A Novel Approach to Error Resilience in Online Reinforcement Learning

Chandramouli Amarnath and Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: chandamarnath@gatech.edu, abhijit.chatterjee@ece.gatech.edu

***Abstract*—Online reinforcement learning (RL) based systems are being increasingly deployed in a variety of safety-critical applications ranging from drone control to medical robotics. These systems typically use RL onboard rather than relying on remote operation from high-performance datacenters. Due to the dynamic nature of the environments they work in, onboard RL hardware is vulnerable to soft errors from radiation, thermal effects and electrical noise that corrupt the results of computations. Existing approaches to on-line error resilience in machine learning systems have relied on availability of the large training datasets to configure resilience parameters, which is not necessarily feasible for online RL systems. Similarly, other approaches involving specialized hardware or modifications to training algorithms are difficult to implement for onboard RL applications. In contrast, we present a novel error resilience approach for online RL that makes use of *running statistics* collected across the (real-time) RL training process to configure error detection thresholds without the need to access a reference training dataset. In this methodology, *statistical concentration bounds* leveraging running statistics are used to diagnose neuron outputs as erroneous. These erroneous neurons are then set to zero (suppressed). Our approach is compared against the state of the art and validated on several RL algorithms involving the use of multiple concentration bounds on CPU as well as GPU hardware.**

***Index Terms*—Neural Networks, Fault Tolerance, Resilience, Soft Errors, Reinforcement Learning**

I. Introduction and Prior Work

Reinforcement Learning (RL) leveraging Deep Neural Networks (DNNs) is being increasingly deployed in a wide range of fields, including safety-critical applications such as autonomous systems [1]. Such RL systems (e.g. robotic controllers) run on onboard hardware and require a high level of accuracy and reliability for mission performance. However, uncertainties in hardware operation due to the unpredictable nature of the environments these systems work in can induce malfunctions due to soft errors in hardware. Typical causes include radiation, thermal effects, noise due to low voltage hardware and field electrical degradation. To maintain the reliability of such systems in the field, work is needed on low-overhead on-line methods for RL to provide resilience against soft errors that can impact RL system performance.

On-line soft error resilience in standard DNNs has been examined in prior work. As discussed in [2], [3], the use of a parallel DropOut training methodology for neuron outputs or weights with specified failure modes allows for increased DNN error resilience at the cost of increased training time. In [4], the authors alter DNN training to generate invariant relationships between neuron outputs, setting neuron outputs that violate these invariants to zero (error suppression). Later work [5] applied median filtering to neuron outputs to filter out extreme erroneous values, altering DNN training to prevent median filtering from degrading DNN inference. Statistical thresholding of inter-neuron gradients to diagnose erroneous neuron outputs for on-line error suppression without modifying DNN training was explored in [6], but not applied to RL systems.

Error resilience through restriction of output ranges has been examined in [7]. Layer-wise quantization is used, adding a regularization term to DNN loss functions to prevent outlier values from being generated. Dynamic fixed-point quantization of the DNN network and a random variability-aware training methodology for RRAM based DNNs has been examined in [8]. Later work [9] presented Ranger, a method for restricting output ranges of neurons in inference by clamping them to the extrema observed across a fraction (approx. 20%) of the training dataset. As opposed to traditional learning paradigms, the neuron output statistics of RL systems evolve over time with the training the RL system has undergone based on stimulus it has seen and corresponding decisions made. This is different from the training of traditional DNNs and CNNs, which are trained on a fixed dataset with corresponding labels prior to deployment. Hence, error resilience approaches in an RL system need to evolve with the amount of learning performed. *This requires a rethink of prior error resilience approaches, as approaches such as [9] and [4] relying on training statistics or fully trained DNN behavior on training data are unreliable*.

Hardware-level resilience to compute errors and device faults has been examined in [10] through resilience-aware scheduling, running vulnerable computations on more error resilient parts of the hardware. However, this assumes the availability of error-hardened computational resources. Extensions to Algorithm-Based Fault Tolerance (ABFT) [11] to detect errors in convolutional neural network computations [12] and detect and correct errors in matrix multiplications on GPUs [13] allowed fast, accurate detection of errors and correction of

979-8-3503-4135-5/23/$31.00 ©2023 IEEE

a restricted range of errors. Error correction using extensions of ABFT [11] remains a problem due to limitations in the ability to *correct multiple errors with one or a small number of algorithmic checks* without correction being compromised by error aliasing effects. The work of [14] presents a framework for per-layer voltage underscaling to balance error rate and energy use by modulating erroneous neuron computations. Extensions of Algorithmic Noise Tolerance (ANT) to DNNs [15], [16] use low-precision redundant computations to correct for errors, again focusing more on voltage scaling effects rather than soft errors.

Prior reinforcement learning work in the resilience space has to the best of our knowledge, not examined on-line soft error resilience in the RL system itself, focusing mainly on resilience to input and action perturbations. Recent work [17] uses causal learning frameworks to filter out and minimize the effects of input space perturbations, marking perturbed inputs in training with 'interference labels' and allowing the system to estimate the true output label and confounder (perturbation) latent state in inference. Action space perturbations such as actuator attacks in robotics are rectified in [18] through 'robustifying' the trained RL system via adversarial training. Actuator degradation correction through reinforced re-learning is addressed in [19]. The effects of hardware-induced noise in quantum RL is examined in [20], and mitigated in part by altering the training process to reduce the number of measurements needed to update the model. However, this examination focuses on quantum hardware rather than conventional digital systems.

Key Contributions: The key contributions of this work are:

- Our approach provides an on-line error resilience approach that evolves over time with the amount of learning performed in the network, providing a statistical method that evolves with RL network training to provide on-line error resilience without the need for a fixed representative training dataset, unlike earlier methods built for traditional DNNs and CNNs.
- Our approach is a statistically grounded method for thresholding neuron outputs and suppressing erroneous values to zero in online RL systems. It does not require modification of the core RL training algorithm itself. This approach is validated on two benchmarks using different RL approaches for a variety of test cases.

Section II presents preliminaries regarding RL training and inference and an overview of the proposed approach. Section III discusses the proposed approach in detail, while Section IV presents experimental results. We conclude in Section V.

II. Overview

A. Preliminaries: Reinforcement Learning

Reinforcement Learning is commonly framed [21] as an interaction learning paradigm. The RL agent when training interacts with the environment, evaluates the reward from the actions it takes, and uses this to adjust its future actions. The RL agent balances *exploration*, taking random decisions

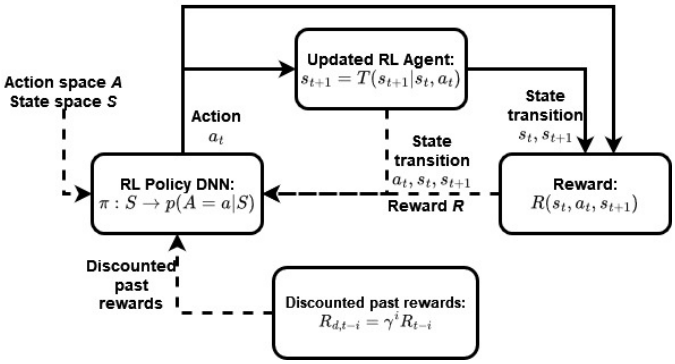


Fig. 1: RL training process overview: Dotted lines indicate the quantities used for updating the weights of the DNN used to generate the RL policy π.

to generate new strategies, and *exploitation*, iterating on past experience to optimize its current strategy. This can be framed as a Markov Decision Process, where the RL agent has a set of *states S* with associated distribution of starting states $p(s)$, a set of possible actions *A* and a state transition function $T(s_{t+1}|s_t, a_t)$ (as shown in Fig. 1) that yields new states s_{t+1} from current states s_t and actions a_t taken at time step t. Furthermore, the RL agent gains a reward $R(s_t, a_t, s_{t+1})$ from its actions taken at the current time step and the following state. The past rewards are modulated by a discount factor $\gamma \in [0, 1]$ (as seen in Fig. 1), which emphasizes more immediate rewards. The RL agent in this manner learns a *policy* π, mapping from states to a distribution of actions $\pi : S \rightarrow p(A = a|S)$. A complete interaction with the environment from an initial to an end state, generating an aggregate reward over the series of states for the RL agent that leads to a policy update is termed an *episode*. The optimal policy maximizing the expected reward function is termed the optimal policy $\pi^* = \text{argmax}_\pi E[R|\pi]$. This policy can be learned or approximated using a Deep Neural Network (DNN). The DNN updates its weights during training based on the reward function to predict the optimal action for each state to maximize expected reward (as in Fig. 1's RL agent). This is done using standard optimizers and a training loss function designed to maximize expected reward.

B. Approach Overview

As shown in Figure 2, the proposed error resilience approach for RL DNNs consists of two major parts: (1) *Statistics collection*, and (2) *Thresholding-based error suppression*. The statistics collected in (1) are used to set periodically-updated flexible statistical thresholds for (2) to enable error resilience. The statistical thresholds are updated after a set number of RL training episodes to adapt to changing RL agent behavior.

In *statistics collection* (Block-1 of Fig. 2), the mean and variance of each neuron output in each layer except the final layer are recorded in an on-line manner during training episodes while the RL system learns the optimal policy. Due to the lack of an explicit, fixed dataset and the need for the RL system to interact with its environment while training, methods such as [9] and [4] are inapplicable. On-line methods are thus

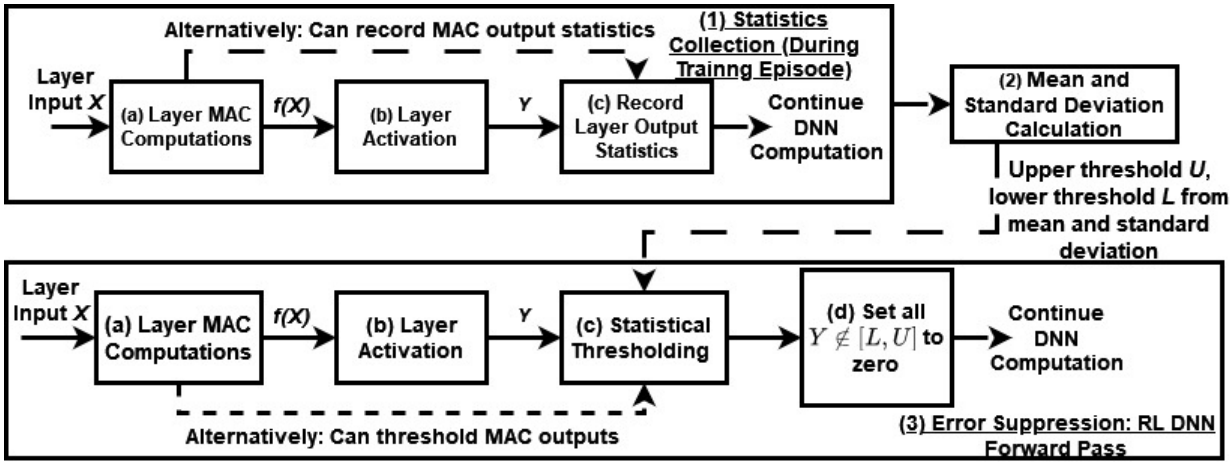


Fig. 2: Flow of statistical threshold-based error resilience in RL Deep Neural Networks. For clarity, statistics collection (Block-1) is shown separately from error suppression during the forward (inference) pass (Block-3).

needed to compute statistical bounds for neuron outputs. This can be done *after* layer MAC operations from the MAC output $f(X)$ (Block 1a), per the dotted line in Block 1 of Fig. 2, or after the layer activation functions from layer outputs Y (Block 1c). These layer output statistics are used to calculate the mean and standard deviation of neuron outputs for each layer after training (Block 2). The process of RL DNN training is unaffected by the statistics collection process.

The mean and standard deviation calculated on completion of training are then used to build an upper and lower statistical threshold for *error suppression* during RL DNN inference (thresholds fed to Block-3c in Fig. 2). These thresholds are based on standard concentration inequalities, allowing adjustment of the threshold sensitivity to detect less severe errors (tight threshold, high sensitivity) or minimize impact on performance from false alarms (low sensitivity, looser threshold). These thresholds are updated as the statistics collected in (1) change, with a user-chosen number of training episodes between threshold updates (using statistics from Block-2 to update Block-3c). Depending on the activation function, different inequalities can be used to set thresholds - for a linear layer two-sided bounds can be used, while for a ReLU activation (zero for all negative inputs), a one-sided (upper) bound is used and the output is lower-bounded at zero. These thresholds are applied to neuron outputs during the network forward pass, and any neuron output violating these thresholds is deemed to be erroneous and set to zero (suppressed). As we see in Fig. 2 Block-3, the thresholding is applied to the same location as the statistics are collected. This can be done after activation (Block 3c) if statistics are collected from layer outputs Y, or after the MAC operations (dotted line from Block 3b) if the statistics are collected from MAC outputs $f(X)$. The proposed zeroing approach cannot be used at the final (output) layer. Due to the output layer's small size we propose to use conventional methods such as TMR [22] for resilience.

The proposed resilience framework is examined in more detail in the following section, beginning with a discussion of DNN computations in Section III-A.

III. Approach Details

A. Neural Network Computations: Overview

RL DNNs considered in this work can use both *dense* and *convolutional* layers. The layer input is denoted by X and the output of the layer's Multiply-Accumulate (MAC) computations is denoted by $f(X)$ (as in Fig. 2). For a dense layer, this is done through weight-bias matrix multiplication such that $f(X) = W.X + b$, where W is the layer weight matrix and b is the layer bias vector.

In a *convolutional* layer, the layer itself consists of C_{out} convolutional neurons (output channels) having C_{in} inputs, so that the layer input tensors have a shape (C_{in}, I_w, I_h) where for each of the C_{in} input channels there is an image of dimension $I_w \times I_h$. The MAC output tensor $f(X)$ thus has dimension (C_{out}, O_w, O_h), where for each of the C_{out} output channels there is an output image of dimension $O_w \times O_h$. Each of the convolutional neurons is associated with a kernel W and a bias b, so that the output of the ith neuron MAC computations is denoted by $f_i(X) = \sum_{k=0}^{C_{in}-1} W_k * X_k + b_i$, where $*$ is the 2-D convolution operator.

DNN activation follows the MAC operations. In this work, two activations are used in the networks examined. Each of them is applied elementwise to $f(X)$ to give the layer output Y as in Fig. 2. The ReLU activation takes the form $y = max(0, x)$, giving a one-sided output. The hyperbolic tangent activation takes the form $y = \frac{e^x - e^{-x}}{e^x + x^{-x}}$, ranging from -1 to 1.

B. On-Line Statistical Threshold Generation and Resilience

During the training of the RL system, we use Welford's online algorithm [23] to collect the mean and standard deviation of neuron outputs. These statistics are collected at the same locations in the network at which thresholding is later performed. Welford's algorithm works in three major steps: (1) First, we initialize the aggregate triple of [sample count (N), mean (μ), M2] as [0,0,0], where sample count refers to the number of outputs seen, the mean refers to the mean of the outputs and M2 denotes the sum of squared squared differences from the mean. In step (2) this aggregate is updated

at each forward pass conducted by the RL network using the network layer outputs Y, so that in each forward pass: (2a) $N = N+1$, (2b) $\delta = Y-\mu$, (2c) $\mu = \mu+\frac{\delta}{N}$, (2d) $\delta_2 = Y-\mu$, and finally (2e) $M2 = M2+\delta\times\delta_2$ for each neuron output. Finally in step (3) we finalize the neuron output statistics to generate [μ,σ,σ_s] where σ denotes the standard deviation and σ_s denotes the sample standard deviation. The mean μ is directly taken from the aggregate vector generated above, and the standard deviations are calculated as:

$$\sigma = \sqrt{\frac{M2}{N}} \tag{1}$$

$$\sigma_s = \sqrt{\frac{M2}{N-1}} \tag{2}$$

In the case of a convolutional layer, this yields two tensors of identical dimension to the convolutional outputs, one tensor for averages and the other for the standard deviation of neuron outputs. For a dense layer these statistics form two vectors of identical dimension to the dense layer outputs, one vector of averages and one vector of standard deviations.

The μ and σ values from the training statistics are then used to set statistical thresholds for error detection and suppression. The statistical thresholds are tuned using confidence parameters to flag and enable suppression of extreme values indicative of errors (discussed further in Section III-C). In this work, the thresholds are applied during RL DNN inference. For a ReLU function this is simply a single upper threshold, with the lower threshold set at zero due to the one-sided nature of the ReLU output (see Section III-A). In the case of hyperbolic tangent (tanh) activation, the thresholding and suppression (and by extension statistics collection) is performed at the linear layer preceding activation, as the tanh's bounded nature allows thresholds at the activation to be used for suppressing activation errors. Suppression of linear layer errors prevents these errors from causing the tanh activation to saturate at 1 or -1, which can affect RL performance.

In the event that a neuron output breaches either statistical bound (lower or upper), that output is deemed erroneous and set to zero (suppressed). The bounds are set such that values at the extreme tails of the distributions are suppressed.

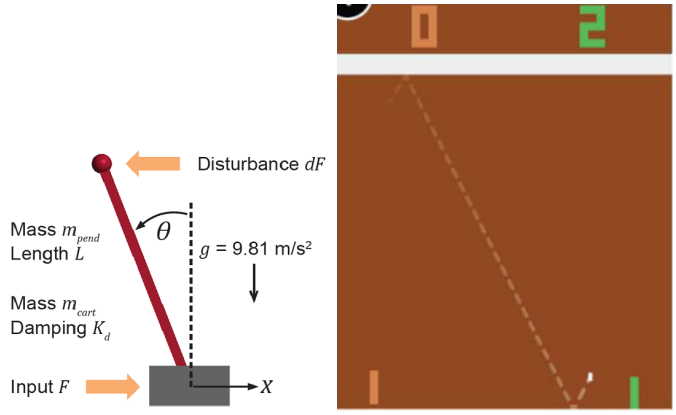
C. Neuron Output Statistical Thresholding

The statistical thresholds for flagging erroneous neuron outputs are set using different concentration inequalities depending on the location of error suppression. For *two sided* outputs such as those from a linear layer, we use Chebyshev's concentration inequality [24] to flag and suppress errors:

$$\Pr(|Y-\mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{3}$$

where Y indicates the neuron outputs being thresholded (or the MAC outputs for a tanh activation), μ is the mean from Section III-B, σ is the corresponding standard deviation (see Equation 1) and k is a user-defined tuning parameter that sets the 'tail' thresholded by the inequality.

For *single sided* neuron outputs such as those from a ReLU activation, Chebyshev's inequality is insufficiently tight. We instead use Cantelli's inequality [25] for one-sided bounds



(a) Inverted pendulum state-space model (image source: Mathworks)

(b) Atari Pong game (image source [21])

Fig. 3: Error injection test cases: (a) The inverted pendulum state-space system, and (b) The Atari Pong game. RL methods used for each are discussed in Section IV-A. Reward details are discussed in Section IV-B and IV-C respectively for each.

(since ReLU outputs are greater than or equal to zero always):

$$\Pr(Y-\mu \geq k) \leq \frac{\sigma^2}{\sigma^2+k^2} \tag{4}$$

where k is again a tunable parameter and Y, μ and σ are identical to Equation 3. The k values in Equations 3 and 4 are used to set the 'tail' probability of the left hand side, allowing thresholding and suppression of extreme (erroneous) outputs.

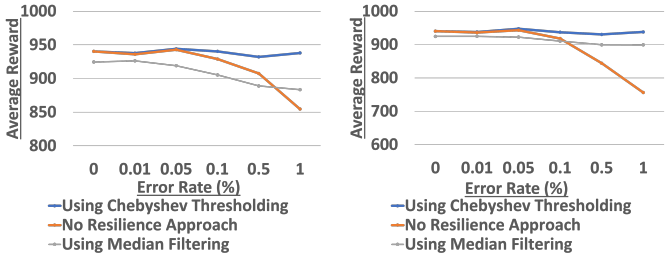
IV. Experimental Results

A. Experimental Details

1) Test Platform Details

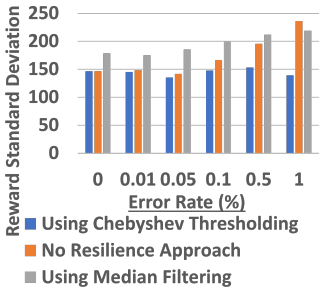
Our proposed RL DNN error resilience approach has been validated on two different reinforcement learning benchmarks. The first test platform is a policy gradient network using the REINFORCE [26] algorithm for learning the optimal policy distribution directly to swing up and balance an inverted pendulum. The second test platform is the Atari Pong benchmark for Deep Q-Learning [21]. The REINFORCE test platform is trained using the Gymnasium library code [27] with statistics collection for threshold generation and compared against a baseline using median feature selection [5]. Difficulties in adapting training of the Atari Pong neural network (memory and convergence issues) prevented the use of a median feature selection baseline. The Pong network is trained using the CleanRL library [28] with statistics collection for threshold generation. All networks were implemented in PyTorch [29].

REINFORCE for inverted pendulum: REINFORCE aims to maximize the Monte Carlo reward [26] by swinging up and balancing an inverted pendulum as long as possible (See Fig. 3a). This RL approach parameterizes the policy using a neural network (run on a CPU), estimating the mean and standard deviation of the policy distribution (assumed Gaussian). The action is then sampled from this distribution. The network in this case is a linear DNN with two hidden layers (the first using 16 neurons and the second using 32 neurons), each using a hyperbolic tangent activation, followed by two output

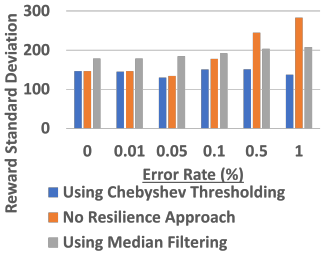


(a) 4-bit error injection results, average accuracy

(b) 8-bit error injection results, average accuracy



(c) 4-bit error injection results, standard deviation



(d) 8-bit error injection results, standard deviation

Fig. 4: Error injection results for the Inverted Pendulum swingup REINFORCE benchmark.

linear layers. One output layer yields the mean of the policy distribution and the other yields the standard deviation.

Atari Pong Q-Learning: Deep Q-Networks (DQNs) are used to train RL agents to be as effective as possible at winning games of Atari Pong, minimizing misses by the agent and maximizing misses by the opposing player in [21] (See Fig. 3 for screenshot). This is done by using DNNs (running on a CPU) to learn an approximation of the optimal action-value function $Q* = \max_\pi E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi]$, maximizing the sum of rewards r_t subject to the discount factor γ and achieved by a policy π that is not explicitly estimated. The method implemented here (that of [21] as in [28]) uses experience replay, randomly sampling from prior observations (state-action pairs in prior timesteps) to remove correlations in data. It iteratively updates the Q-values, adjusting them towards periodically updated target values to further reduce correlations with the target. Unlike the REINFORCE DNN, the DQN uses ReLU activation.

2) Experimental Parameters and Metrics

The REINFORCE test platform was trained for 8000 episodes with statistics collection for threshold generation and 10K episodes using median feature selection. The REINFORCE network used statistical thresholds after each linear layer (before tanh activation). Here, k was set to 31.62 (thresholding the 0.1% tails of the distribution) in Chebyshev's bound (Equation 3). This network used 32-bit floating point.

The Atari Pong DQN was trained for 10 million episodes normally with statistics collection for threshold generation. The DQN used statistical thresholds for error resilience after each ReLU activation, with the upper bound provided by Cantelli's inequality (Equation 4) and lower bound set to zero (ReLU lower bound). Here, the inequality thresholded at the 0.025% upper tail of the distribution ($\Pr(Y - \mu \geq k) \leq 0.00025$). This network used 16-bit floating point.

Error injection into RL DNN inference post-training was done using the PyTorchFI [30] library. Error injection was governed by an *error rate* parameter, denoting the probability of each neuron output in each layer (except the final layer) being erroneous. Erroneous neuron outputs were subject to varying *error severity*, varying the number of bits flipped.

Each error injection experiment recorded *average* and *standard deviation* of reward over a number of test (inference) episodes, varying error rate and error severity. Over the REINFORCE test platform, overhead was recorded using CPU program counter measurements via the PyPAPI library [31].

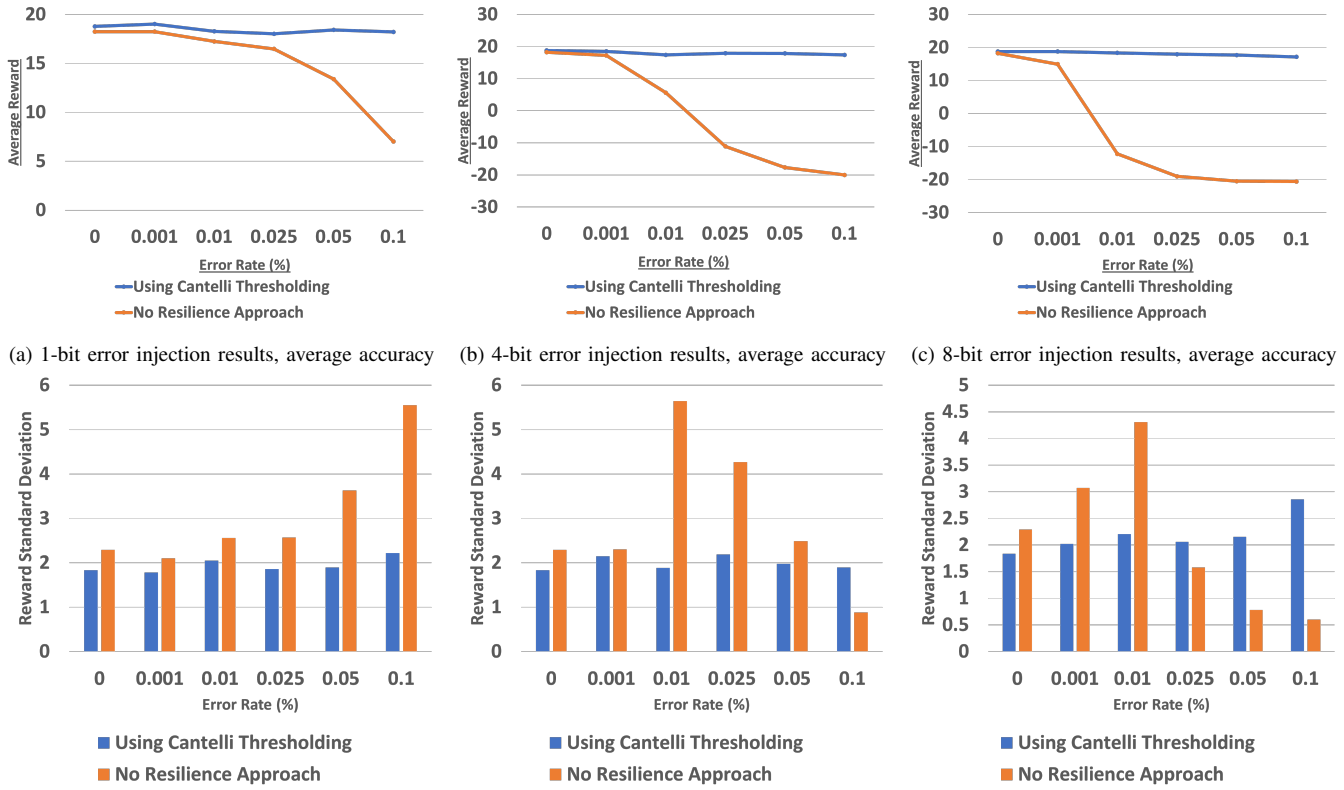
B. Error Injection: REINFORCE for Inverted Pendulum

Figure 4 shows results for error injection into the inverted pendulum using REINFORCE. The *reward* here consists of +1 for each timestep the inverted pendulum remains upright, up to a maximum episode length of 1000. The average reward over 600 test episodes is shown in Fig. 4, for a range of error rates and 4- and 8-bit errors. It was found that 1-bit errors caused little to no drop in performance until high error rates even in the absence of any error resilience method. Fig. 4a and Fig. 4c show the mean and standard deviation of reward under 4-bit error injection for the proposed approach, median filtering (baseline) and the case of no error resilience enabled. Figs. 4b and 4d show the same quantities for 8-bit error injection.

We see that the average performance of the RL DNN degrades as error rate and error severity increase in the absence of error resilience methods, shown by a gradual decline in average reward both across error rates (in Figures 4a and 4b) and across error severity (a greater decrease in reward is seen in Fig. 4b than Fig. 4a). Median filtering shows a similar trend with more graceful degradation, ending in a slight average loss of performance at the cost of nominal reward. When using the proposed resilience method we see no material change in average reward across the error rates and severity.

The standard deviation in reward for the RL DNN shows a similar trend - showing a rise in reward standard deviation without any resilience method (indicating *less consistent performance*) for rises in error rate and error severity. The use of median filtering is seen to show a similar rise in reward standard deivation across the 4-bit error injection (Fig. 4c) and 8-bit errors (Fig. 4d). The use of the Chebyshev threshold restores near-nominal reward standard deviation (consistent performance) across the error rates on 4-bit and 8-bit errors.

Table I shows the overhead of our approach on the RL PPO network used for the inverted pendulum, compared to the overhead of the median filtering approach. All modules were implemented in PyTorch using off-the-shelf functions, and run on an Intel Xeon W-2123 CPU. Floating point operations and vector operations incurred no overhead over the RL DNN for both the thresholding approach and the median filtering approach. We can see that the thresholding-based approach incurs highest overhead for cache reads (due to threshold



(a) 1-bit error injection results, average accuracy (b) 4-bit error injection results, average accuracy (c) 8-bit error injection results, average accuracy

(d) 1-bit error injection results, standard deviation (e) 4-bit error injection results, standard deviation (f) 8-bit error injection results, standard deviation

Fig. 5: Error injection results for the Atari Pong DQN benchmark.

Measured Indicator	Statistical Thresholding Overhead (%)	Median Filtering Overhead (%)
Cache Reads (L2 and L3)	38.271	129.03
Cache Writes (L2 and L3)	26.21	97.43
Conditional Branch Instructions	19.152	64.51
Unconditional Branch Instructions	36.75	119.1

TABLE I: Average overhead (CPU program counter data) for Chebyshev-based thresholding and the median filtering baseline, as a percentage of the RL DNN overhead.

comparisons for neuron outputs) and unconditional branch instructions (additional calls for the thresholding module), and is still sub-40% even when using off-the-shelf PyTorch code.

C. Error Injection: Atari Pong Q-Learning

Fig. 5 shows results on the Atari DQN benchmark. Due to the larger size of this network, its much longer training time (10 million episodes) and its use of convolutional layers (as opposed to the linear (dense) layers of the test case in Section IV-B), this network was used to test the scalability of our approach. The Atari Pong agent earns +1 *reward* for the opponent missing the ball, -1 for missing the ball and zero for neither event happening in that timestep. One episode consists of one game of Pong, where a side wins if it reaches 21 points. Reward thus ranges from +21 to -21 per episode.

The plots in Fig. 5a, 5b and 5c show average reward over 50 inference episodes for a range of error rates and error severity (1, 4 and 8-bit errors). It can be seen that more severe errors have a greater effect on average reward for the case of no error resilience approach (comparing Fig. 5b and 5c to Fig. 5a). Similarly, higher error rates cause greater losses in performance (average reward) for the case of no error resilience approach. The use of Cantelli-based thresholding (Equation 4) restores near-nominal performance in all cases.

The plots in Fig. 5d, 5e and 5f show the standard deviation in reward for the same 50 test episodes. For less severe errors (1-bit errors) the DQN without any error resilience approach sees a steady rise in reward standard deviation (less consistent performance). For 4- and 8-bit errors, the DQN sees a similar rise and then fall in standard deviation as the average reward drops, indicating consistently poor performance. The use of Cantelli-based thresholding again restores near-nominal reward standard deviation (consistent performance).

V. Conclusion

In this work we present an on-line, theoretically grounded error resilience approach for RL DNNs, validating our approach on two different RL algorithms as well a variety of error rates and error severity conditions. In all examined tests cases in Section IV, RL agent performance degrades under error injection, while on-line statistical error suppression allowed more consistently optimal behavior than the baselines.

Acknowledgment

This research was supported by the U.S. National Science Foundation under Grant No. 2128149.

REFERENCES

[1] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, and Q. Miao, "Deep reinforcement learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022.

[2] E. Ozen and A. Orailoglu, "Architecting decentralization and customizability in dnn accelerators for hardware defect adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3934–3945, 2022.

[3] Y. Ibrahim, H. Wang, J. Liu, J. Wei, L. Chen, P. Rech, K. Adam, and G. Guo, "Soft errors in dnn accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113969, 2020.

[4] E. Ozen and A. Orailoglu, "Just say zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression," in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 2020, pp. 75:1–75:9. [Online]. Available: https://doi.org/10.1145/3400302.3415680

[5] ——, "Boosting bit-error resilience of dnn accelerators through median feature selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 1–1, 11 2020.

[6] C. Amarnath, M. Mejri, K. Ma, and A. Chatterjee, "Soft error resilient deep learning systems using neuron gradient statistics," in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2022, pp. 1–7.

[7] E. Ozen and A. Orailoglu, "Snr: S queezing n umerical r ange defuses bit error vulnerability surface in deep neural networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021.

[8] Y. Long, X. She, and S. Mukhopadhyay, "Design of reliable DNN accelerator with un-reliable reram," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 1769–1774. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715178

[9] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 1–13.

[10] C. Schorn, A. Guntoro, and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 979–984, 2018.

[11] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[12] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2546–2558, 2022.

[13] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, 2013.

[14] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[15] S. Zhang and N. R. Shanbhag, "Embedded algorithmic noise-tolerance for signal processing and machine learning systems via data path decomposition," *IEEE Transactions on Signal Processing*, vol. 64, no. 13, pp. 3338–3350, 2016.

[16] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Hardnn: Feature map vulnerability evaluation in cnns," 2020.

[17] C. H. Yang, I. D. Hung, Y. Ouyang, and P. Chen, "Causal inference q-network: Toward resilient reinforcement learning," *CoRR*, vol. abs/2102.09677, 2021. [Online]. Available: https://arxiv.org/abs/2102.09677

[18] K. L. Tan, Y. Esfandiari, X. Y. Lee, Aakanksha, and S. Sarkar, "Robustifying reinforcement learning agents via action space adversarial training," in *2020 American Control Conference (ACC)*, 2020, pp. 3959–3964.

[19] S. Banerjee and A. Chatterjee, "Alera: Accelerated reinforcement learning driven adaptation to electro-mechanical degradation in nonlinear control systems using encoded state space error signatures," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 4, pp. 1–25, 2019.

[20] A. Skolik, S. Mangini, T. Bäck, C. Macchiavello, and V. Dunjko, "Robustness of quantum reinforcement learning under hardware errors," *EPJ Quantum Technology*, vol. 10, no. 1, p. 8, Feb 2023. [Online]. Available: https://doi.org/10.1140/epjqt/s40507-023-00166-1

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[22] M. A. Hanif and M. Shafique, "Dependable deep learning: Towards cost-efficient resilience of deep neural network accelerators against soft errors and permanent faults," in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2020, pp. 1–4.

[23] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022

[24] R. Vershynin, "High-dimensional probability," 2019. [Online]. Available: https://www.math.uci.edu/~rvershyn/papers/HDP-book/HDP-book.pdf

[25] B. K. Ghosh, "Probability inequalities related to markov's theorem," *The American Statistician*, vol. 56, no. 3, pp. 186–190, 2002. [Online]. Available: http://www.jstor.org/stable/3087296

[26] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[27] F. Foundation, "Gymnasium," https://github.com/Farama-Foundation/Gymnasium, 2022.

[28] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-1342.html

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[30] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 25–31.

[31] D. Terpstra, H. Jagode, H. You, and J. Dongarra, *Collecting Performance Data with PAPI-C*. Springer Berlin, 2009.