Finding Effective Simulation Patterns for Coverage–Driven Verification Using Deep Learning

Mami Miyamoto

Interdisciplinary Graduate School of
Science and Engineering
Shimane University
Matsue, Shimane, 690-8504 JAPAN
e-mail : s169513@matsu.shimane-u.ac.jp

Kiyoharu Hamaguchi

Interdisciplinary Graduate School of
Science and Engineering
Shimane University
Matsue, Shimane, 690-8504 JAPAN
e-mail : hama@cis.shimane-u.ac.jp

Abstract - Verification of RT/gate-level designs has been a long-standing bottleneck in the process of hardware design. Mainly simulation-based verification has been used for this purpose, and recently, coverage-driven verification has been used. In coverage-driven verification, it is important that each cover point is covered with fairly a large number of input patterns. Therefore, generating many new simulation patterns, in particular, for hard-to-cover points is necessary. We propose a method to learn features of simulation patterns by machine learning, that is, *deep learning,* and to find simulation patterns covering a certain cover point which is hard to be covered, based on *reconstruction errors*. The experimental results show that the proposed method is efficient in finding effective simulation patterns.

I. Introduction

Verification of RT/gate-level designs has been a long-standing bottleneck in the process of hardware design. There are two types of verification methods, which are formal verification and simulation-based verification.

In simulation-based verification, simulation patterns are prepared manually or automatically. Automatic simulation pattern generation is desirable because manual simulation pattern generation takes a high cost. However, simple random pattern generation cannot verify rare corner cases effectively. In simulation-based verification, we use a metric called coverage that is a measure of progress of verification.

Coverage-driven verification aims improvement of coverage. It performs simulation first, and then it generates input patterns that are expected to improve coverage based on the result of coverage analysis, and repeat these processes. In coverage-driven verification, it is important that each cover point is covered with fairly a large number of input patterns. Therefore, generating many new simulation patterns, in particular, for hard-to-cover points is necessary. In this paper, we propose a method that learns features of simulation patterns by machine learning, that is, deep learning, and finds effective new simulation patterns among randomly generated patterns.

Deep learning is a machine learning technique using multilayer neural networks, which has been attracting attention recently. It has been applied to image recognition [1], speech recognition [3] and so on, and it shows higher performance than the other machine learning techniques. We apply it to find effective input patterns for coverage-driven verification.

Given an input vector, a neural network propagates values from the input nodes to the output nodes, depending on the parameters such as weights and biases given to nodes in the network. We consider training vectors, which is usually composed of an input vector and a target vector we obtain as an observation. Given a set of training vectors, that is, a training set, learning a network means tuning the parameters so that the network fits the training set most under some score such as log-likelihood. In deep learning, we perform "pre-training" to find better parameters. When we use a pre-training method based on Restricted Boltzmann Machine (RBM) [4,11], we can learn abstract features of a training set. More importantly, given an input vector, RBM-based approach can give so called a "reconstruction vector". The input vector can be propagated toward the output layers of the network, and then, can be propagated backward to the input layer, which results in a reconstructed vector. The difference between the original vector and the reconstructed vector is the reconstruction error.

We use this deep learning method, in particular, pre-training and reconstruction errors to generate input patterns in coverage-driven verification, in order to cover hard-to-cover points more frequently. First, we perform simulation once with a number of random patterns and learn the features of the simulation patterns covering the target cover points using deep learning in advance. Next, we generate simulation patterns randomly and we give the patterns to the multilayer neural network. Then, we calculate their reconstruction errors and determine the ranking of the patterns based on the reconstruction errors. Finally, we select the patterns with ranks higher than a given threshold, and use them for simulation. Then, we can expect that hard-to-cover points can be covered more frequently and more quickly than simple random patterns. We implemented the above proposed method and investigated its effectiveness through experiments.

II. Related Works and Contribution

There are many studies on application of machine learning techniques for coverage-driven verification systems. The relations between the simulation patterns and coverage points are learned using Bayesian networks in [7,10], inductive logic programming in [2] and Markov models in [5]. In these approaches, based on the learned results, we can generate input patterns effective to cover target cover points. The above approaches, however, require to specify manually which features such as types of instructions or signal transition probabilities should be learned. In other words, these approaches cannot learn input patterns directly.

On the other hand, SAT-based approaches [8,9] do not require to give features explicitly, but the computational costs are high, because it must apply SAT solvers to designs represented as boolean expressions. In [8], only combinational circuits were handled. In [9], heuristic approaches specific to microprocessors were introduced.

The contributions of the method we propose are as follows:

1. It does not require to specify features to be focused on at learning. It can handle input patterns directly. This means our method requires less of manual work.
2. It does not deal with boolean expressions, but only input patterns and their corresponding coverage. This implies our method can scale better.

III. Deep Learning

A. Deep Learning

Deep learning is a machine learning technique using multilayer neural networks as shown in the Figure 1. Since the number of layers is large, neural networks of this type can extract features hierarchically. Its power of expression is much stronger than classical neural networks of three layers and it can handle more complex data.

The characteristic of recognition using deep learning is to perform extraction of features from training sets by unsupervised learning automatically. This unsupervised learning is called pre-training. By performing pre-training, multilayer neural networks have come to be able to perform learning effectively.

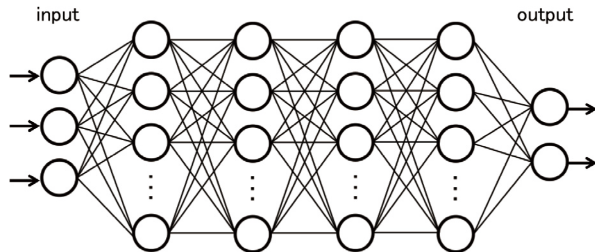


Figure 1. Multilayer Neural Network

Specifically, in pre-training, it learns the parameters for each of succeeding two layers from the input layer. Mainly, RBM (Restricted Boltzmann Machines) [4,11] or Autoencoder [6] is used for the learning of this step. Here, we use RBM in pre-training. In fine-tuning, the network performs supervised learning using the parameter obtained by pre-training. Mainly, backpropagation is used for the learning of this step.

1) Neural Network Architecture

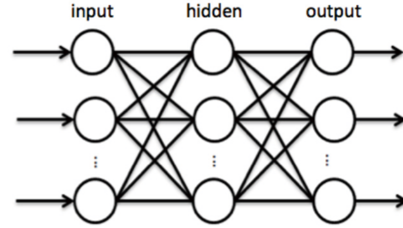


Figure 2. Neural Network

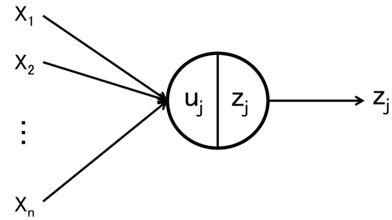


Figure 3. Configuration of unit

Figure 2 shows a classical three-layer neural network underlying deep learning. An input vector is given to the input layer. Figure 3 shows a configuration of each unit. z_j is the output of a unit, and is calculated as follows:

$$u_j = \sum_i W_{ij} x_i + b_j$$
$$z_j = f(u_j),$$

where x_i is an input to a unit, W_{ij} is a connection weight between units, b_j is the bias of a unit. f is called a sigmoid function, which is defined as follows:

$$f(u) = \frac{1}{1 + \exp(-u)}$$

In the learning process, a set of training vectors, or a training set is given, and the parameters are tuned to fit the set under score such as log-likelihood. A training vector is processed one by one, where the parameters of the network are updated for each training vector. Since handling the vector one by one is costly, batch learning is often used, where an update is done for a subset of training vectors, called a batch, instead of each training vector. Number of learnings refers to the number of the updates.

2) Restricted Boltzmann Machine

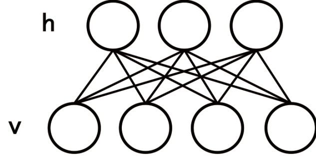


Figure 4. Restricted Boltzmann Machine

In pre-training, each of successive two layers of a multilayer neural network is regarded as an RBM (Restricted Boltzmann Machine), and all of the RBMs are learned from the input layers. Figure 4 shows a configuration of an RBM. An RBM has visible and hidden units, and consists of a matrix of weights $\boldsymbol{W} = (W_{ij})$, where W_{ij} is associated with the connection between visible unit v_i and hidden unit h_j, as well as bias weights b_i for a visible unit and c_j for a hidden unit. Given these, the energy for configuration (v, h) is defined as:

$$E(\mathbf{v}, \mathbf{h}; \theta) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} v_i W_{ij} h_j ,$$

where $\theta = (\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{W})$ is a set of parameters of the network. As in general boltzmann machines, probability distributions over visible and hidden vectors are defined in terms of the energy function:

$$p(\mathbf{v}, \mathbf{h}; \theta) = \frac{\exp(-E(\mathbf{v}, \mathbf{h}; \theta))}{\sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta))} \quad (1)$$

From Eq. 1, we can obtain the following:

$$p(h_j = 1 | \mathbf{v}; \theta) = \sigma(c_j + \textstyle\sum_i v_i W_{ij}) \quad (2)$$

and

$$p(v_i = 1 | \mathbf{h}; \theta) = \sigma(b_i + \textstyle\sum_j W_{ij} h_j) \quad (3) ,$$

where σ denotes the sigmoid function.

In learning of an RBM, input vectors only for visible units are given as a training set. By using Eq. 2 and Eq. 3 alternately, the parameters are adjusted so that they maximize the likelihood of θ.

Update equation of θ is as follows:

$$\theta \leftarrow \theta + \eta \frac{\partial L(\theta)}{\partial \theta} ,$$

where η is called a learning rate, which is the weight of update of θ. $L(\theta)$ is the log-likelihood of marginal distribution $p(v; \theta) = \sum_h p(v, h; \theta)$.

3) Pre-training

The pre-training is performed as follows:

(i) Learn the parameters between the first two layers including the input layer using the training set by RBM.

(ii) Give the value of the hidden units obtained in (i) to the next two layers and perform learning by RBM.

(iii) Repeat (ii) for each layer.

An RBM learns the parameters in order to approximate the distribution of the visible layer to the distribution of the training set. In other words, the parameters are learned so that the hidden layer represents the feature of the training set of visible layer. Therefore, the network can extract features of the training set by performed pre-training.

B. Reconstruction Error

In the proposed method, we perform only pre-training by RBM, without performing fine-tuning. The reconstruction vector represents how the multilayer neural network recognizes the input vector. Therefore, the input vector having a small reconstruction error has the feature that is similar to the training vectors.

A reconstruction vector and a reconstruction error are obtained as follows. After the pre-training is finished, we give an input vector $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ to the multilayer neural network. The value of each unit can be obtained stochastically by forward propagation (Eq. 2). After the input vector reaches the final layer, the value of each unit can be obtained stochastically by backward propagation (Eq. 3).

The vector $\boldsymbol{x'} = (x'_1, x'_2, \cdots, x'_n)$ that is obtained at the input layer is referred to as the reconstruction vector of the input vector $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$. We compare x' and x by the following equation (the reconstruction error).

$$reconstruction\ error = \sqrt[2]{(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + \cdots + (x_n - x'_n)^2}$$

IV. Coverage and Coverage-Driven Verification

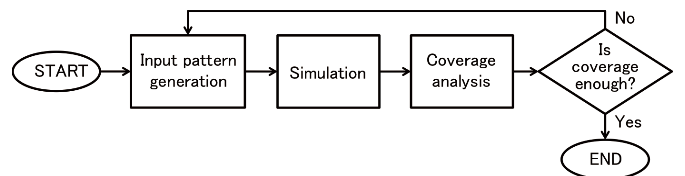


Figure 5. A flow of coverage-driven verification

Coverage-driven verification is one of hardware design verification methods, which aims improvements of coverage. Figure 5 shows a flow of coverage-driven verification. Coverage-driven verification performs simulation first, and then it generates input patterns that are expected to improve

coverage based on the result of coverage analysis, and repeat these processes.

Coverage is a measure for progress of verification. In this study, we use toggle coverage that is one of code coverage metrics. Cover points of toggle coverage are signals in the design, where transitions from 0 to 1 and from 1 to 0 in signals are checked. We distinguish the transition 0 to 1 from the transition 1 to 0, and treat them as distinct corner points. Toggle coverage indicates how far the given design has been activated by simulation.

V. Finding Effective Simulation Patterns

The proposed method does not perform fine-tuning for the neural network and only performs pre-training in deep learning. We perform ranking of the reconstruction errors for randomly generated simulation patterns and find effective simulation patterns.

The flow of the procedure is as follows:

1) Pre-training the network.
   - (i) Choose one cover point that was hard to be covered on the basis of the results of random simulation.
   - (ii) Perform pre-training of a multilayer neural network using the patterns that cover the cover point selected in (i).

2) Ranking input patterns.
   - (i) Generate input patterns randomly.
   - (ii) Give the patterns to the multilayer neural network and calculate their reconstruction errors.
   - (iii) Rank the patterns in an ascending order of their reconstruction errors.

3) Filtering patterns.
   - (i) Filter the random patterns based on threshold of reconstruction errors.
   - (ii) Use the simulation patterns that passed the filter to simulation.

We compress the simulation patterns in order to perform deep learning more quickly and easily. We compressed input vectors of multi-bit to those of one-bit by using only the most significant bits. Then, we performed experiments similar to the next section and compared the case of compressing with the case of not compressing. As a result, there was no big difference between them. Therefore, in this study, we compressed the simulation patterns using only the most significant bits.

VI. Experimental Results

All experiments were performed on an Intel Corei5-4590@3.30GHz with 24GB RAM. We implemented a deep learning algorithm using C language.

A. Experiments

We evaluate the proposed method using the four designs from the IWLS 2005 benchmarks shown in Table 1. #FF is the number of flip-flops and #Logic is the number of logic elements, #CovPnt is the number of all the cover points. We set coverage points at the output of all flip-flops in the designs. We distinguish a transition from 0 to 1 and that from 1 to 0. Therefore, #CovPnt is twice as many as #FF. Table 2 shows the designs, the number of units of each layer, the number of learnings in pre-training and the execution time of deep learning, respectively. In any case, the number of cycles of simulation was 50, the number of layers of the multilayer neural networks was 5, the number of training vectors was 200 and the learning rate was 0.1.

The number of training vectors was decided by an experiment. Specifically, we performed learning of a multilayer neural network by changing the number of training vectors for design of fifo. As a result, the multilayer neural network that has number of units of each layer is 150, 140, 110, 130, 70 can extract features of simulation patterns when the number of training vectors is 200. However, it could not extract features without increasing the number of units of each layer when the number of training vectors is 500. The learning time becomes unrealistically large. Therefore, we used 200 training vectors while considering the execution time. Each of learning update was performed by a batch including 100 training vectors.

Table 1. Design descriptions

DUV	#FF	#Logic	#CovPnt
fifo	283	1463	566
uart	28	181	56
simple_spi_top	132	895	264
usb_phy	98	503	196

Table 2. Learning parameters for experiments

DUV	Number of units of each layer	Number of learnings	Time (s)
fifo	150,140,110,130,70	2000	131
uart	150,180,150,150,60	5000	504
simple_spi_top	300,250,150,150,80	2500	421
usb_phy	300,250,150,150,60	2500	402

B. Ranking patterns

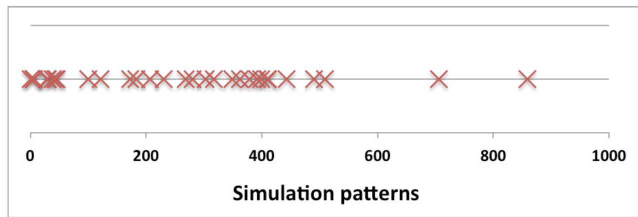
The experimental results are shown in Table 3. Since the experiments have random nature, we show three results for each design. N is the number of patterns that cover each hard-to-cover point among 1,000 different randomly generated patterns. Each pattern consists of signal values for 50 cycles. We chose, as each hard-to-cover point, cover points with fewest coverage among the points covered more than or equal to 10 times in the random simulation. Table 3 shows the number of patterns that cover each hard-to-cover point and the ratio when stopped at half the number of simulations for the random patterns and ranked patterns.

Some experimental results for each design are shown in Figure 6. It shows distribution of patterns that cover the chosen hard-to-cover point in the ranked 1,000 simulation

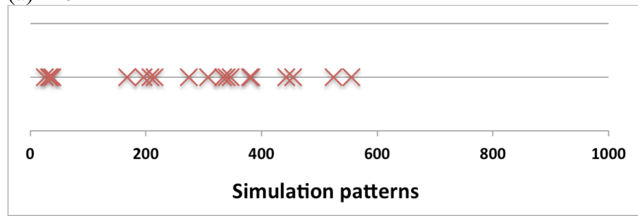
patterns. Table 3 and Figure 6 show that more of patterns covering the target cover points can be found in the top 50% of the ranked simulation patterns.

Table 3. The experimental results

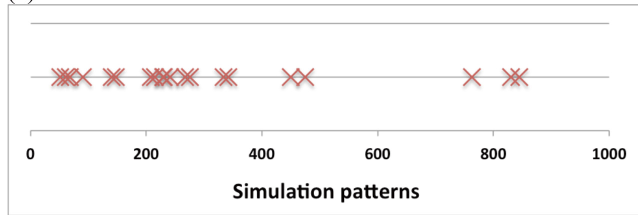
DUV	N	random patterns		ranked patterns	
		included in 50%	ratio	included in 50%	ratio
fifo	30	12	0.400	27	0.900
	29	18	0.621	26	0.897
	18	7	0.389	15	0.833
uart	19	11	0.579	17	0.895
	19	8	0.421	16	0.842
	20	9	0.450	16	0.800
simple_spi_top	20	13	0.650	17	0.850
	24	9	0.375	19	0.792
	26	13	0.500	20	0.769
usb_phy	16	8	0.500	14	0.875
	21	12	0.571	17	0.810
	13	8	0.615	10	0.769



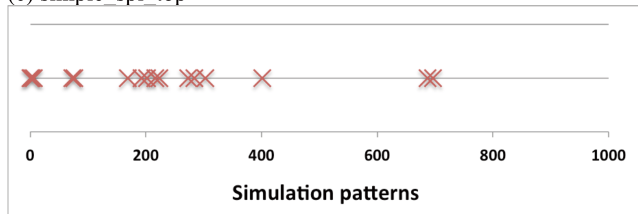
(a) fifo



(b) uart



(c) simple_spi_top



(d) usb_phy

Figure 6. Distribution of input patterns covering hard-to-cover point. × is the simulation pattern that covered the chosen each hard-to-cover point. Patterns closer to 0 have higher ranks.

C. *Filtering patterns*

Based on these experimental results, we performed other experiments in which input patterns are filtered using a threshold value. That is, we set the 500th (50%) and 200th (20%) reconstruction errors as the threshold values and use them as filters. Note that those threshold values can be obtained by calculating reconstruction errors for randomly generated patterns without performing simulation.

We prepared 1,000 filtered input patterns, and 1,000 randomly generated patterns. The experimental results are shown in Table 4. The "filtered patterns" shows how many times each hard-to-cover point is covered when performed the simulation using 1,000 different filtered patterns. The "random patterns" shows how many times each hard-to-cover point is covered when performed the simulation using 1,000 different random patterns. The "filtering time" shows the execution time of the filtering. The "simulation time" shows the execution time of the simulation. "50%" shows the results with 500th reconstruction error as the threshold values for filters. "20%" shows the results with 200th reconstruction error as the threshold values for filters. Similarly, Table 5 shows the results of same experiments that we set the number of patterns to 5,000.

Table 4 and 5 show that the patterns by filtering can cover each hard-to-cover point more than random patterns in general.

Table 4. Results of filtering patterns when the number of simulations is 1,000.

DUV	Filtered patterns		Random patterns	Filtering time (s)		Simulation time (s)
	50%	20%		50%	20%	
fifo	38	48	24	0.57	1.10	125
	46	38	18			
	39	34	18			
uart	35	38	19	0.70	1.42	109
	36	40	12			
	41	40	21			
simple_spi_top	49	72	16	1.09	2.26	129
	45	48	14			
	66	60	14			
usb_phy	29	27	16	1.09	2.17	130
	27	27	14			
	33	28	14			

Table 5. Results of filtering patterns when the number of simulations is 5,000.

DUV	Filtered patterns		Random patterns	Filtering time (s)		Simulation time (s)
	50%	20%		50%	20%	
fifo	171	246	118	2.20	5.28	629
	165	252	107			
	144	240	114			
uart	176	227	111	2.26	6.27	549
	149	229	109			
	135	227	110			
simple_spi_top	225	301	132	4.26	10.60	650
	207	273	124			
	206	286	142			
usb_phy	125	153	78	4.68	10.18	655
	123	164	69			
	128	153	80			

Table 6 and 7 shows the number of the simulations using the random patterns that could perform while performing deep learning + filtering + filtered simulation. Table 6 shows the experimental results using 1,000 filtered simulation patterns. Table 7 shows the experimental results using 5,000 filtered simulation patterns. The "time" is the execution time of deep learning + filtering + filtered simulation. The "number of random patterns" is the number of the random patterns used in the simulation during the "time". The "number of covered" shows how many times each hard-to-cover point is covered when the simulation using the random patterns was performed.

Comparing the "filtered pattern" of Table 4 with the "number of covered" of Table 6, it can be seen that the simulation using the random patterns is more efficient than the proposed method since the deep learning takes time. However, comparing the "filtered pattern" of Table 5 with the "number of covered" of Table 7, it can be seen that the proposed method is more efficient than the simulation using the random patterns. Table 4, 5, 6 and 7 show that the proposed method is more effective when the number of simulations is large since the execution time of deep learning is fixed.

Table 6. Results of the simulation using the random patterns while performing the proposed method using 1,000 filtered patterns.

DUV	Time (s)	Number of random patterns	Number of covered
fifo	257	2111	43
		2101	49
		2079	43
uart	614	5684	129
		5785	131
		5723	133
simple_spi_top	552	4364	82
		4344	120
		4342	117
usb_phy	534	4130	62
		4140	67
		4135	57

Table 7. Results of the simulation using the random patterns while performing the proposed method using 5,000 filtered patterns.

DUV	Time (s)	Number of random patterns	Number of covered
fifo	765	6121	128
		5979	127
		5923	127
uart	1059	9663	216
		9169	211
		9345	215
simple_spi_top	1081	8431	186
		8187	183
		8109	183
usb_phy	1067	7887	107
		8094	114
		7924	117

VII. Conclusions

In this paper, we proposed a method that learns features of simulation patterns by deep learning and finds effective new simulation patterns. The experimental results show that the proposed method is efficient in finding effective simulation patterns. We can expect that hard-to-cover points can be covered more frequently and more quickly than simple random patterns by applying the proposed method to the input patterns generation in coverage-driven verification. The proposed method is more effective when the number of simulations is large since the execution time of deep learning is fixed. Moreover, in case of performing the regression verification, the proposed method is more efficient since the deep learning is performed only once.

In future it is necessary to evaluate the effectiveness of the proposed method in more realistic and larger settings of coverage-driven verification, to try using another machine learning method such as Autoencoder, to improve the execution time of deep learning and to make it possible to choose more than one hard-to-cover point.

References

[1] A. Krizhevsky, I. Sutskever, G. H. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advance in neural information processing systems*, pp.1097-1105, 2012.

[2] C. Ioannides, K. Eder, "Coverage-Directed Test Generation Automated by Machine Learning -- A Review," ACM Trans. on Design Automation of Electronic Systems, vol.17, no. 1, pp. 7:1-7:21, 2012.

[3] G. E. Dahl, D. Yu, Li. Deng, A. Acero, "Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition," *IEEE Trans. Audio, Speech, & Language Process*., Vol.20, No.1, pp.30-42, 2012.

[4] G. E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines," *Momentum*, Vol. 9, No.1, p.926, 2010.

[5] I. Wagner, V. Bertacco, T. Austin, "Microprocessor Verification via Feedback-Adjusted Markov Models," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.26, no.6, pp.1126-1138, 2007.

[6] P. Vincent, H. Larochelle, Y. Bengio, P-A. Manzagol, "Extracting and Composing Robust Features with Denoising Autoencoders," in *Proceedings of the 25th international conference on Machine learning*, pp. 1096-1103, 2008.

[7] S. Fine, A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," Design Automation Conference, pp. 286-291, 2003.

[8] S. M. Plaza, I. L. Markov, V. Bertacco, "Toggle: A Coverage-guided Random Stimulus Generator," Int' l Workshop on Logic Synthesis, pp.351-357, 2007.

[9] S. Shyam, V.Bertacco, "Distance-Guided Hybrid Verification with GUIDO," Proceedings of the conference on Design, automation and test in Europe, p.1211-1216, 2006.

[10] Y. Katz, M. Rimon, A. Ziv and G. Shaked, "Learning microarchitectural behaviors to improve stimuli generation quality," Design Automation Conference, pp. 848-853, 2011.

[11] Y. Bengio, P. Lamblin, D. Popovici and H. Larochelle, "Greedy Layer-Wise Training of Deep Networks," *Advances in neural information processing systems*, Vol. 19, p. 153, 2007.