Neural Combinatorial Logic Circuit Synthesis from Input-Output Examples

Peter Belcak
ETH Zürich
belcak@ethz.ch

Roger Wattenhofer
ETH Zürich
wattenhofer@ethz.ch

Abstract

We propose a novel, fully explainable neural approach to synthesis of combinatorial logic circuits from input-output examples. The carrying advantage of our method[1] is that it readily extends to *inductive* scenarios, where the set of examples is incomplete but still indicative of the desired behaviour. Our method can be employed for a virtually arbitrary choice of atoms – from logic gates to FPGA blocks – as long as they can be formulated in a differentiable fashion, and consistently yields good results for synthesis of practical circuits of increasing size. In particular, we succeed in learning a number of arithmetic, bitwise, and signal-routing operations, and even generalise towards the correct behaviour in inductive scenarios. Our method, attacking a discrete logical synthesis problem with an explainable neural approach, hints at a wider promise for synthesis and reasoning-related tasks.

1 Introduction

Logic circuit synthesis is the process of producing a logic circuit that satisfies a given specification and is a classical problem in computer science. The synthesis of high-level designs to circuits is typically done as direct compilation of hardware description language code coupled with post-processing optimisation [1, 13], although recent work hints at that there exists room for merging the two by the use of neural compiler architectures [15]. Transformer-based neural architectures have also been used for comprehension of linear temporal logic [9], its synthesis (as the input, high-level specification) into circuits [16], and the understanding of abstract mathematical patterns more generally [2].

Synthesising logic circuits from input-output listings, on the other hand, falls under the paradigm of programming by example. If the listings are incomplete (i.e. not all possible inputs are considered) but indicative of the desired behaviour, one talks of *circuit induction*. Note that a well-posed instance of circuit induction still has a unique desired behaviour for the target circuit but is nevertheless more difficult, since any attempt to address it has to incorporate the guiding assumptions while relying on example data that may potentially contradict some interpretations of these assumptions.

We focus on the synthesis of combinatorial logic circuits from input-output examples. This old problem of theoretical interest to mathematical logic [7, 5] has recently been enjoying a small renaissance due to plethora of specific machine-learning applications [4, 6, 8, 11, 14, 12]. The traditional synthesise-and-optimise approaches [10] do not scale well and and are not applicable for later deployment on modern, by design already heavily optimised hardware. Further, it is often far from practical to describe the desired circuit in terms of input-output examples *completely*, and there is therefore an appetite for methods that could induce the correct behaviour from a few guiding examples, or query the user interactively.

To this end, we present a new method for synthesising circuits for combinatorial behaviour utilising any appropriately formulated design of atomic unit. It is in the design of this unit where the user

[1] We make all our code and data available at https://github.com/pbelcak/neccs.

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on Math-AI.

may choose to incorporate their implicit inductive assumptions, say in the form of making certain operations readily available (so that they need not be learned) or initialising decision weights in favour of particular configurations. Our method relies on a neural-network-like layout of atomic units and admits training with gradient descent and loss, as is common in deep learning. In situations where some examples are missing from the input-output specification, our method manages to incorporate the implicit design bias of the atoms to generalise well to previously unseen inputs, which is in its own right a challenging problem for classical deep neural networks [3].

As a broader message of this workshop contribution, we wish to suggest that with appropriate architectures and inductive biases in place, neural methods can feasibly be used to tackle traditionally strictly discrete and reasoning-heavy problems such as circuit synthesis while achieving high levels of accuracy and possessing full model explainability through its outputs. We further believe that neural methods may be the natural choice for situations where inductive assumptions need to be incorporated into the synthesis and reasoning processes.

Formal setting. Let $f : \{0,1\}^{w_i} \to \{0,1\}^{w_o}$ be a Boolean function and let $\mathcal{D} := \{(x, f(x)) : x \in \{0,1\}^{w_i}\}$ be the dataset of all input-output pairs. We call the task of constructing a circuit $\mathcal{C}$ from $\mathcal{D}$ such that $\mathcal{C}(x) = f(x)\, \forall x \in \{0,1\}^{w_i}$ *circuit synthesis* from (complete) examples. If, instead of $\mathcal{D}$, we are given $\mathcal{D}' \subset \mathcal{D}$, constructing $\mathcal{C}$ from $\mathcal{D}'$ is called circuit induction.

2 Method

Our method can be used to address both the task of circuit synthesis and induction. It encompasses the translation of discrete logical units such as traditional logic gates or FPGA blocks, and the feed-forward combinatorial differentiable wiring technique that allows connections to emerge based on need.

Given these logical units, the challenge of the traditional circuit synthesis is to make an appropriate choice of available units and then wire them in a fashion that leads to correct functionality. In contrast, our method relies on a single choice of the universal unit type that is made before the synthesis.

2.1 Logical Units from Differentiable Operators

For given differentiable input signal lines $i_1, i_2 \in [0,1] \subset \mathbb{R}$, the basic logical operations such as NOT, AND, OR, or XOR can be readily translated to their differentiable counterparts as $1 - i_1$, $i_1 i_2$, $i_1 + i_2$, $i_1(1 - i_2) + i_2(1 - i_1)$, respectively. As any Boolean function $f : \{0,1\}^{w_i} \to \{0,1\}^{w_o}$ can be represented as a network of a subset of these gates, such f may also be differentiably implemented simply by composing the individual gates' differentiable counterparts appropriately. Thus, any Boolean function representing a combinatorial circuit can be implemented with the use of differentiable operators.

Unlike the traditional synthesis setting where the algorithm has to make the choices of *which unit to use* and *what to connect it with* to ensure correctness, we decide to use only units that are *universal* in the sense that a homogeneous network of such properly configured units can represent any f as above. Note that our approach to network building (cf. Section 2.2) extends to the scenario where multiple types of units are to be interconnected, but such situations raise the problem of arrangement of heterogeneous units into network layers so that an arbitrary function can still be learned.

We consider three chosen types for the universal unit, namely the and-inverter gate (AIG), 4-to-1 look-up table (LUT), and the corresponding 4-to-1 LUT-adder block (LAB).

And-inverter gate. The (2-bit) AIG simply computes $1 - i_1 i_2$ for input signals i_1, i_2 as above.

Look-up table. Our LUT is a continuous variant of look-up tables commonly found in field-programmable gate arrays [10] and consists of a 16-dimensional learned (bias) tensor $\mathcal{T}$ arranged into four axes corresponds to a filtering decision on the four inputs to compose the final output. Let $\boldsymbol{i}$ be the vector of inputs, denote $\mathcal{I}$ a list of indices, $0 \leq |\mathcal{I}|$ its length, and ϵ the empty list. Then the output of the lookup operation $\mathrm{LUT}(\boldsymbol{i})$ can be computed as

$$
\begin{aligned}
\mathrm{LUT}(\boldsymbol{i}) &:= \mathrm{LUT}(\boldsymbol{i}, \epsilon) \\
\mathrm{LUT}(\boldsymbol{i}, \mathcal{I}) &:= \boldsymbol{i}_{1+|\mathcal{I}|} \mathrm{LUT}(\boldsymbol{i}, \mathcal{I}{:}1) + \left(1 - \boldsymbol{i}_{1+|\mathcal{I}|}\right) \mathrm{LUT}(\boldsymbol{i}, \mathcal{I}{:}0) && \text{for } |\mathcal{I}| < 4 \\
\mathrm{LUT}(\boldsymbol{i}, \mathcal{I}) &:= \mathcal{T}[\mathcal{I}] && \text{for } |\mathcal{I}| = 4
\end{aligned}
$$

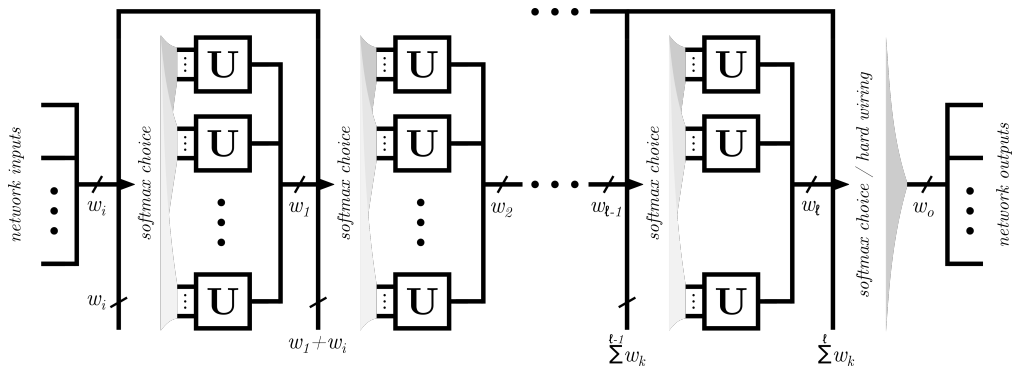

Figure 1: An illustration of the universal unit arrangement (**U**) and softmax-choice wiring as described in Section 2.2. Mimicking residual connections from convolutional neural networks, the outputs of all previous layers of universal units are combined and then offered for connection to the current layer. The output of the last layer is either hardwired or selected by constant attention to yield the outputs of the network.

where $\mathcal{I}{:}n$ denotes the list formed by appending n to $\mathcal{I}$ and $\mathcal{T}\,[\mathcal{I}]$ denotes the indexing the four-axis table tensor $\mathcal{T}$ by the four indices (deterministic; either 0 or 1) listed by $\mathcal{I}$ in order.

LUT-adder block.

The LAB is a LUT combined with a 1-bit full adder, and its output is decided as a learned softmax choice (constant attention) over the two outputs of the adder (including carry) and the one output of the LUT. The adder, which is learnably an option for the output of the LAB unit, is an example of an architectural inductive bias towards addition. Let $\boldsymbol{c}$ be the learned parameter vector for the constant attention $\boldsymbol{\alpha}$, $\boldsymbol{i}$ vector of inputs as above, and let $\boldsymbol{\alpha} := \mathit{softmax}\,(\boldsymbol{c})$. Then

$$\mathrm{LAB}\,(\boldsymbol{i}) := \boldsymbol{\alpha}_1\,(\Sigma \bmod 2) + \boldsymbol{\alpha}_2 \mathbb{1}_{\Sigma\geq 2} + \boldsymbol{\alpha}_3 \mathrm{LUT}\,(\boldsymbol{i}) \qquad \text{for } \Sigma := \boldsymbol{i}_1 + \boldsymbol{i}_2 + \boldsymbol{i}_3,$$

where σ denotes the sum of the carry-in and the two inputs, and $\mathbb{1}_{\text{condition}}$ is the binary indicator function yielding 1 if the condition is satisfied and 0 otherwise.

2.2 Softmax-Choice Wiring

We stack universal units in layers $1 \leq k \leq \ell$ of widths w_k. For simplicity, we refer to the inputs of the network as the 0th layer and have $w_0 := w_i$. Let $\boldsymbol{i}_p^{k,m}$ be the pth input of the mth unit in layer $k \geq 1$ where $1 \leq m \leq w_k$, and let $o^{k,m}$ be the output of the mth unit in layer $k \geq 0$. Then, for each $k \geq 1$ and m, p as above, the pth input of the mth unit in the kth layer is computed as

$$\boldsymbol{i}_p^{k,m} := \sum_{l=0}^{k-1} \sum_{n=1}^{w_l} o^{l,n} \mathit{softmax}\left(\boldsymbol{c}^{k,m,p}\right)_{l,n},$$

where $\boldsymbol{c}^{k,m,p}$ is a learned bias matrix of dimension $k \times \max_{0\leq l<k} w_l$ and the softmax is computed over both dimensions.

In other words, every input to a unit is computed by applying constant attention to the outputs of all the previous layers, input layer included. Finally, outputs of the last layer are designated as the outputs of the network. This can be done by hard-wiring a selection of unit outputs to the w_o network outputs, or by simply adding one more layer of w_o softmax choices. The entire wiring approach is depicted in Figure 1.

2.3 Output Loss and Sharpening Softmax Loss

We use per-signal binary cross entropy loss to compute the loss on the outputs of the network. Further, since we implicitly expect every softmax signal selection (both within units and wiring between units) to eventually rely only on a single output, we added a sharpening entropy loss, controlled by a hyperparameter σ. More specifically, for an n-dimensional softmax choice $\boldsymbol{s}$, we

compute $H(\boldsymbol{s}) = \sum_{m=1}^{n} -\boldsymbol{s}_m \log \boldsymbol{s}_m$. It is the sharpening loss that forces all softmax choices to eventually become 0 or 1, warranting explainability when inspecting the networks after the training concludes. The total training loss for true example outputs $\boldsymbol{o}_\mathrm{T}$ given prediction outputs $\boldsymbol{o}_\mathrm{P}$ is then $\mathit{loss}(\boldsymbol{o}_\mathrm{T}, \boldsymbol{o}_\mathrm{P}) := \mathrm{BCE}(\boldsymbol{o}_\mathrm{T}, \boldsymbol{o}_\mathrm{P}) + \sigma \sum_{\boldsymbol{s} \in S} H(\boldsymbol{s})$, where S is the set of all softmax choices made within the network. Once the training has concluded, the synthesised network can be read out using the procedure described in Appendix B.

3 Evaluation

We evaluate our method on datasets comprising input-output examples for a number of tasks, namely: arithmetic negation, subtraction, addition, multiplication, long division, remainder computation, bitwise and, or, xor, not, bit shifts left and right, line multiplexing, de-multiplexing, decoding, and priority encoding. These were inspired by [14]. Each task is described in detail in Appendix A.

Two training datasets, `EC-2-100` and `EC-4-100`, are formed from the examples for the above tasks where the key inputs or outputs are 2 bits and 4 bits wide, respectively. Four more testing datasets are then generated by leaving out 5% and 10% of the examples for each training dataset. Thus, we consider six datasets in total, four of which test the ability of the given network to generalise already-seen behaviour to previously unseen inputs. We denote these datasets by strings of the form `EC-width-ccc`, where `EC` stands for "elementary circuits", `w` denotes width and is `2` or `4`, and `ccc` (denoting completeness) is `100` (all examples), `95` (5% dropped out), or `90`.

We find that the choice of the universal unit significantly influences the accuracy of the method. AIG and LUT networks have the best and worst accuracy scores, respectively. Moreover and perhaps surprisingly, the best-performing LUT configurations did not respond to the changes to the proportion of training examples being left out. This suggests that the LUT networks may be implicitly internalising the desired functionality in the loss minimisation process, relying on the broader picture of the behaviour rather than individual example pairs.

LUT-adder blocks are the only type of universal unit we considered with explicitly incorporated bias towards the tasks. We observe that their performance is only slightly lower than that of LUTs, likely due to the higher number of learnable parameters, but that it increases with the *decreasing* number of training examples. This can be interpreted as the LABs finding it easier to learn inductive behaviour than its peers, and easier against its own performance when only few key examples are provided.

In sum, the performance of LUTs suggests that even neural networks tailored to a specific purpose show signs of developing an high-level picture of the task, while the performance of LABs shows the utility of architecturally incorporating implicit knowledge about the task at hand for its inductive performance on discrete, logical tasks.

Dataset	Metric Type	Universal Unit Type		
		2-bit AIG	4-bit LUT	4-bit LAB
EC-2-100	signal	*0.956*	***0.969***	0.961
	example	*0.930*	***0.953***	0.938
EC-2-95	signal	0.956	***0.969***	0.964
	example	0.922	***0.953***	0.941
EC-2-90	signal	0.941	***0.969***	***0.969***
	example	0.898	***0.953***	***0.969***
EC-4-100	signal	*0.925*	***0.967***	0.952
	example	*0.778*	***0.889***	0.849
EC-4-95	signal	0.923	***0.967***	0.960
	example	0.735	***0.889***	0.868
EC-4-90	signal	0.922	***0.967***	***0.967***
	example	0.723	***0.889***	***0.889***

Table 1: The results of a systematic evaluation of our method on the datasets. **Emphasis** and *emphasis* mark the best results per dataset and per-unit, respectively. Each experiment was run with 20 different configurations but training and initialisation seeds fixed, as described in Appendix C.

References

[1] Mario R Barbacci. A comparison of register transfer languages for describing computers and digital systems. *IEEE Transactions on Computers*, 100(2):137–150, 1975.

[2] Peter Belcak, Ard Kastrati, Flavio Schenker, and Roger Wattenhofer. Fact: Learning governing abstractions behind integer sequences. *arXiv preprint arXiv:2209.09543*, 2022.

[3] Peter Belcak and Roger Wattenhofer. Periodic extrapolative generalisation in neural networks. *arXiv preprint arXiv:2209.10280*, 2022.

[4] Sina Boroumand, Christos-Savvas Bouganis, and George A Constantinides. Learning boolean circuits from examples for approximate logic synthesis. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 524–529, 2021.

[5] J Richard Buchi and Lawrence H Landweber. Solving sequential conditions by finite-state strategies. In *The Collected Works of J. Richard Büchi*, pages 525–541. Springer, 1990.

[6] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv preprint arXiv:2110.11292*, 2021.

[7] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic*, 28(4), 1963.

[8] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. Deep learning for logic optimization algorithms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2018.

[9] Christopher Hahn, Frederik Schmitt, Jens U Kreber, Markus N Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. *arXiv preprint arXiv:2003.04218*, 2020.

[10] Soha Hassoun and Tsutomu Sasao. *Logic synthesis and verification*, volume 654. Springer Science & Business Media, 2001.

[11] Abdelrahman Hosny, Soheil Hashemi, Mohamed Shalan, and Sherief Reda. Drills: Deep reinforcement learning for logic synthesis. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 581–586. IEEE, 2020.

[12] Shubham Rai, Walter Lau Neto, Yukio Miyasaka, Xinpei Zhang, Mingfei Yu, Qingyang Yi, Masahiro Fujita, Guilherme B Manske, Matheus F Pontes, Leomar S da Rosa, et al. Logic synthesis meets machine learning: Trading exactness for generalization. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1026–1031. IEEE, 2021.

[13] Edwin D Reilly. *Milestones in computer science and information technology*. Greenwood Publishing Group, 2003.

[14] Lior Rokach, Alexander Feldman, Meir Kalech, and Gregory Provan. Machine-learning-based circuit synthesis. In *2012 IEEE 27th Convention of Electrical and Electronics Engineers in Israel*, pages 1–5. IEEE, 2012.

[15] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[16] Frederik Schmitt, Christopher Hahn, Markus N Rabe, and Bernd Finkbeiner. Neural circuit synthesis from specification patterns. *Advances in Neural Information Processing Systems*, 34:15408–15420, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Please see Appendix C.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Please see Appendix C.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Tasks of the `EC-w-100` Datasets

A number of bitwise, arithmetic, and signal control learning tasks form the complete datasets of width `w`. We refer to the individual training instances as tasks, since they specify behaviour that is to be learned from input-output examples. The behaviour of each task is given as a sequence of (x, y) pairs, where x is a binary vector that is to be fed into logic circuit, and y is the binary output representing the desired output for input x. The dimensions of vectors x, y depend on the task and on `w`.

A.1 Bitwise Operations

A.1.1 Bit Negation

The dimension of both x and y is exactly `w`. The input-output pairs for this task give the behaviour of a `w`-bit logical NOT gate. For example, for $x = 1011_2, y = 0100_2$.

A.2 Bitwise AND, OR, and XOR

The dimension of x is 2`w`, the dimension of y is `w`. The input-output pairs describe the behaviour of the `w`-bit AND/OR/XOR gate, performing the operation on the two `w`-bit slices of x. For example, AND $(1011_20110_2) = 0010_2$.

A.3 Unsigned Bit Shift to the Left/Right

The dimension of both x and y is exactly `w`. The input-output pairs are given by shifting x to the left/right by one bit and filling the empty bit with 0. As an example, SHL $(1011_2) = 0110_2$.

A.4 Arithmetic Operations

A.4.1 Arithmetic Negation

The dimension of both x and y is exactly `w`. The task is to perform arithmetic negation (in the two's complement system for bit width `w`). This is equivalent to performing the ones' complement and then adding 1, e.g. NET $(1110_2) = 0010_2$.

A.4.2 Arithmetic Addition and Subtraction

The dimension of x is 2`w`, the dimension of y is `w` $+ 1$. The task is to perform arithmetic addition/negation, with negative numbers represented in the two's complement system of widths `w`, `w` $+ 1$ for inputs and outputs, respectively. The `w` $+ 1$st bit is the carry/borrow bit, respectively. As an example, SUB $(0001_21110_2) = 00011_2$.

A.4.3 Arithmetic Multiplication

Both x and y are of dimension 2`w`. Unsigned arithmetic multiplication is performed on the individual slices of x. For example, MUL $(0011_21110_2) = 00101010_2$ because $3 \times 14 = 42$.

A.4.4 Integral Division and Remainder (Modulo)

The dimension of x is 2`w`, the dimension of y is `w`. The output is the output of unsigned long division (remainder) of the value of the first `w`-bit slice of x by the value of the second. For example DIV $(0110_20011_2) = 0011_2$, REM $(0110_20011_2) = 0001_2$.

A.5 Signal Control

A.5.1 `w`-to-1 Multiplexer

The dimension of x is `w` $+ \lfloor \log_2$ `w` $\rfloor$, the dimension of y is 1. The first `w`-bit slice of x contains the multiplexer's input signals. The remaining slice of x contains the value deciding which of the signals to output, with the lines being numbered from 0. For an example with $w = 5$, to choose the 3rd signal from 01101_2, MUX $(0111_211_2) = 0_2$. The value of the binary logarithm is rounded to avoid having to specify undefined behaviour.

A.5.2 1-to-w Demultiplexer

The dimension of x is $1 + \lfloor \log_2 \mathtt{w} \rfloor$, the dimension of y is $\mathtt{w}$. The value of the slice of x that runs from the second bit onwards describes which output line (numbering from 0) to forward the input signal (stored in the first bit of x) to. The remaining output lines are set to 0. As an example for $w = 4$, DEMUX $(1_2 11_2) = 1000_2$.

A.5.3 $\lfloor \log_2 \mathtt{w} \rfloor$-to-w Line Decoder

The dimension of y is $\mathtt{w}$, the dimension of x is $\lfloor \log_2 \mathtt{w} \rfloor$. The line decoder drives the xth output line to 1 and all the remaining lines to 0, as in $DEC\,(10_2) = 0100_2$ for $\mathtt{w} = 4$.

A.5.4 w-to-$\lceil \log_2 \mathtt{w} \rceil$ Priority Encoder

The dimension of x is $\mathtt{w}$, the dimension of x is $\lceil \log_2 \mathtt{w} \rceil$. The priority encoder takes the $\mathtt{w}$ signals of x as inputs and outputs binary value representing the position (numbered from 0) of the first non-zero line on its input. For an example with $w = 5$, ENC $(00110_2) = 001_2$.

B Explainability – Reading Out the Learned Network Logic

Once the training has concluded and the sharpening loss has converged, one can proceed to read out the synthesised network logic as described below.

Initialisation. If the selector had been used, its softmax choices (now sharp – either 0 or 1 per signal line) are used to find the output units $\mathcal{O}$. If it had not been used, the $\mathcal{O}$ is set to the hard-wired output units.

Wire identification. In line with the notation and definitions of Section 2.2, define the wire presence indicator $\omega\left(U, U', p'\right)$ for U, U' units of the network used for training as

$$\omega\left(U^{k,m}, U^{k',m'}, p'\right) := \begin{cases} 1 & \text{if } \textit{softmax}\left(\boldsymbol{c}^{k',m',p'}\right)_{k,m} > \tau \\ 0 & \text{otherwise} \end{cases},$$

where $\tau \in (0, 1)$ is a threshold for the softmax values for which the wire is to be considered present. In our experimentation, we found that the appropriate threshold depends on the values of σ, learning rate, and the number of epochs ("tightness" of convergence), but could usually comfortably be 0.95 or above.

Put in words, $\omega\left(U_1, U_2, p\right) = 1$ means that the output of U_1 is connected to the p-th input of U_2 in the synthesised network.

Network Extraction. With ω defined as above, the units $\mathcal{U}$ used by the synthesised network can be extracted recursively as follows:

$$\mathcal{U}_0 := \mathcal{O}$$

$$\mathcal{U}_i := \left\{U^{k,m} : \exists k', m', p' \exists k < k' \exists 1 \leq m \leq w_k \text{ s.t. } \omega\left(U^{k,m}, U^{k',m'}, p'\right) = 1\right\} \text{ for } 1 \leq i \leq \ell + 1$$

$$\mathcal{U} := \bigcup_{i=0}^{\ell+1} \mathcal{U}_i$$

Then the pair $(\mathcal{U}, \omega)$ gives the synthesised network.

C Training and Testing Configurations

C.1 Train-Test Splits

Given the nature of the tasks considered, the training and test dataset are largely the same, depending on the situation.

For `EC-2-100` and `EC-4-100`, the two sets are identical. For all other datasets `EC-w-cc`, the networks are trained on `EC-w-100` and `EC-w-100` and then tested on a fixed subset `EC-w-cc`, where is the percentual proportion of the examples of `EC-w-100` that were retained for testing.

C.2 Architecure

For each unit and each task in each dataset, we considered a maximum of 4 layers of 40 units for the network to emerge through softmax choices. Half of our configurations further added softmax selections for the outputs of the final layer as per Section 2.2.

Overall, we observed improvements in performance for AIG networks that had the softmax output selectors but performance deterioriation for LUT and LAB networks in the corresponding configurations.

C.3 Initialisation and Data Shuffle Seeds

These were fixed, with the particular choices detailed in the code.

C.4 Training Parameters and Hyperparameters

A total of 20 configurations was considered for the test runs, and each run was run for a maximum of 100 epochs with early stopping for when the perfect accuracy of 1 has been achieved.

The fixed batch sizes lied in $[4, 16]$, learning rates in $(0.1, 0.6)$, learning rate exponential decay factors in $(0.9, 1.0)$. We used the Adam optimiser.

After initial experimentation, we fixed $\sigma = 1.0$, but engaged it only in the second half of the training (i.e. after 50 epochs).

C.5 Computational Resources

Each of our experiments was run on a single core of a cluster of Dual Deca-Core Intel Xeon E5-2690 v2 processors and fit within 10 GiB of RAM memory. In this setup, all experiments terminated within 12 hours.