# Predicting Formal Verification Resource Needs (Computation Time and Memory) through Machine Learning

Jason Twigg[1], Erik Torkelson[1] and Nazanin Mansouri[#]

[1]Shiley School of Engineering, University of Portland, OR, USA
[#]Advisor

## ABSTRACT

This paper presents an application of Machine Learning (ML) for estimating the resource requirements (time and memory) for formal verification of digital systems. Formal Verification has become the main bottleneck in the digital design process. As designs grow in size, the verification cost exponentially increases. Some verification efforts fail when the cost of the verification is too large. The most common failures are due to state explosion problem when memory runs out or when verification does not meet time constraints and cannot complete within the given time. For this reason, having an estimation of the computation resources beforehand can be valuable. In this work, a neural network is used to predict the formal verification cost for a diverse collection of designs. Experiments with a large variety of Verilog benchmarks have been conducted. In each experiment, the specific features of a given design are extracted, the design is synthesized and formally verified, and the verification time and memory usage are recorded. The neural network then uses this data to learn the correlation between design features and verification cost. A genetic algorithm experimented with different neural network designs to determine an appropriate framework. The experimental results confirm that the developed neural network can predict the resource requirements and memory usage with reasonable accuracy.

## Introduction

Formal Verification is an integral part of most digital design processes. Advances in the field, and the availability of more computing power and memory have made the verification of significantly more complex designs possible. However, as designs grow over time, the required verification resources grow exponentially, and the state explosion problem remains one of the main challenges. Hence, it is becoming critical to predict the verification cost in advance to avoid losing precious time and scarce resources due to failure. The ideas presented in this paper are of great value for this reason and can be used to predict the required verification resources with reasonable accuracy prior to the verification effort, so that computing power may be budgeted and allocated accordingly. While this has not been possible through conventional design automation practices, the application of machine learning (ML) has the hope of offering a solution to this problem.

In what follows we present a discussion on how machine learning was applied to estimate the formal verification cost. A neural network learns the relationship between input (design features) and output (verification time and verification memory usage) values during training so that it can predict the output given a set of inputs later. A collection of Verilog benchmarks is synthesized and corresponding netlists are generated for each design. Specific design features that are considered important (such as the number of different types of gates, wire count, number of inputs, ...) are extracted from the benchmarks. The input values to the neural network correspond to the extracted data from the synthesized netlists. The output data is collected from the formal verification tool after verification of the designs. It consists of the time and memory usage during the verification process. Using this input and output training data, the

neural network determines the correlation between the features of a design and the verification cost. This neural network can then be used to predict the time taken and memory requirement for the future verification processes with reasonable accuracy. In what follows we will discuss the details of our approach and our experimental set up.

## Hypothesis

Neural Networks are unparalleled at determining patterns in data where they exist, even if the patterns present are extremely abstract and incomprehensible. It would be sensible to apply such a technology to any field where predicting something from given data will be useful. The time required and memory used to verify a hardware design is a perfect example of a useful prediction. A correctly designed neural network should be able to predict those variables with reasonable accuracy given an adequate data sample.

### Related Work

In recent years, several researchers in the field of Electronic Design Automation (EDA) have considered the promise of ML, particularly for solving logic and physical synthesis problems. However, at the time of writing this paper, not much research exists that parallels our work to address the problem at hand.

In [1] El Mandouh and Wassal explain a process for predicting the time it takes for formal verification to succeed. Linear regression machine learning was used to develop this prediction model. In contrast to their approach, this paper proposes using a neural network designed by a genetic algorithm which provides the ability to learn more abstract features than a differentiable function would allow. This method was chosen in hope that the performance of the model would improve with the added flexibility to learn more complex and high-level patterns.

In [2] the authors propose the utilization of supervised machine learning classification techniques to guide the orchestration step by predicting the formal engines that should be assigned to a design property. In comparison to our work that targets resource estimation, they focus on scheduling of the formal engines to optimize the resource usage.

In [3], the authors describe the issues of starting formal verification after the register-transfer level (RTL) description. The authors proposed a new approach to identify hard-to-verify logic early. While this method wasn't used in this paper, the methodology and description of the issues with formal verification provided much a much needed base for this paper.

The authors in [4] present a genetic algorithm method that evolves neural network architectures for specific tasks. This concept is used in this paper to design a neural network model for resource estimation that we believe outperforms a default model.

## Methods

Machine learning and neural networks in particular require a substantial amount of training data to work well with new unseen data. Using machine learning in the area of digital design verification requires thousands of design examples (behavioral Verilog) to develop a network that will function properly. In this research, an assortment of standard Verilog benchmarks are used as training data for the neural network. This training data contains different types of designs that prevent the network from over-learning. If all the hardware designs had similar attributes that are extracted to create training data, the network would probably learn these relationships quicker, but it wouldn't be useful to learn since it wouldn't apply to any other slightly different designs. Therefore, choosing designs that are fundamentally different by covering many of the possible design methods and types of logic implementations will allow for a more general relationship between designs and verification cost to be learned.

## Technical Approach

This section describes our technical approach and experiments that were conducted in several steps:

### *Benchmark Collection:*

A variety of Verilog design benchmarks taken from the open source resources available online were used as training data. Benchmarks are open source Verilog models of digital circuits provided by companies, individuals and universities for research and development use. They allow a more meaningful comparison of results, as experiments are performed on the same standard designs. Benchmarks are great for training as they provide various design types and cover most of the possible uses of Verilog in the design process. The benchmarks we used come from different sources so they were designed in different ways and for widely different applications. On the downside, finding a multitude of benchmarks proved to be a struggle. Some needed to be modified slightly to work with the synthesis tools and methods used in this research.

### *Design Generation:*

Even though this exercise did not succeed in providing us with more design examples, it proved to be valuable as a learning experience. To supplement the benchmarks, a script in python was developed to automatically generate very large design examples. It took master design files, changed parameters and variables, and merged design files. This method created thousands of designs rather quickly. The disadvantage of using this approach was that the designs had similar characteristics despite efforts to increase variety, and did improve learning. Consequently, we opted for running the experiments without the generated designs. This method was ceased so that only more realistic designs would be used in the model's learning process.

### *Synthesis:*

The Synopsys Suite's Design Compiler [5], was used for the synthesis process in this research. Synthesis generates low level implementation of digital designs (as netlists) from their specifications given as high-level descriptions (e.g. behavioral Verilog). Netlists consists of gates and their interconnection (wires) in this implementation of the design. The synthesis process itself can be quite time consuming similar to verification. Given a high-level design as specification, a library of resources and a set of constraints, the tool must create a netlist (implementation at a lower level) from the components in the synthesis library while satisfying the constraints. This netlist is very important for the following verification process.

### *Verification:*

Verification determines if two different implementations of a digital design have the same functionality. Many formal and simulation-based tools are used for functional verification. In this work, we used Formality, the formal verification tool from Synopsys [6]. Verification is critical in ensuring that the high-level designs have been correctly implemented at the logic level and lower. The process of verification becomes very resource intensive as designs grow in size and complexity. In this project, the results of verification along with how long it took and the memory it used are written to an output log to be extracted as data for the ML engine in the next step.

### *Data Extraction:*

This step in the process analyzes the output logs from formal verification. The files are parsed to acquire the time spent and memory used for verification of each design automatically in a python script. The extraction script reads in the output logs from the verification and saves them as data for the model to be trained on. Then testing and training is split apart and kept separate for accurate testing.

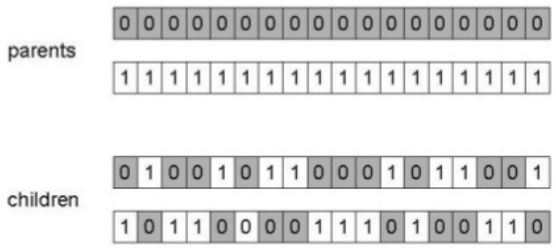
In addition, various elements of each design are measured and extracted to be used as input training and testing data for the neural network. The elements are extracted from the synthesized netlist file. Each input is a summation of the following features from the netlist: inputs, outputs, wires, and all used gates and digital elements. The similar gates were grouped into similar categories, based on their function and number of inputs. This grouping ensured that there was a reasonably small number of inputs for the model to train on.

### *Machine Learning:*

The machine learning library, Tensorflow [7] was used in this research. Although a higher level API for deep learning built on top of Tensorflow called TFLearn [8] was more directly used. TFLearn provided the framework to design a neural network that could run on a GPU using CUDA. This allowed for more focus to go into the data itself and the neural network design. The neural network was designed in python and training was completed using the data extracted earlier. This data is used to teach the neural network how to predict the correct verification time and memory.

## Genetic Algorithm Details

A genetic algorithm is a useful tool to solve optimization problems. They are part of the super-set of evolutionary algorithms. Genetic algorithms attempt to simulate the process of natural selection that takes place in the real world for different species. They work by artificially simulating chromosomal crossover, the mutation of genes, and the selection of which species survive to the next generation (Figure 1). Through this simulation, a population of chromosomes that have ever increasing fitness scores and different designs is created over many generations. Ideally the genetic algorithm will determine good combinations of genes that are a decent solution to the given optimization problem.




A genetic algorithm is used in this research to determine good parameters for the neural network design. Neural networks have many different features that all affect how they will function. These need to be chosen with care so that the network model will perform well for a specific application.

A genetic algorithm would be able to optimize the neural network to perform well with the given data and regression task. It has the ability to determine the following items in the neural network model:

- Number of Layers
- Number of Neurons
- Dropout Percentage
- Activation Function
- Optimization Algorithm
- Learning Rate

**Table 1** which follows shows some of the key features in the design of the genetic algorithm. The population size is the total number of chromosomes tested in each generation. This means there are 40 differently designed neural

networks being tested each generation. A generation is a round of crossover, mutation, and selection of the next population. The fitness of each chromosome was calculated by taking the average percent error during two tests of its design. The inverse of each percent error was taken and then normalized with the rest of the scores. Selection was based on these fitness scores and each score directly corresponded to the percent chance of that chromosome becoming a parent for that spot in the parent pool.

Overall the genetic algorithm is able to create an optimized neural network that should perform exceptionally with the given data.

**Table 1.** Genetic Algorithm Details

| | |
|---|---|
| Population Size | 40 |
| Number of Generations | 10 |
| Starting Mutation Rate | 10% |
| Mutation Rate Decay | 15% per Generation |
| Crossover | Uniform Crossover |
| Fitness Function | Averaged Error Rate |

## Neural Network Details

The neural network was designed using the Tensorflow and TFLearn library and API in Python. The data used in the neural network is read from the output files of the previous scripts and organized into training and testing data. Out of all the data collected, 90% is used for training and 10% is used for testing. Out of the training data, another 10% is used for validation to make sure no over-fitting is taking place during the training process. The testing data is used to check the model with unseen data and check its performance once training is completed. A description of the neural network originally designed follows in Table 2.

**Table 2.** Neural Network Originally Designed

| | |
|---|---|
| Input Layer Size | 22 |
| Number of Hidden Layers | 4 |
| Output Layer Size | 1 |
| Number of Nodes per Layer | 64 |
| Activation Function | Rectified Linear Unit |
| Regularizer | L2 |
| Dropout Percentage | 20% |
| Output Function | Linear |
| Optimization Algorithm | Adam |
| Learning Rate | 0.001 |

The second iteration of testing used a neural network designed by the genetic algorithm described earlier. The model of that neural network follows in Table 3.

**Table 3.** Neural Network Designed by Genetic Algorithm

| | |
|---|---|
| Input Layer Size | 22 |
| Number of Hidden Layers | 3 |
| Output Layer Size | 1 |
| Number of Nodes per Layer | 256 |
| Activation Function | Exponential Linear Unit |
| Regularizer | L2 |
| Dropout Percentage | 20% |
| Output Function | Linear |
| Optimization Algorithm | RMSProp |
| Learning Rate | 0.00005 |

Two copies of the same neural network model are used for the time and memory calculations to give better accuracy for each. If just one network with two outputs was used, the accuracy would be reduced because each did not correlate directly. The time and memory taken rely on some similar aspects but ultimately correspond to different areas.

Figure 2 below shows an example neural network with two hidden layers, three inputs and a single output. This is different from the network used but the image helps visualize how it functions.

The neural network calculates the output based on the input values. Each arrow shown in Figure 2 has an associated weight which is a value that the network changes as it learns. In fact, the machine learning process really is just the network changing all the different weights so that the outputs it predicts improve over time. This is done with a back propagation algorithm. The input to each node in a layer is just the weight of the incoming arrow multiplied by the output of the node it came from. This is simplified to matrix multiplication which is why it is so simple and fast to use a neural network after training has taken place. One just multiplies the inputs by weights and passes the values into the corresponding activation functions.

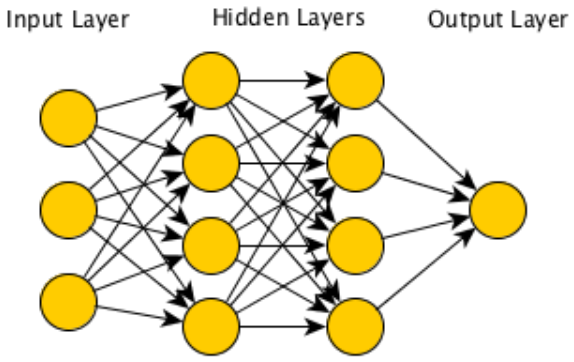



**Figure 2.** Neural Network Example Image

## Results

Overall, the methodology described worked well. Once the training data is collected, the automated synthesis process can begin. Although this work is technology independent, for a practical approach, the computer to create the input data and run the genetic algorithm had the following specs. Intel(R) Core(N) i7-7700HQ CPU @ 2.80 GHz, 16 GB of ram. With the benchmark sized described in this paper, the training of the neural network can be done on any

modern personal computer in under an hour. The original neural network, after 20,000 epochs, has a 20.92% average error on time used and a 2.5% average error on memory usage. The neural network predicted the time required for verification on the testing data with a lower 14.22% error and memory usage with 1.57% error after 20,000 epochs of training.

## Discussion

The error presented in the results was based on testing data which the neural network had never seen before. The testing data was intentionally removed from the overall data sample before training. This allowed for accurate testing after the network had finished learning to prove that it actually functioned properly. The implications of these findings are quite great.

This same machine learning process could be used to predict other useful data such as the time taken and memory used during synthesis, power usage of a design, layout size and transistor count to name a few. Anytime there is a feature that has some correlation to the design, machine learning could be used to predict information about that feature.

Our hypothesis was proved correct since the neural network was able to learn a correlation between the design and verification time. Error percentages below 5% are low enough that one could say that the neural network did in fact learn.

It should be cautioned that these results were for a smaller data set than originally hoped. Ideally, the neural network training data would have consisted of 10,000+ different designs and corresponding verification cost information. However, it is hard to have access to such a large bank of benchmarks in an academic setting, and as discussed earlier, automatic generation of designs for training proved to be unsuccessful. There may have been some over-learning present with such a small data sample of only 500 benchmark files. Precautions were taken to avoid this and it was still able to work with new data that the network had never seen before.

## Future Directions

There are many possibilities to improve this process. This is just one of the many uses of machine learning to assist with automating the design process. Machine learning has great potential to drastically make the design process quicker and more efficient.

In future work, different parts of the design process can be automated by machine learning. This includes, but is not limited to, formal verification and synthesis. Instead of estimating the amount of resources needed for verification, machine learning can fight the huge increase in resources needed to verify designs. Machine learning can help identify red flags in design verification and speed up current software programs. This concept can also be used to help with synthesis of Verilog code. With machine learning, synthesis tools have the potential to simplify designs in ways that was not imaginable before.

Improvements can also be made to this specific method for predicting verification cost. Neural network models can be improved with a larger number of more diverse training data. Having access to more designs would improve the accuracy of this model. There is also opportunity for improvement in the selection of input data. Different input data could have a drastic effect on the neural network. Extracting more design features and taking out unnecessary ones could help the efficiency and result of the network.

Currently, we are also experimenting with applications of machine learning for solving other digital synthesis and optimization problems.

## References


[1] A. G. W. Eman M. Elmandouh, "Estimation of Formal Verification Cost Using Regressing Machine Learning," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Santa Cruz, CA, 2016.

[2] A. G. W. Eman M. Elmandouh, "Guiding Formal Verification Orchestration Using Machine Learning Methods," in *ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 23 Issue 5*, 2018.

[3] M. T. M. M. D. R. D. J. G. a. D. E. S. C Richard Ho, "Early formal verification of conditional coverage points to indentify intrinsically hard-to-verify logic," in *45th Annual Design Automation Conference*, 2008.

[4] G. F. M. a. P. M. T. a. S. U. Hegde, "Designing Neural Networks using Genetic Algorithms," in *International Conference on Genetic Algorithms*, 1989.

[5] "Synopsys Design Compiler: RTL Synthesis," in *https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html*.

[6] "Synopsys Formality," in *https://www.synopsys.com/implementation-and-signoff/signoff/formality.html*.

[7] "Tensorflow: An Open Source Machine Learning Framework for Everyone," in *https://www.tensorflow.org/*.

[8] "TFLearn: Deep learning library featuring a higher-level API for TensorFlow," in *http://tflearn.org/*.

[9] T. Parr, "The Definitive ANTLR 4 Reference", Pragmatic Bookshelf, 2013.

[10] D. K. D. a. R. Sanyal, "Semi-automatic generation of UML models from natural language requirements," in *ISEC '11: Proceedings of the 4th India Software Engineering Conference*, February 2011 .

[11] B. T. a. R. K. Hammond Pearce, "DAVE: Deriving Automatically Verilog from English," in *MLCAD '20: Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, November 2020.

[12] C. B. H. a. I. G. Harris, "GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.