Stochastic Optimisation of Lookup Table Networks, for Realtime Inference on Embedded Systems

Chris Kiefer

Experimental Music Technologies Lab,
University of Sussex
c.kiefer@sussex.ac.uk

Abstract. Neural networks running on FPGAs offer great potential for creative applications in realtime audio and sensor processing, but training models to run on these platforms can be challenging. Research in TinyML offers methods for transforming trained neural networks to run on embedded systems. Further gains might be made by training networks directly constructed from lookup tables (LUTs), the basic element of FPGA hardware. A novel method, Stochastic Logic Optimisation, is presented for supervised learning with feed-forward networks of LUTs. The method is found to significantly improve on the use of both a genetic algorithm and memorisation in a beat prediction task.

Keywords: FPGA, machine learning, embedded computing

1 Introduction

When using machine learning for creative work, there is sometimes a need to run models in realtime in resource-constrained embedded systems. Examples might be sensor processing in a hybrid musical instrument, gesture recognition in a wearable augmented reality device, machine listening and sound classification in a sound art installation, or generative control voltage signals in a eurorack module. It can be a challenge to run more complex models in realtime even on conventional computers; for example when machine learning systems rely on parallel processing in GPU pipelines that are naturally optimised for graphics and struggle with faster audio or sensor signals, highlighting a wider need to experiment with alternative non-GPU architectures for realtime work. When working with embedded systems there are often tight memory and data processing constraints, measured in KB rather than GB, and in MHz rather than GHz. There is a need to develop models which are optimised to work with vastly reduced resources compared to those demanded by conventional deep learning techniques. Embedded systems have long been a part of the creative toolbox, with systems such as *Arduino* and *Teensy*. These systems use microcontroller architectures; they are capable of running lightweight machine learning models, but might struggle with more demanding realtime tasks such as inference with realtime audio or high bandwidth sensor signals. An alternative type of computing architecture, the Field Programmable Gate Array (FPGA), is becoming more readily available to non-commercial users, with open source toolchains (Yu, Lee, Lee, Kim, & Lee, 2018) such as Yosys+nextpnr (Shah et al., 2019) and Symbiflow (Murray et al., 2020), and systems such as Lattice's ICE40 (Lattice, 2021). Broadly, FPGAs are becoming more openly accesible to artist and maker communities.

FGPAs are computing systems, reconfigurable at the hardware level, offering basic building blocks from which customised architectures can be constructed. For the purposes of creative machine learning, they offer some distinct advantages over microcontroller boards because of their potential to run massively parallel processes at very high frequencies. This gives them the ability to process high frequency data such as audio in realtime. The major constraint of these systems is that they are more challenging to program and are less user-friendly compared to embedded systems such as *Arduino*. Currently, there is a great deal of research activity around machine learning on FPGAs. This research typically takes the approach of training models with conventional techniques and then applying transformation processes so that the model can by run on an embedded system. This project offers a novel alternative approach, by directly training networks of lookup tables (LUTs), the basic element of FPGA hardware.

2 Machine Learning for FPGAs

Research in TinyML (e.g. Sanchez-Iborra & Skarmeta, 2020) aims to transform machine learning models to run on embedded systems. The field has seen significant successes such as Bonsai (Kumar, Goyal, & Varma, 2017) and XNOR-Net (Rastegari, Ordonez, Redmon, & Farhadi, 2016), which can optimise models to run with very low memory requirements and without the need for a floating point unit. These networks are configured for microcontrollers, but there is also work on FPGAs. Early work in this field explored evolvable hardware (Thompson, 1996). Contemporary work typically focuses on transforming trained models to run within FPGA hardware constraints. LogicNets (Umuroglu, Akhauri, Fraser, & Blott, 2020), LUTNets (Wang, Davis, Cheung, & Constantinides, 2020), Binary Neural Networks (Murovič & Trost, 2019), TiNBiNN (Lemieux et al., 2019) and work on Counterfactual Simulation (Chatterjee & Mishchenko, 2020) and learnable logic (Brudermueller et al., 2020) demonstrate how trained models can be translated to logic circuits.

An alternative approach is to directly train a network of LUTs. This approach is appealing because it might result in more competitive use of logic resources by forgoing high-level abstractions and directly working with LUTs during training. This factor could be important for working with the current generation of chips with open-source toolchains, which tend towards lower levels of computing resources. It could also offer in-situ learning on FPGAs that enable self-reconfiguration.

The basic FPGA building block is the LUT, a configurable logic block typically with 4 or 6 inputs, a single output, and a set of mappings (a truth table) to determine the output for each possible set of inputs. LUTs are linked together with configurable connectivity to form larger logic networks. To facilitate learning with a network of LUTs, the training process must optimise the LUT truth tables, and possibly also the connectivity. In the absence of gradients and weights, a discrete optimisation process would typically be used for training. Chatterjee (2018) suggests an incremental memorisation method for LUT networks. Currently, this is the only documented method for directly training LUT networks, although this was designed as an exploration of generalisation rather than as an optimal training method. Other approaches could be, for example, genetic algorithms (GAs) (e.g. Harvey, 2001) or simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983). As a potential improvement to learning LUT networks with these approaches, a new algorithm, Stochastic Logic Optimisation (SLO) is proposed which works directly on LUT truth tables. The efficacy of SLO is tested in comparison with a GA and Chaterjee's memorisation method, in a sequence prediction task.

3 Stochastic Logic Optimisation

The SLO algorithm for optimising logic networks is now described. Full source code is available from two repositories: (1) A C++ library for running LUT Networks (with Python bindings)[1], (2) A Python notebook with the training algorithm and code for the experiment below[2]

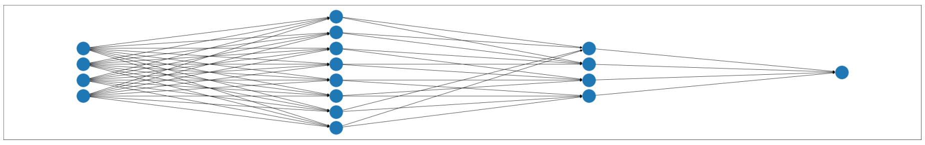

Fig. 1. A network of 4-LUTs, with 4 inputs, 1 output, and two hidden layers

SLO optimises a feed-forward network of LUTs to learn the mappings in training data. SLO uses LUTs with 4 inputs and a single output (a 4-LUT). Figure 1 shows an example of 4-LUTs connected into a network with 4 binary inputs, hidden layers of 8 and 4 LUTS, and a single binary output. To run the network, the values of the LUTs in the input layer are set, and then the LUTs are successively updated layer-by-layer, beginning with the first hidden layer. The values of the LUT truth tables determines the overall function of the network, and the architecture can be adjusted to suit task complexity. Connectivity is limited by the fan-in of the LUTs which can only connect to maximum four

[1] `https://github.com/chriskiefer/liblutnet`
[2] `https://github.com/chriskiefer/SLO_BeatPrediction`

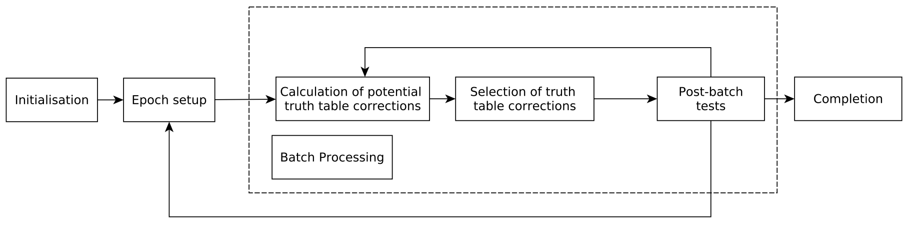
Initialisation
Epoch setup
Calculation of potential truth table corrections
Selection of truth table corrections
Post-batch tests
Completion
Batch Processing

Fig. 2. An overview of the SLO training process

inputs. Connectivity can be configured such that an LUT either shares inputs with other LUTs in the same layer or is independent from them.

Algorithm 1 Batch Processing

```
 1: for all batches do
 2:     $\boldsymbol{allCorrections} \leftarrow []$
 3:     for all training pairs $\boldsymbol{t}$ do
 4:         Calculate the network output based on $\boldsymbol{t}[\boldsymbol{input}]$
 5:         $\boldsymbol{incorrectNodes} \leftarrow$ the incorrect output LUTs in comparison to $\boldsymbol{t}[\boldsymbol{output}]$
 6:         while $layer < trainingLayer$ do
 7:             $\boldsymbol{corrections} \leftarrow$ CalculateLayerCorrections($\boldsymbol{incorrectNodes}$)
 8:             $\boldsymbol{incorrectNodes} \leftarrow$ the LUTs present in $corrections$
 9:             $layer \leftarrow layer - 1$
10:         end while
11:         append($\boldsymbol{allCorrections}, \boldsymbol{corrections}$)
12:     end for
13:     SelectCorrections($\boldsymbol{allCorrections}$)
14: end for
```

3.1 Training

Figure 2 shows an overview of the SLO training process. Each item is now explained in detail. Broadly, the algorithm trains the network by examining errors in the output of the network for a given input, and then choosing truth table entries to change which would ensure the correct output given the same input.

During **initialisation**, beginning with a network of LUTs $\boldsymbol{N}$, the truth tables in the output layer are randomly initialised with linear probability, and the truth tables in hidden layers are initialised sparsely such that a single randomly chosen element in each table is set to 1. During **epoch setup**, training data is randomly shuffled, and split into batches. A target hidden layer $trainingLayer$ is randomly selected for training each epoch, with weighting to give a higher probability of choosing lower layers. Algorithm 1 describes the management of **batch processing**. This process involves the **calculation of potential truth**

Algorithm 2 Search for compatible truth tables

```
function FindCompatibleTruthTables($\boldsymbol{e}$)
    $\boldsymbol{compatibleCombos} \leftarrow []$
    $\boldsymbol{combos}$ = the list of all possible combinations of $\boldsymbol{e}$
    Randomly shuffle $\boldsymbol{combos}$
    for all sets of truth tables $\boldsymbol{c}$ in $\boldsymbol{combos}$ do
        Calculate the values of inputs that correspond to $\boldsymbol{c}$
        if overlapping inputs match then
            append($\boldsymbol{compatibleCombos}$, $\boldsymbol{c}$)
        end if
    end for
    return $\boldsymbol{compatibleCombos}$
end function
```

Algorithm 3 Calculation of potential truth table corrections

```
function InputBitPattern($ttableSet, LUTS$)
    return bit pattern required as input to $LUTS$ that would map to $ttableSet$
end function
function CalculateLayerCorrections($layer$, $\boldsymbol{incorrectNodes}$)
    $\boldsymbol{nodeCorrections} \leftarrow []$
    Split $\boldsymbol{incorrectNodes}$ into clusters $\boldsymbol{G}$, based on shared inputs
    for all $\boldsymbol{cluster}$ in $\boldsymbol{G}$ do
        for all $\boldsymbol{LUTS}$ in $\boldsymbol{cluster}$ do
            Calculate a list $\boldsymbol{e}$ of truth table entries that match the target output
            $ttables$ = FindCompatibleTruthTables($\boldsymbol{e}$)
            $\boldsymbol{inputState} \leftarrow$ the current input values to $\boldsymbol{LUTS}$
            for all Sets of truth tables $\boldsymbol{ttableSet}$ in $\boldsymbol{ttables}$ do
                $\boldsymbol{p} \leftarrow$ InputBitPattern($\boldsymbol{ttableSet}, \boldsymbol{LUTS}$)
                append($\boldsymbol{dists}$, hammingDistance($\boldsymbol{p}, \boldsymbol{inputState}$)),
            end for
            $\boldsymbol{chosenTtableSet} \leftarrow$ the set corresponding to the lowest value of $\boldsymbol{dists}$
            $\boldsymbol{chosenBitPattern} \leftarrow$ InputBitPattern($\boldsymbol{chosenTtableSet}, \boldsymbol{LUTS}$)
            Compare $\boldsymbol{inputState}$ to $\boldsymbol{chosenBitPattern}$
            for all $inputNode$ whose outputs do not match $\boldsymbol{chosenBitPattern}$ do
                $correction \leftarrow$ the truth table entry in $inputNode$ that would match $\boldsymbol{chosenBitPattern}$ if flipped
                append($\boldsymbol{nodeCorrections}, correction$)
            end for
        end for
        return $\boldsymbol{nodeCorrections}$
    end for
end function
```

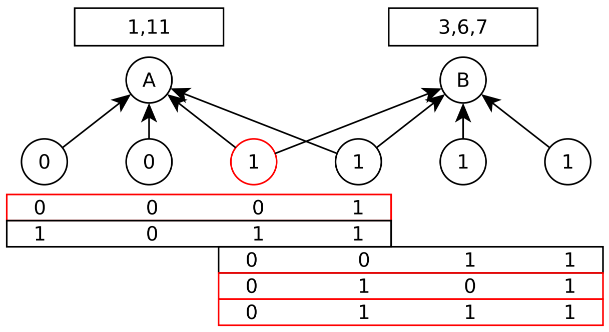

Fig. 3. An example of calculating truth table corrections

table corrections for each training item (algorithms 2 and 3). The core of this process is illustrated in figure 3. There are two LUTs in a cluster (a group of nodes with co-dependent inputs), both with incorrect outputs. It is calculated that truth table entries 1 and 11 in LUT A and 3,6,7 in LUT B would yield the correct output. The system examines the overlapping bit patterns for these entries and finds combinations (A:1, B:6) or (A:1, B:7) as matches, with a bit patterns 000111 and 000101. With the intention of making the smallest adjustment possible, SLO chooses the pattern 000111 as the pattern closest in hamming distance to the current outputs of the LUTs in the lower layer. A potential correction is then logged for the LUT marked in red, such that its truth table would map to a 0 given its current input, and therefore both LUTs in the top layer would yield correct outputs.

A list of potential truth table corrections is generated from all training pairs in the batch. During **selection of truth table corrections**, a frequency table of corrections to all truth table entries is constructed, and the most frequent changes are applied (the top 5% or 10% yield good results). During **Post-batch tests**, if the mean batch error is less than the previous minimum error, then a copy of $\boldsymbol{N}$ is stored. If the required number of epochs has been reached, then this copy of $\boldsymbol{N}$ is returned as the final result.

4 Experiment: Beat Prediction

The performance of SLO was tested in comparison two other methods: (1) Chatterjee (2018)'s memorisation (MEM) method, as a baseline, and (2) a discrete optimisation method; in this case a GA was chosen, future experiments will explore more varieties. A task of beat prediction was chosen, representing the kind of realtime processing this type of network could be suited for. For the purposes of this experiment the network was simulated in Python rather than deployed to an FPGA. The trained model could be used as a live improvisation system that responded to different drumming patterns. This task was designed to examine the efficacy of SLO for supervised learning, rather than exhaustively test the limts of SLO.

4.1 Method

A dataset of drum patterns was hand-programmed. The dataset consisted of 8 different drum patterns, played with the same 5 instruments, repeated for 4 bars each at a resolution of 16 ticks per bar. All data was binary, indicating when a drum was triggered. Models were trained to predict the next pattern of triggers, given the patterns of triggers in the previous 16 ticks. Experiments were run to predict the patterns of source and target instruments in varying combinations, for example to predict the hihat pattern based on the history of the kick drum, or to predict the cymbal and hihat from the snare (full details in the source code). Training data for each experiment was split 80:20 into training and test sets. Experiments were evaluated on classification accuracy.

Both SLO and the GA are stochastic and show some sensitivity to initial random conditions, therefore the best results were chosen over a number of runs and iterations within each run. These quantities, along with meta-parameters, were determined manually to give opportunity to produce the best representative results. Twenty different experiments were run in this manner, as follows:

Phase 1: Training using SLO Training was run 30 times over 300 epochs, and the best result was chosen based on test data score. Each run was initialised with a random choice of layer architecture from the following structures (indicating the number of LUTs in each layer): [128,64,16,4,1] [512,256,64,16,4,1], [2048,1024,256,64,16,4,1]. Where the training set had two outputs, these sizes were doubled. These layers were prepended with an input layer, with size determined by the number of source instruments in the particular experiment. The choice of structure represented a random search for the best architecture.

Phase 2: Training with the GA Meta-parameters were manually optimised for a GA from DEAP library (Fortin, De Rainville, Gardner, Parizeau, & Gagné, 2012): tournament selection from 3 individuals, 0.1 crossover probability, 0.1 mutation probability, 0.05 chance of flipping a bit during mutation, and population size of 50. Individuals were initialised using the same method as for SLO, and were evaluated as networks with identical structure to the best network from phase 1. The best result was chosen from ten runs of the GA with 1000 epochs.

Phase 3: Training with MEM This algorithm is deterministic, with no meta-parameters, so it was simply run once on the training data, using identical architecture to phase 2.

4.2 Results

Figure 4 summarises the results. Plot **A** shows the distribution of test data scores, with SLO scoring better overall. Significance testing was carried out; the data had a non-normal (tested using Shapiro Wilk), symmetrical distribution, with dependent samples, therefore a Wilcoxon signed-rank test was used, showing a significant difference between SLO and the GA *(statistic: 7, p=0.002)*, and

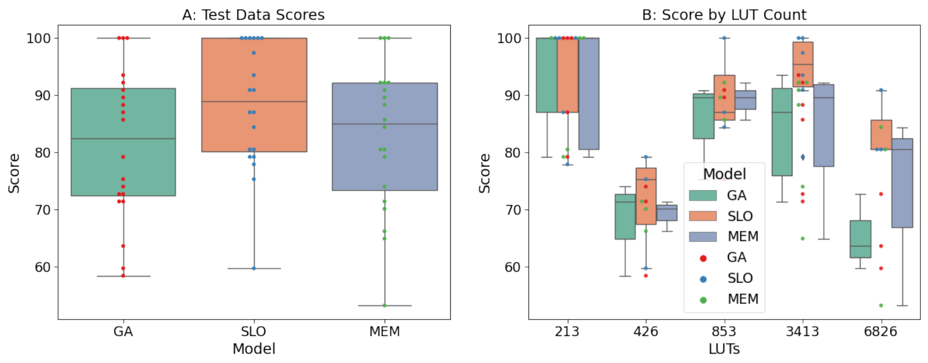

Fig. 4. Results from the beat prediction task, comparing SLO with a GA and MEM

between SLO and MEM *(statistic: 22, p=0.017)*. Plot **B** shows the results for each type of model, grouped by the number of LUTs in the model. For smaller model sizes, the three algorithms had similar performance, while for larger models, SLO outperformed GA and MEM.

5 Conclusions

A new algorithm, SLO, has been presented, which is capable of supervised optimisation of a network of LUTs, with the broader aim of creating lightweight models that can run in realtime on embedded systems. As the resulting models are made from LUTs they can be implemented directly as a logic circuit on an FPGA. For example, on *Lattice ICE* devices, LUTs can be directly specified using the *SB_LUT4* primitive, and a network of LUTs can be implemented with the addition of clocked flip-flops. The models can also run on microcontrollers using the C++ library, with the potential to run at high speed as the model requires only simple arithmetic calculations for memory mappings, and with low memory requirements (at 2 bytes per LUT, the largest network in the above experiment could be stored in approximately 13KB RAM).

The experiment with beat prediction revealed SLO to significantly improve on the GA and memorisation for this task. The results tentatively indicate that the performance of the algorithm is better for larger networks, and further investigation is needed to support this.

This experiment has established the basic efficacy of SLO. Future research will focus on comparison with TinyML systems mentioned earlier, which translate trained neural networks into FPGA logic (e.g. LUTNet), and comparison with other discrete optimisation methods. Current ongoing experiments focus on optimisation with continuous numerical training data, questioning how numbers should be optimally represented. Future work will also investigate creative applications in sound synthesis and machine listening, and time series processing using random boolean network reservoirs (Snyder, Goudarzi, & Teuscher, 2012).

References

Brudermueller, T., Shung, D. L., Laine, L., Stanley, A. J., Laursen, S. B., Dalton, H. R., … Krishnaswamy, S. (2020). Making logic learnable with neural networks. *arXiv preprint arXiv:2002.03847*.

Chatterjee, S. (2018). Learning and memorization. In *International conference on machine learning* (pp. 755–763).

Chatterjee, S., & Mishchenko, A. (2020). Circuit-based intrinsic methods to detect overfitting. In *International conference on machine learning* (pp. 1459–1468).

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., & Gagné, C. (2012, jul). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, *13*, 2171–2175.

Harvey, I. (2001). Artificial evolution: a continuing SAGA. In *International symposium on evolutionary robotics* (pp. 94–109).

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, *220*(4598), 671–680.

Kumar, A., Goyal, S., & Varma, M. (2017). Resource-efficient machine learning in 2 kb ram for the internet of things. In *International conference on machine learning* (pp. 1935–1944).

Lattice. (2021). *iCE40 LP/HX/LM - Lattice Semiconductor.* Retrieved 2021-04-14, from `http://www.latticesemi.com/iCE40`

Lemieux, G. G., Edwards, J., Vandergriendt, J., Severance, A., De Iaco, R., Raouf, A., … Singh, S. (2019). TinBiNN: Tiny binarized neural network overlay in about 5,000 4-luts and 5mw. *arXiv preprint arXiv:1903.06630*.

Murovič, T., & Trost, A. (2019). Massively parallel combinational binary neural networks for edge processing. *Elektrotehniski Vestnik*, *86*(1/2), 47–53.

Murray, K. E., Elgammal, M. A., Betz, V., Ansell, T., Rothman, K., & Comodi, A. (2020). SymbiFlow and VPR: An open-source design flow for commercial and novel FPGAs. *IEEE Micro*, *40*(4), 49–57.

Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision* (pp. 525–542).

Sanchez-Iborra, R., & Skarmeta, A. F. (2020). TinyML-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, *20*(3), 4–18.

Shah, D., Hung, E., Wolf, C., Bazanski, S., Gisselquist, D., & Milanovic, M. (2019). Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial FPGAs. In *2019 ieee 27th annual international symposium on field-programmable custom computing machines (fccm)* (pp. 1–4).

Snyder, D., Goudarzi, A., & Teuscher, C. (2012). Finding optimal random boolean networks for reservoir computing. In *Artificial life conference proceedings 12* (pp. 259–266).

Thompson, A. (1996). An evolved circuit, intrinsic in silicon, entwined with physics. In *International conference on evolvable systems* (pp. 390–405).

Umuroglu, Y., Akhauri, Y., Fraser, N. J., & Blott, M. (2020). LogicNets: Co-designed neural networks and circuits for extreme-throughput applications. In *2020 30th international conference on field-programmable logic and applications (fpl)* (pp. 291–297).

Wang, E., Davis, J. J., Cheung, P. Y., & Constantinides, G. A. (2020). Lutnet: Learning fpga configurations for highly efficient neural network inference. *IEEE Transactions on Computers*, *69*(12), 1795–1808.

Yu, H., Lee, H., Lee, S., Kim, Y., & Lee, H.-M. (2018). Recent advances in FPGA reverse engineering. *Electronics*, *7*(10), 246.