide better results.

We analyse the reward function plot shown in Fig. 5a and observe that the RL agent starts to perform better than the baseline script within 65 episodes. This plot grows very slowly after 90 episodes, hence causing minute changes in the value function as can be seen in Fig. 5b. We believe this occurs due to two reasons, namely – (i) the nature of the action space, resulting in similar actions being taken that provide a sufficient reward; (ii) an inherent minima for the logic graph size and depth that can be achieved by these combinations of algorithms In either case, we believe this to be a good starting point for further analysis and exploitation of reinforcement learning for logic synthesis over MIGs.



(a) (b)

Fig. 5: Reward and value function plots for environment 6 based on EPFL mockturtle (a) Reward plot showing the performance of the RL agent on environment 6 for value pair (2, 9). (b) Value function plot showing the betterment of the RL agent on different environments for value pair (2, 9). The shadow describes the standard deviation. The red line indicates the baseline performance.

VI. Results

We estimate the effectiveness of the RL agent's performance over environments 2-6, by showing both the training and inference results over the EPFL benchmark suite[2]. As *Adder, arbitrer* and *decoder* showed no improvement on applying optimization algorithms, they were excluded from our experimentation along with *div* and *hyp*, which showed significant training times, being large circuits. The training results are obtained from the best script generated in the last 10 episodes during the training phase. Each result tabulation describes the range of optimizations achieved by showing the number of nodes and levels of the worst and best value pair setting for each benchmark denoted by the superscripts w and b. The baselines for environments are provided in Table I. We show the results on each benchmark by the RL agent under *RF_L* and *RF_N* which represent the level and nodes of the logic graph. The best and worst cases for each benchmark are determined by performing a value function calculation of the resulting nodes and level with (c_1, c_2) as (1, 3).

ABC_L, *ABC_N* as well as *MTL_L*, *MTL_N* depict the level and nodes achieved by applying the baseline over the benchmark. *Ratio* refers to the ratio of the runtimes of the

[2]excluding adder, arbitrer, decoder, div and hyp

TABLE VII: The inference results of the RL agent on Env 4

Benchmark	Ratio^b	RF_N^w	ABC_N	RF_N^b	RF_L^w	ABC_L	RF_L^b	Opt^b
sin	0.79	5175.0	4983.0	5019.0	224.0	159.0	160.0	1_0
voter	0.62	10065.0	7930.0	8015.0	59.0	60.0	57.0	1_0
priority	1.33	785.0	427.0	429.0	235.0	44.0	33.0	2_7
max	0.88	2830.0	2828.0	2828.0	296.0	175.0	163.0	1_0
router	0.87	244.0	147.0	146.0	52.0	19.0	19.0	1_1
mem_ctrl	0.45	44944.0	39733.0	30522.0	114.0	89.0	78.0	2_7
cavlc	0.87	650.0	612.0	600.0	18.0	14.0	15.0	1_1
sqrt	0.65	18340.0	18220.0	18229.0	6054.0	6018.0	5939.0	1_0
i10	0.79	2118.0	1690.0	1654.0	49.0	37.0	31.0	1_1
multiplier	0.85	26058.0	24336.0	24348.0	273.0	264.0	262.0	1_1
i2c	0.82	1210.0	1015.0	997.0	20.0	13.0	12.0	1_1
square	0.47	16933.0	15842.0	15790.0	248.0	248.0	246.0	2_7
bar	0.82	2952.0	2952.0	2952.0	14.0	12.0	12.0	2_1
int2float	1.80	231.0	206.0	193.0	16.0	11.0	10.0	2_7
ctrl	3.27	89.0	88.0	90.0	11.0	10.0	7.0	2_7
log2	0.44	29618.0	28676.0	28599.0	402.0	303.0	303.0	2_7
Improvement from initial circuit			23.8%	25.5%		23.4%	27.9%	

w Worst Case. b Best Case.

TABLE VIII: The training results of the RL agent on Env 5

Benchmark	RF_N^w	ABC_N	RF_N^b	RF_L^w	ABC_L	RF_L^b	Opt^b
sin	4949.0	4983.0	4965.0	165.0	159.0	157.0	2_9
voter	7974.0	7930.0	8153.0	58.0	60.0	55.0	1_1
priority	670.0	427.0	427.0	202.0	44.0	41.0	2_1
max	2807.0	2828.0	2826.0	208.0	175.0	157.0	2_9
router	201.0	147.0	143.0	23.0	19.0	17.0	1_0
cavlc	578.0	612.0	596.0	15.0	14.0	12.0	1_1
i10	1591.0	1690.0	1634.0	35.0	37.0	31.0	1_0
i2c	1011.0	1015.0	1026.0	14.0	13.0	11.0	2_7
bar	2952.0	2952.0	2952.0	12.0	12.0	12.0	2_1
int2float	194.0	206.0	193.0	11.0	11.0	10.0	2_1
ctrl	79.0	88.0	102.0	8.0	10.0	6.0	2_3
Improvement from initial circuit		34.9%	46.2%		35.6%	53.4%	

w Worst Case. b Best Case.

baseline and the generated script by the RL agent. *Opt* refers to the value pair used. We compare the results of the best scripts and our RL framework by deriving the betterment from the initial circuits with the value pair (c_1, c_2) as (1, 3).

Table VII displays the inference results of our RL agent over environment 4. We note that our inference framework outperforms the best heuristic-based scripts of Berkeley-ABC by an average margin of 3.8% in value function reduction, and an average overall betterment of 27.3% from the initial benchmark configurations. On further analysis, we see similar patterns of value pair (2, 7) performing well in environment 3 over the benchmarks in both training and inference. The same overall trend is noted wherein, a similar value pair performs well on a particular benchmark over each environment. The complete set of results for all environments are not shown here due to space constraints and can be viewed here[3] instead.

Table VIII displays the training results on environment 5 which was specifically designed with a smaller and more concise action space. The training results of environment 5 outperform the best heuristic-based optimizations of Berkeley-ABC by 7% in average reduction of value function and shows an overall of 36.5% reduction in value function from the initial circuits. We note that the reduction in depth compared to the baseline is consistent among all the benchmarks.

We leverage newer more configurable optimization algorithms and the logic representation of MIG provided within EPFL mockturtle in designing environment 6. Table IX shows the inference results for environment 6 based upon EPFL mockturtle. We notice a significant increase in performance by the inference engine from the baseline with a average reduction in value function of 18% with an overall value function

[3]https://github.com/YasasviPeruvemba/reinforcedLS

TABLE IX: The inference results of the RL agent on Env 6

Benchmark	Ratio^b	RF_N^w	MTL_N	RF_N^b	RF_L^w	MTL_L	RF_L^b	Opt^b
sin	1.00	8141.0	12974.0	6632.0	138.0	144.0	150.0	2_3
voter	0.98	17230.0	20263.0	11807.0	62.0	65.0	68.0	2_3
priority	0.82	1553.0	1885.0	1201.0	186.0	145.0	186.0	1_0
max	1.02	4704.0	7056.0	2926.0	116.0	159.0	95.0	2_3
router	0.86	360.0	514.0	426.0	31.0	17.0	19.0	1_1
mem_ctrl	0.98	57789.0	84180.0	57779.0	108.0	65.0	95.0	2_9
cavlc	0.88	727.0	761.0	733.0	16.0	12.0	12.0	2_9
sqrt	0.50	59157.0	48050.0	16322.0	4471.0	4060.0	2416.0	1_0
i10	0.90	2683.0	3663.0	2940.0	38.0	26.0	27.0	2_9
multiplier	1.04	35857.0	63295.0	33823.0	201.0	252.0	190.0	2_3
i2c	0.90	1570.0	1631.0	1546.0	17.0	10.0	10.0	2_9
square	0.72	21735.0	30146.0	19580.0	159.0	207.0	55.0	2_9
bar	0.76	3615.0	3615.0	3309.0	12.0	12.0	12.0	2_1
int2float	0.76	264.0	274.0	269.0	15.0	10.0	10.0	2_9
ctrl	0.54	129.0	139.0	127.0	7.0	6.0	6.0	2_9
log2	0.90	44690.0	76271.0	42222.0	343.0	304.0	313.0	1_1
Improvement from initial circuit			-52.3%	-3.2%		32.4%	37.4%	

w Worst Case. b Best Case.

reduction of 27.3% from the original circuit representations. Specifically for the benchmarks *voter, sqrt* and *multiplier*, we see significant improvements over their respective baselines. We believe that with more optimization algorithms being integrated into the mockturtle python interface, the RL agent will be able to perform even better.

For outlier cases with respect to runtime (eg - *log2, square* in Table VII and *ctrl, sqrt* in Table IX), we note that the final action chosen is an action with a significant contribution to the total runtime of the script generated (eg - *dch, resub -K 16 -l, rewrite -azg*). Hence, choosing such large optimizations as the last action by the RL agent before the threshold being violated, leads to a larger runtime than anticipated. We aim to thoroughly investigate this occurrence in our future work.

VII. Conclusion

In this work, we develop a runtime-constrained reinforcement learning based framework incorporating a graph convolutional network to create tailor-made scripts that outperform the prevalent state-of-the-art and heuristic-based scripts of Berkeley-ABC within the bounds of a similar runtime. We develop frameworks for both AIG and MIG as logic graph representations. We model 6 different environments using varied action spaces to develop models for our RL agent to provide inference. To demonstrate the results of our RL agent, we compare the inference results of our framework to the results provided in [24] and note a performance increase of 9.5% in reduction of level in approximately 25% of the overall time taken. We perform inference runs over the EPFL Benchmark Suite and show that our agent was able to learn the search space effectively and outperform the best heuristic-based scripts of Berkeley-ABC in *compress2rs, dch and dc2*, within the threshold of their runtime. We note that scripts generated from the inference models for environments 1, 2, 3, 4 and 6 all outperform their respective baselines. Further, the results of environment 5 based on EPFL mockturtle display that the RL agent can provide very proficient optimizations with up to an average of 36.5% value function reduction from the initial circuit structure. In conclusion, this framework generates custom scripts faster and provides better reduction of logic graph depth and size within a threshold of the runtime of the competing baseline heuristic of choice.

REFERENCES

[1] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. 1994.

[2] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *NIPS*. 1999.

[3] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. "DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis". In: *DAC*. 2006.

[4] Alan Mishchenko et al. *ABC: A system for sequential synthesis and verification*. 2007.

[5] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: CAV'10. 2010.

[6] A. Mishchenko et al. "Scalable don't-care-based logic optimization and resynthesis". In: *ACM TRECTS.* (2011).

[7] L. Amarú, P. Gaillardon, and G. De Micheli. "Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization". In: *DAC*. 2014.

[8] L. Amarù, P. Gaillardon, and G. Micheli. *The EPFL Combinational Benchmark Suite*. 2015.

[9] L. Amarú, P. E. Gaillardon, and G. De Micheli. "Majority-Inverter Graph: A New Paradigm for Logic Optimization". In: *TCAD* (2016).

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: *NIPS*. 2016.

[11] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[12] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *ICLR*. 2017.

[13] L. G. Amarù et al. "Improvements to Boolean resynthesis". In: *DATE*. 2018.

[14] W. Haaswijk et al. "Deep Learning for Logic Optimization Algorithms". In: *ISCAS*. 2018.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[16] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. "Developing Synthesis Flows without Human Knowledge". In: *DAC*. Association for Computing Machinery, 2018.

[17] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS*. 2019.

[18] H. Riener et al. "On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis". In: *DATE*. 2019.

[19] H. Riener et al. "Scalable Generic Logic Synthesis: One Approach to Rule Them All". In: *DAC*. 2019.

[20] Mathias Soeken et al. *The EPFL logic synthesis libraries*. arXiv:1805.05121v2. Nov. 2019.

[21] Minjie Wang et al. "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks". In: *arXiv preprint arXiv:1909.01315* (2019).

[22] A. Hosny et al. "DRiLLS: Deep Reinforcement Learning for Logic Synthesis". In: *ASP-DAC*. 2020.

[23] H. Riener, A. Mishchenko, and M. Soeken. "Exact DAG-aware Rewriting". In: *DATE*. 2020.

[24] Keren Zhu et al. "Exploring Logic Optimizations with Reinforcement Learning and Graph Convolutional Network". In: *MLCAD*. Association for Computing Machinery, 2020.