



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/244438/>

Version: Accepted Version

Proceedings Paper:

WALTER, Andrew, WU, SHIMENG, TYRRELL, ANDY et al. (2026) Fault Detection in Control Software via Program Flow Monitoring using Spiking Neural Networks. In: 25th UK Workshop on Computational Intelligence. 25th UK Workshop on Computational Intelligence, 09-11 Sep 2026 UK Workshop on Computational Intelligence. , GBR.

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:

<https://creativecommons.org/licenses/>

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

Fault Detection in Control Software via Program Flow Monitoring using Spiking Neural Networks

Andrew Walter¹, Shimeng Wu¹, Andy M. Tyrrell²,
Liam McDaid², Malachy McElholm², Nidhin Thandassery Sumithran²,
Jim Harkin², and Martin A. Trefzer¹

¹ School of Physics, Engineering & Technology, University of York

² School of Computing, Engineering and Intelligent Systems, Ulster University

Abstract. Modern machinery depends on embedded software controllers operating reliably and precisely across a wide range of processors. As integrated circuits continue to shrink in scale and grow in complexity, they become increasingly susceptible to faults from a variety of sources. These faults raise the likelihood of controller failure, which can in turn compromise the systems they regulate. This paper presents an approach for early detection of such failures by exploiting the temporal classification capabilities of Spiking Neural Networks (SNNs). The proposed method identifies patterns in processor activity caused by program faults and determines whether a controller is deviating from expected behaviour. To evaluate the approach, we implement a test system consisting of a Proportional–Integral–Derivative (PID) controller regulating a Van der Pol oscillator. Faults are injected at run time by corrupting instructions after they are fetched from memory. Using this setup, we generate a dataset of processor signal recordings captured during both normal and fault-injected execution. These recordings are then converted into spike trains representing the instruction stream. We describe a preliminary SNN-based fault detection method and present experimental results demonstrating its ability to detect faulty controller behaviour within the generated dataset.

Keywords: Control Systems · Spiking neural networks · Fault Detection · Fault Injection · Bio-inspired systems

1 Introduction

Many of the world’s machines depend on reliable and precise electronic control mechanisms. While historically these would have been purpose designed circuits or electro-mechanical systems, modern control systems are overwhelmingly computerised, with a few square centimetres of silicon being given responsibility over expensive industrial machinery, vital infrastructure, or transportation with irreplaceable human cargo [1]. However, as the complexity & ubiquity of such machines increases, and the feature sizes on silicon devices decreases, both the probability and cost of failure grows. The IEEE’s 2021 International Roadmap

for Devices and Systems notes that “the size of a critical defect decreases with scaling. These trends will translate into an increased time spread of the failure distributions and, thus, a decreasing time to first failure” [18]. This is borne out by the difference in failure rates between two studies on Programmable Logic Controllers (PLCs), a very common piece of industrial control machinery; with a survey from 1993 giving 0.0082-0.025 failures per year (0.94-2.85 per million hours) [25], while a reliability analysis from 2010 gives 0.016-0.118 failures per year (1.92-13.49 per million hours) [5]. An increase of between 105-370%.

2 Background

Current runtime fault-tolerance techniques can be divided based on what form of additional resources they leverage (hardware, software, information, time) [28, 8], and what fault handling capabilities they provide (detection, classification, correction, masking); for example the gold standard for safety critical systems, Triple Modular Redundancy (TMR), uses additional hardware to provide fault masking [8].

The majority of hardware fault detection methodologies are based on duplicating some portion of the architecture and making comparisons between its execution & the execution of the original. This includes duplication within components, like latches [17, 21]; duplicates of components, such as with the Razor Flip-Flops [10]; duplication of portions of the processing pipeline, a method used in the Dynamic Implementation Verification Architecture (DIVA) [3]; or duplication of the whole architecture in cases like Duplication with Comparison (DWC) or Dual-Modular Redundancy (DMR) [8].

While these methods have many positive qualities, such as their high reliability and conceptual simplicity, there are a few factors that place limits on their utilisation. The first is the additional hardware needed to implement them, as all versions of this approach require at least twice as many resources due to the duplication; along with the additional hardware needed to perform the comparisons, which scales depending on the desired degree of granularity of said comparisons. For example, more comparators will be needed to compare a processor’s internal signals than would be needed for comparing only the output lines of the same.

The second limitation of these methods is timing, as for the comparison to work the two duplicates must be operating in lockstep; performing not just the same operation within a given clock cycle, but possessing identical (or close to identical) levels of propagation delay so as to ensure that the values being compared come from the equivalent execution steps.

Furthermore, these and many other fault detection methods are noted to have difficulties when it comes to detecting patterns of failure in the temporal domain, commonly manifesting with respect to transient fault scenarios [31]. To address this issue, methods such as Machine Learning (ML) and Artificial Neural Networks (ANNs) have been explored; often using techniques from the field of Time Series Classification (TSC) to predict failures due to phenomena such as tran-

sient faults or hardware ageing [20, 31]. While these approaches show promise, their TSC heritage brings with it complex feature extraction methodologies that reduce practicality.

These factors indicate that a desirable form for future fault detection mechanisms to take would be one that utilises as small a hardware footprint as possible, does not need to operate in lockstep with the system it is protecting, and requires minimal preprocessing of any signals it receives from the system.

2.1 Spiking Neural Networks

Biological organisms possess a wide variety of mechanisms that have been evolved over millions of years to pre-empt, circumvent or otherwise ameliorate failures: from redundant organs, to immune systems [29], to protective reflexes [30, 9]. It is this last case which is of interest in this work, specifically the sophisticated temporal pattern discrimination such reflexes possess. A classic example is found within the escape reflex of crayfish, the sensory neurons of which respond not just to different directions of water movement, but to different frequencies at which the water is moving. These sensory neurons variously excite or inhibit a small number of associated interneurons in such a fashion that temporal patterns in the water’s motion can be discriminated between, allowing for energy intensive escape motions to be triggered only when threatened [32].

This temporal pattern discrimination capability, with minimal alteration of the input prior to processing while operating asynchronously with its environment aligns well with the desirable properties outlined previously. It is therefore advantageous that they have been inherited by artificial Spiking Neural Networks (SNNs) [15]. This paper presents work to apply these capabilities to the problem of detecting faults in computerised control systems by monitoring the sequence of operations being performed. These opcode sequences are highly regular during normal operation, while the introduction of alterations into not only the opcode itself but other parts of the instruction manifest as changes in this operational sequence.

An example controller and plant is detailed, taking the form of a PID controller and Van der Pol oscillator respectively; followed by the description of a method to inject faults into the controller, via modification of the RISC-V processor it is running on. This set up is used to produce a dataset of processor signal recordings, taken during unaltered (Clean) and Faulted operation of the controller, which are translated into spike trains through an event based encoding methodology. Finally, an experiment to train a SNN to differentiate between these recordings is detailed, and directions of future development discussed.

3 Dataset Generation

The example plant and controller chosen for this work consists of a software implementation of a Van der Pol oscillator running on a standard desktop computer, and a PID controller running on an FPGA implementation of a NEORV32

RISC-V Processor [23] (selected as its modular & open source nature made the addition of components for fault injection and signal recording straightforward), with the controller forcing the non-linear Van der Pol oscillator into a circular trajectory. This set up was chosen as PID controllers are a well understood and commonly used form of feedback control mechanism, while the forced Van der Pol oscillator is a control task that is simple to implement but non-trivial to carry out [16, 6].

3.1 Fault Injection & Recording Mechanisms

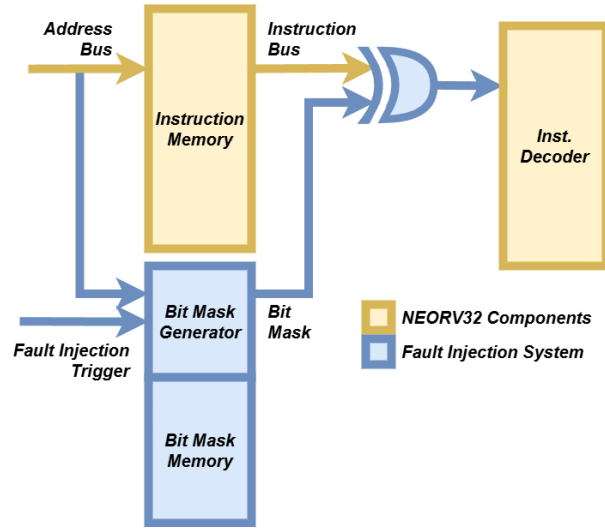


Fig. 1: Block diagram of the fault injection mechanism.

The fault injection mechanism is shown in Figure 1. It consists of an XOR unit interposed between the output of the instruction memory unit (IMEM) and the input of the instruction decoder, as well as a bit mask generator/storage module. During normal operation, the bit mask module passes zeros to the XOR unit, resulting in the instruction coming from the IMEM being unaltered. When fault injection is triggered, the bit mask module randomly selects 10 addresses in the IMEM and generates inversion masks for them. At least one bit will be inverted in each location, with a diminishing probability of further inversions: 10% for a second, 1% for a third, and so on down to a $10^{-29}\%$ chance for all 32 bits in the instruction. When the address being fed to the IMEM has a bit mask in storage, the mask is passed to the XOR unit, resulting in the alteration of the instruction as it passes through to the decoder. While the mechanism has the capability to clear the masks after a period of time (to emulate transient faults), in this case

the bit masks persist until the processor is reset, with further triggering of the fault injection unit adding additional masks rather than replacing existing ones.

Also added to the processor is a Xilinx Integrated Logic Analyser (ILA) [33], which was used to record the signals within the processor during execution. Each record produced is 131,072 time steps long (the maximum memory capacity of the ILA), and consists of 131 channels encompassing: the program counter; instruction memory output; instruction word; post-instruction decoder control signals for the registers, ALU and memory; and the CPU status signals.

3.2 Recording Process & Spike Encoding

To provide a baseline, 500 Clean runs of the plant & controller were carried out, with the ILA recording being triggered a random short duration after program start (5-10 seconds wall clock time). 500 Faulted runs were then performed, with the start of the ILA recording coinciding with the activation of the fault injection mechanism. The Faulted recordings were then filtered to remove any where the injection of the faults did not lead to an error, most likely indicating that the fault injector had altered an instruction that was not executed during the recording period, leaving 407 Faulted runs for a balanced dataset of 814 entries. Due to the constraints of the experimental set up, 500 time step samples were taken from the full 131,072 time step recordings, with the samples being localised around the time step at which a given Faulted recording first showed deviation from its Clean counterpart.

In order for an SNN to make use of the data gathered from the test system, the recordings need to be encoded into spike trains. At each time step in the recording, the value of the instruction word's Opcode (7-bits) is extracted and compared to the list of valid NEORV32 Opcodes. A valid opcode would result in a spike on a channel corresponding to that Opcode, giving 17 channels plus an additional 18 channel for unrecognised opcodes. Such an encoding scheme can be implemented in hardware using a trivial amount of purely combinatorial logic; thus occupying minimal resources while also overcoming a common pitfall of SNNs, complex encoding schemes [2].

4 Fault Detecting Spiking Neural Network

To demonstrate how these opcode occurrence recordings can be used to detect the injected faults, an illustrative SNN based detection system was created using SNN-Torch [14], a Python library that extends the widely used PyTorch neural network library [24].

After performing various initial parameter searches and adjustments to the network topology to ensure sufficiently optimal execution, the parameters shown in table 1 were used (any additional parameters made available by SNN-Torch or PyTorch models are their defaults). The number of inputs (576) is a consequence of applying a sliding window, 32 steps long with a stride of 1, to the spike trains before they were fed to the network.

Table 1: Network and Training Parameters

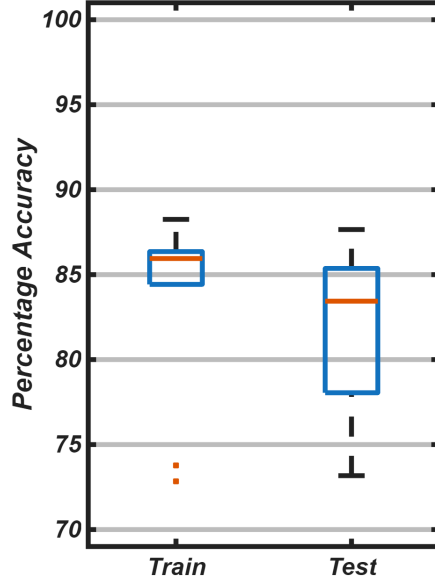
Parameter	Value
Neuron Model	First-order Leaky Integrate and Fire [13]
Decay Rate (β)	0.5
Threshold	1.0
Reset	Zero
Synapse Model	Linear [27]
Weight Range	± 1.0
Initial Weight Distribution	Random Uniform
Bias	False
Inputs	576
Hidden Neurons	256
Output Neurons	2
Epochs	3000
Loss Function	Cross Entropy Spike Count [12]
Accuracy Function	Rate Accuracy [11]
Optimiser	Adam with AMSGrad [26]
Learning Rate	0.0005

The input spike trains were divided into approximately evenly weighted training and testing sets of 732 and 82 individuals respectively; with 10 fold cross-validation ensuring each individual would be represented in a test set.

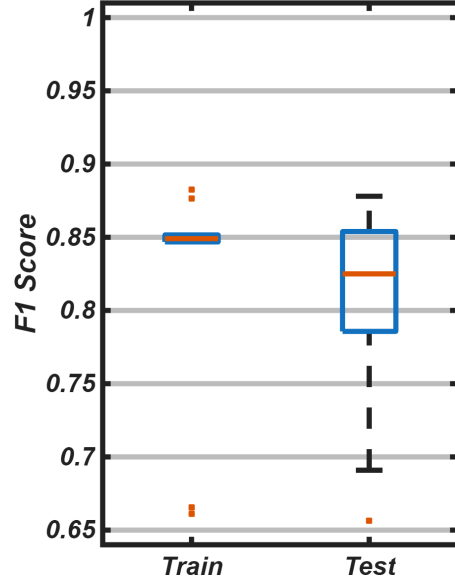
4.1 Results

Figure 2a shows the peak training and test accuracies achieved by the SNN over the 10 folds of the cross-validation. During training the network reached a median accuracy of 85.95% and a maximum of 88.25%; while testing returned a median of 83.44% and a maximum of 87.65%. This pattern of relative performances is reflected in the cross-validation’s F1 scores, shown in Figure 2b, which represents the harmonic mean of the SNN’s precision (How many samples classed as Faulted actually contained Faults) and Recall (How many of the Faulted samples in the dataset were identified as Faulted). The Confusion Matrices associated with these scores are shown in table 2

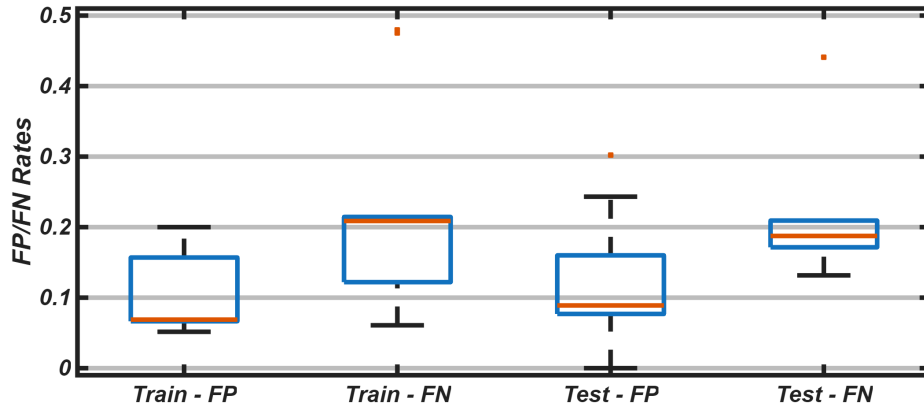
Further analysis of the Confusion Matrices is reflected in the False Positive (FP) and False Negative (FN) Rate values, shown in Figure 2c. While both rates are low across training and test, the higher median FN Rates (0.21 train, 0.19 test) indicate that the performance loss is due to Faulty cases being incorrectly classified as Clean rather than vice versa.



(a) Peak training & test accuracies from the cross-validation runs



(b) Peak training & test F1 Scores from the cross-validation runs



(c) Box plot of the False Positive (FP) and False Negative (FN) rates from the cross-validation runs. The higher median FN rate indicates that the loss of accuracy is due more to Faulted cases being incorrectly identified as Clean cases than vice versa.

Fig. 2: Plots showing (a) Accuracies, (b) F1 Scores, and (c) FP/FN Rates across all folds of the cross-validation runs.

Table 2: Confusion Matrices

			Result			
			Training		Testing	
			Clean	Faulted.	Clean	Faulted.
Fold 0	Target	Clean	323	45	36	3
		Faulted.	41	323	9	34
Fold 1	Target	Clean	296	74	28	9
		Faulted.	22	340	9	36
Fold 2	Target	Clean	301	56	42	8
		Faulted.	58	317	6	26
Fold 3	Target	Clean	349	19	39	0
		Faulted.	173	191	22	21
Fold 4	Target	Clean	337	24	42	4
		Faulted.	79	293	6	29
Fold 5	Target	Clean	340	20	45	2
		Faulted.	179	194	15	19
Fold 6	Target	Clean	349	25	30	3
		Faulted.	77	282	9	39
Fold 7	Target	Clean	346	26	30	5
		Faulted.	74	287	8	38
Fold 8	Target	Clean	344	25	35	3
		Faulted.	78	286	7	36
Fold 9	Target	Clean	296	68	30	13
		Faulted.	45	324	5	33

5 Conclusions

The work presented in this paper makes a compelling case for how faults in computerised control systems can be detected by observing the deviations they cause in the pattern of operation execution, and how this methodology can be implemented through the use of SNNs. In this context, SNNs offer non-lockstepped real-time operation with a potential reduction in fault-detection latency, depending on the choice of decoding scheme and implementation. Furthermore, the experiments have shown that such an approach can be realised using a small number of input channels; an event based spike train encoding; and minimal pre-processing: factors that reduce resource overheads in real deployment scenarios.

However, the architecture used in this paper, while small, needs to be made more compact before deployment in hardware. Several approaches to achieve more compact network topologies [19, 34] and efficient VLSI implementations [4, 22] exist, providing promising direction for future investigation. In addition, the size of the dataset used here (814 samples from only one workload) is comparatively small, something that can put limits on the maximum performance of a neural network model. Enlarging the dataset by collecting more program exe-

cution recordings is therefore also a logical route forward; as would be the use of such an enlarged dataset to carry out comparison experiments with other methodologies such as Time Series Classifiers eluded to previously [31], or alternative neural network models such as Recurrent Neural Networks (RNNs) & Convolutional Neural Networks (CNNs) [7, 35].

References

1. Åström, K.J., Wittenmark, B.: *Computer-Controlled Systems: Theory and Design*, Third Edition. Courier Corporation (2013)
2. Auge, D., Hille, J., Muller, E., Knoll, A.: A survey of encoding techniques for signal processing in spiking neural networks. *Neural Processing Letters* **53**, 4693–4710 (07 2021). <https://doi.org/10.1007/s11063-021-10562-2>
3. Austin, T.: DIVA: a reliable substrate for deep submicron microarchitecture design. MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (1999). <https://doi.org/10.1109/MICRO.1999.809458>
4. Chen, C.Y., Dai, Y.S., Hong, H.C.: A neuromorphic spiking neural network using time-to-first-spike coding scheme and analog computing in low-leakage 8T SRAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **32**(5) (2024). <https://doi.org/10.1109/TVLSI.2024.3368849>
5. Choi, K.C., Hwang, S.J., Jung, T.H.T., Kim, H., Song, S.W.: Reliability analysis for safety grade PLC. Transactions of the Korean Nuclear Society Autumn Meeting 2010 (2010)
6. Cooper, M., Heidlauf, P., Sands, T.: Controlling chaos — forced Van der Pol equation. *Mathematics* (2017). <https://doi.org/10.3390/math5040070>
7. Cruse, H.: Neural Networks as Cybernetic Systems – 2nd and revised edition, chap. 14.1 Elman Nets and Jordan Nets, pp. 146–147. *Brains, Minds Media* (2006)
8. Dubrova, E.: *Fault-Tolerant Design*. Springer (2013)
9. Edwards, D.: Crayfish escape. In: Sherman, S.M. (ed.) *Oxford Research Encyclopedia of Neuroscience*. Oxford University Press (2017). <https://doi.org/10.1093/acrefore/9780190264086.013.158>
10. Ernst, D., Kim, N.S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K., Mudge, T.: Razor: a low-power pipeline based on circuit-level timing speculation. Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. (2003). <https://doi.org/10.1109/MICRO.2003.1253179>
11. Eshraghian, J.K.: `accuracy_rate`. https://snntorch.readthedocs.io/en/latest/snntorch.functional.html#snntorch.functional.acc.accuracy_rate (2021)
12. Eshraghian, J.K.: `ce_count_loss`. https://snntorch.readthedocs.io/en/latest/snntorch.functional.html#snntorch.functional.loss.ce_count_loss (2021)
13. Eshraghian, J.K.: `snn.leaky`. https://snntorch.readthedocs.io/en/latest/snn.neurons_leaky.html (2021)
14. Eshraghian, J.K., Ward, M., Neftci, E., Wang, X., Lenz, G., Dwivedi, G., Benamoun, M., Jeong, D.S., Lu, W.D.: Training spiking neural networks using lessons from deep learning. *Proceedings of the IEEE* **111**(9), 1016–1054 (2023). <https://doi.org/10.1109/JPROC.2023.3308088>

15. Fang, H., Shrestha, A., Qiu, Q.: Multivariate time series classification using spiking neural networks. *Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN)* (2020). <https://doi.org/10.1109/IJCNN48605.2020.9206751>
16. Gan, T., Stepney, S., Trefzer, M.A.: Combining multiple inputs to a delay-line reservoir computer: Control of a forced Van der Pol oscillator system. *Proceedings of the 2023 International Joint Conference on Neural Networks (IJCNN)* (2023). <https://doi.org/10.1109/IJCNN54540.2023.10191630>
17. Hazucha, P., Karnik, T., Walstra, S., Bloechel, B., Tschanz, J., Maiz, J., Soumyanath, K., Dermer, G., Narendra, S., De, V., Borkar, S.: Measurements and analysis of SER tolerant latch in a 90 nm dual-vt CMOS process. *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference*, 2003. (2003)
18. IEEE More Moore Team: International Roadmap for Devices and Systems 2021, International Focus Teams' Roadmap Report: More Moore (2021)
19. Kasabov, N.K.: NeuCube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Networks* **52**, 62–76 (2014). <https://doi.org/10.1016/j.neunet.2014.01.006>
20. Koneru, A., Vijayan, A., Chakrabarty, K., Tahoori, M.B.: Fine-grained aging prediction based on the monitoring of run-time stress using DfT infrastructure. *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2015). <https://doi.org/10.1109/ICCAD.2015.7372549>
21. Lunardini, D., Narasimham, B., Ramachandran, V., Srinivasan, V., Schrimpf, R., Robinson, W.: A performance comparison between hardened-by-design and conventional-design standard cells. *Proceedings of the 2004 Workshop on Radiation Effects on Components and Systems* (2004)
22. Moriya, S., Ishikawa, M., Ono, S., Yamamoto, H., Yuminaka, Y., Horio, Y., Madrenas, J., Sato, S.: Analog VLSI implementation of subthreshold spiking neural networks and its application to reservoir computing. *IEEE Transactions on Circuits and Systems I: Regular Papers* **72**(10) (2025). <https://doi.org/10.1109/TCSI.2025.3550876>
23. Nolting, S., et al.: The NEORV32 RISC-V processor. <https://github.com/stnolting/neorv32> (2021)
24. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E.Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. *Computing Research Repository* (2019)
25. Paula, H.M.: Failure rates for programmable logic controllers (1993). [https://doi.org/10.1016/0951-8320\(93\)90007-L](https://doi.org/10.1016/0951-8320(93)90007-L)
26. PyTorch Contributors: Adam. <https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html> (2023)
27. PyTorch Contributors: Linear. <https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html> (2023)
28. Safari, S., Ansari, M., Khdr, H., Gohari-Nazari, P., Yari-Karin, S., Yeganeh-Khaksar, A., Hessabi, S., Ejlali, A., Henkel, J.: A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues. *IEEE Access* **10** (2022). <https://doi.org/10.1109/ACCESS.2022.3144217>
29. Trefzer, M.A.: Bio-inspired fault-tolerance in electronic systems. In: *Post-Apocalyptic Computing*, pp. 229–258. WORLD SCIENTIFIC (2005). https://doi.org/10.1142/9789811297144_0009

30. Vu, E.T., Berkowitz, A., Krasne, F.B.: Postexcitatory inhibition of the crayfish lateral giant neuron: A mechanism for sensory temporal filtering. *Journal of Neuroscience* (1997). <https://doi.org/10.1523/JNEUROSCI.17-22-08867.1997>
31. Wang, X., Jiang, L., Chakrabarty, K.: LSTM-based analysis of temporally- and spatially-correlated signatures for intermittent fault detection. *Proceedings of the 38th IEEE VLSI Test Symposium (VTS)* (2020). <https://doi.org/10.1109/VTS48691.2020.9107600>
32. Wine, J.J.: The structural basis of an innate behavioural pattern. *Journal of Experimental Biology* **112**(1), 283–319 (09 1984). <https://doi.org/10.1242/jeb.112.1.283>
33. Xilinx Inc.: Integrated logic analyzer (ILA) v2.0 product specification. <https://docs.amd.com/v/u/en-US/ds875-ila> (2012)
34. Yan, S., Meng, Q., Xiao, M., Wang, Y., Lin, Z.: Sampling complex topology structures for spiking neural networks. *Neural Networks* **172**, 106121 (2024). <https://doi.org/10.1016/j.neunet.2024.106121>
35. Zhao, B., Lu, H., Chen, S., Liu, J., Wu, D.: Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics* **28**(1), 162–169 (2017). <https://doi.org/10.21629/JSEE.2017.01.18>