Adding Noise to the Momentum Term in ANN to Reduce Overfitting

by

Santiago Villarreal Villarraga

A Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

of The University of Manitoba

in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, a thesis entitled:

ADDING NOISE TO THE MOMENTUM TERM IN ANN TO REDUCE OVERFITTING

submitted by SANTIAGO VILLARREAL VILLARRAGA in partial fulfillment of the requirements of the degree of Master of Science

Dr. Ken Ferens. Price Faculty of Engineering
Supervisor

Abstract

Overfitting remains a common problem in training ANNs, where models fit the training data well but lose accuracy on new inputs. Many regularization methods have been proposed, yet most focus on changing the architecture or adding constraints on weights and gradients. This work investigates a different method: adding noise to the momentum term of an optimizer. The idea is that choosing a noise can help the training process escape sharp minima in the loss surface and settle in flatter regions that generalize better.

The optimizers SGD, Adam, Nadam, and RMSprop were modified to accept fractional Brownian motion (fBm) noise, both with fixed and adaptive learning based on the gradient. Furthermore, an intermittent 'burst' noise strategy is introduced that injects fractional Brownian motion in concentrated intervals, allowing the optimizer to alternate between periods of high-energy exploration and stable refinement. Noise is injected into the momentum term or directly into the weight updates. Experiments are run on five datasets, MNIST, CIFAR-10, NSL-KDD, BoT-IoT, and CIC-IoT, with three Hurst exponents (0.2, 0.5, 0.7) to test different temporal correlation patterns. Performance is measured across accuracy, F1-score, ROC-AUC, cross-entropy, and the generalization gap, averaged over multiple runs.

Results show that noise on the momentum term works better than

adding it directly to weights, and it doesn't slow down training much. Anti-persistent noise ($H = 0.2$) achieves the largest gap reduction, though Brownian noise ($H = 0.5$) offers a better balance between stability and capacity. The best overall results are obtained with Adam or Nadam combined with fixed-variance momentum noise, improving test accuracy on both vision and traffic-analysis tasks without extra training cost.

Preface

This practicum report is submitted in partial fulfillment of the requirements for the Master of Science in Electrical and Computer Engineering at the University of Manitoba. The work presented here investigates the use of fractional Brownian motion noise in the momentum term of common optimization algorithms to improve generalization and reduce overfitting in artificial neural networks.

The idea for this project originated from research carried out by the supervisor, Dr. Ken Ferens, in the field of cybersecurity. Using those concepts, A set of experiments was designed and implemented to test whether structured noise could be applied to training optimizers in both vision and network traffic classification tasks. While the original focus was on network security, the scope was extended to include image datasets so that the methods could be evaluated across different data domains.

All experiments and code development were conducted by the author. The study uses five publicly available datasets: MNIST and CIFAR-10 for vision, and NSL-KDD, Bot-IoT, and CIC-IoT for network traffic analysis.

The report is organized into four main chapters, followed by appendices. Chapter 1 outlines the motivation, background, and objectives of the work. Chapter 2 outlines the methodology, encompassing problem definition, liter-

ature review, experimental design, and implementation. Chapter 3 presents the results and evaluation, with separate sections for each Hurst parameter tested and a broader analysis of runtime, accuracy, and generalization. Chapter 4 summarizes the main findings and conclusions. The appendices contain complete metric tables, figures for experiments, and the source code used in the project.

Table of Contents

List of Tables

List of Figures

Acronym List

ANN	Artificial Neural Network
SGD	Stochastic Gradient Descent
HB	Heavy-Ball momentum
PNM	Positive-Negative Momentum
SGHMC	Stochastic Gradient Hamiltonian Monte Carlo
fBm	Fractional Brownian Motion
H	Hurst exponent
ROC-AUC	Receiver Operating Characteristic Area Under Curve
PR-AUC	PrecisionRecall Area Under Curve
CE	Cross-Entrop
XLA	Accelerated Linear Algebra compiler (TensorFlow)
PACBayes	Probably Approximately Correct Bayes
MNIST	Modified National Institute of Standards and Technology (dataset)
CIFAR-10	Canadian Institute For Advanced Research 10-class dataset (dataset)

NSL-KDD	Network Security Laboratory Knowledge Discovery and Data Mining (dataset)
CIC-IoT	Canadian Institute for Cybersecurity Internet of Things (dataset)

Acknowledgments

I would like to thank my supervisor, Dr. Ken Ferens, for his guidance, patience, and encouragement throughout this project. His work in cybersecurity inspired the idea behind this study, and his advice was essential in shaping its direction from start to finish.

I am also grateful to the Faculty of Engineering at the University of Manitoba for providing the resources and facilities needed to complete this research, and to the International Graduate Student Entrance Scholarship for supporting me during my studies.

Finally, I want to express my deepest appreciation to my partner, Luciana, and my family. Their constant support, understanding, and patience gave me the motivation to keep going through the long hours of coding, testing, and writing.

Chapter 1

Introduction

Artificial Neural Networks (ANNs) are computational models made up of layers of interconnected units, often referred to as neurons, that are designed to learn patterns from the data. These models have been highly successful in a wide range of tasks, like image analysis, NLP, and sequence predictions, largely due to their ability to approximate nonlinear functions with high accuracy. However, training ANNs involves navigating a complex, high-dimensional loss surface, a process that depends heavily on the optimization algorithm used.

One of the most used optimization methods for training ANNs is Stochastic Gradient Descent (SGD). This algorithm updates the network's parameters using the gradient of the loss function, calculated on small batches of data. However, standard SGD often struggles to make consistent progress when the loss is steep or curved, causing the optimization to become unstable or slow. To address this, an enhancement known as momentum was introduced. Originally proposed by Polyak in 1964 [Pol64], momentum helps the optimizer accumulate gradient information over time, smoothing out updates and enabling faster convergence. Momentum acts like a form of memory, allowing the algorithm to continue moving in favorable directions

while preventing any erratic shifts. This technique has since become a fundamental component in the training of ANNs.

Despite these advantages, ANNs are flexible models that can easily overfit the training data, especially when the datasets are small or noisy. Overfitting happens when the model learns the patterns and random fluctuations in the data, resulting in low generalization to unseen examples. Although several regularization strategies, such as weight decay, dropout, and data augmentation, have been proposed to mitigate this issue, recent research has suggested a different method: introducing noise directly into the optimization process.

An emerging idea is to add random noise to the momentum term itself [XYZS21a]. This can help the optimizer escape narrow, sharp minima, which often correspond to overfitted solutions, and instead favor broader, flatter regions of the loss that generalize better. For instance, Neelakantan [NVL+15] demonstrated that adding decaying Gaussian noise to gradients during training led to better performance in deep networks. Similarly, Chaudhari and Soatto [CCS+19] introduced Entropy-SGD, which uses a short burst of noisy updates within each training step to push the model toward flatter minima. More recently, Xie [XYZS21b] proposed a Positive-Negative Momentum, which keeps two independent momentum terms, one updated in the usual (positive) direction, the other in the opposite (negative) direction, and uses their difference as the weight update.

These developments point to a hypothesis: by adding noise to the momentum component of gradient-based optimizers, it is possible to improve generalization without compromising the speed of convergence. This project

explores that hypothesis by implementing and evaluating noisy versions of several popular optimizers. These include traditional SGD with momentum, RMSprop with momentum, and Adam. In each case, noise is added to the velocity or the weights, depending on the optimizer's structure.

This work focuses on how different types of noise, fixed noise and adaptive noise, influence training performance and generalization in neural networks. These approaches are compared on tasks ranging from vision to network security to see whether momentum noise can be a practical regularization tool that limits overfitting without slowing training.

1.1 Objectives

The primary goal of this research is to determine if the integration of structured noise into the optimization process can effectively guide artificial neural networks toward flatter regions of the loss surface that generalize better to unseen data. To achieve this, the following objectives are defined:

- Investigate Structured Noise as a Regularization Tool: Evaluate the use of fractional Brownian motion (fBm) and curvature-adaptive noise within standard gradient-based optimizers, specifically SGD, Adam, Nadam, and RMSProp, as a mechanism for reducing overfitting

- Determine Optimal Noise Placement: Identify the most effective point of injection by comparing the performance impact of noise applied directly to weight updates against noise integrated into the internal momentum (velocity) terms of the optimizers

- Quantify the Impact of Temporal Correlations: Assess how different Hurst exponents (H=0.2,0.5,0.7) influence training dynamics, specifically looking for a balance between aggressive gap reduction (anti-persistence) and model stability (Brownian or persistent motion).

- Evaluate Cross-Domain Robustness: Validate the efficacy of momentum-based noise across distinct data environments, including image classification and high-dimensional network traffic analysis for cybersecurity.

1.2 Contributions

This thesis makes the following contributions to the study of optimization and regularization in artificial neural networks:

- It proposes and evaluates the injection of structured stochastic noise into the momentum term of gradient based optimizers as a practical regularization mechanism to reduce overfitting. Unlike prior approaches that perturb gradients or weights directly, this work focuses on modifying the internal velocity dynamics of the optimizer.

- It introduces the use of fractional Brownian motion as a source of noise for momentum based optimization. By controlling the temporal correlation of the noise through the Hurst exponent, the method enables a systematic study of how long range dependence and anti persistence affect convergence behavior and generalization.

- It presents a comparative analysis of noise placement strategies by evaluating noise injection in the momentum term versus direct per-

turbation of the model weights. The results consistently show that momentum based noise achieves better generalization with less disruption to training stability.

– It develops and evaluates fixed variance, curvature adaptive, and intermittent burst noise strategies within the same experimental framework. This allows a controlled comparison of continuous versus intermittent stochastic exploration during training.

– It extends the evaluation of noisy momentum methods beyond standard vision benchmarks by applying them to high dimensional network traffic classification tasks. This cross domain analysis demonstrates that the proposed approach is not limited to image data and remains effective in cybersecurity related datasets.

– It provides an empirical analysis of generalization gap, convergence time, and performance stability across multiple optimizers, including SGD, Adam, Nadam, and RMSProp. The results show that anti persistent noise yields the largest gap reduction, while Brownian noise offers a balanced tradeoff between stability and generalization.

Chapter 2

Methodology

2.1 Problem Definition

Despite their strengths, ANNs often suffer from overfitting; they minimize the training loss, yet lose accuracy on data that is shown few times. Empirical and theoretical studies tie this gap to the geometry of the solution to which training converges. Small batch stochastic gradient descent (SGD) naturally explores the loss surface and tends to settle in broad, low-curvature 'flat' minima, whereas other training settings can lead to sharp, narrow basins that generalize poorly.

One solution is to inject extra randomness so the optimizer keeps changing until it discovers flatter regions, hopefully to be a new, better minimum. Work along these lines ranges from Entropy-SGD, which adds Langevin perturbations inside an internal optimization, to Positive-Negative Momentum (PNM), which modulates two opposing momentum terms to reproduce anisotropic gradient noise. Yet most of this research perturbs either the gradients themselves or a single optimizer (usually plain SGD), leaving fundamental questions unanswered.

First, where should the noise be applied? Injecting it into the momentum (velocity) term could run updates differently from adding it directly to

the weights, but comparative evidence across mainstream optimizers such as Adam, Nadam, and RMSProp is low. Second, what kind of noise best mimics the beneficial randomness of SGD? Simple white Gaussian noise ignores the long-range correlations seen in real training dynamics, whereas fractional Brownian motion (fBm) explicitly models those correlations. Recent proposals for curvature-adaptive noise, whose variance grows in steep regions and shrinks on plateaux, are promising but still preliminary.

This project studies whether carefully shaped noise, either fixed or adaptive, inserted either into the momentum term or directly into the weight update of several popular optimizers (SGD, Adam, Nadam, RMSProp), reduces overfitting in classification tasks by guiding training toward better generalizing minima. Finding this answer would show not just where noise should be placed, but also how strong it should be, giving a clear path to making networks more robust without altering their design.

2.2 Literature Review

2.2.1 Optimizers and Momentum

The concept of momentum is a term used with the analogy of physics, which, by Newton's principles of motion, uses the negative gradient as a force to move a particle through space. It was first proposed in computer science in 1964 by Polyak as the heavy ball method [Pol64]. This idea originated with Polyak, but Rumelhart, Hinton, and Williams transplanted it into back-propagation in 1986 [RHW86], popularizing what is currently termed SGD with momentum.

Momentum is introduced to address two weaknesses of the normal SGD: slow progress in flat directions and zig-zagging in narrow "gaps". It increases each parameter with a velocity vector V_t that accumulates an exponential moving average of previous gradients.

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla L(\theta_{t-1}) \tag{2.1}$$

$$\theta_t = \theta_{t-1} - \eta v_t \tag{2.2}$$

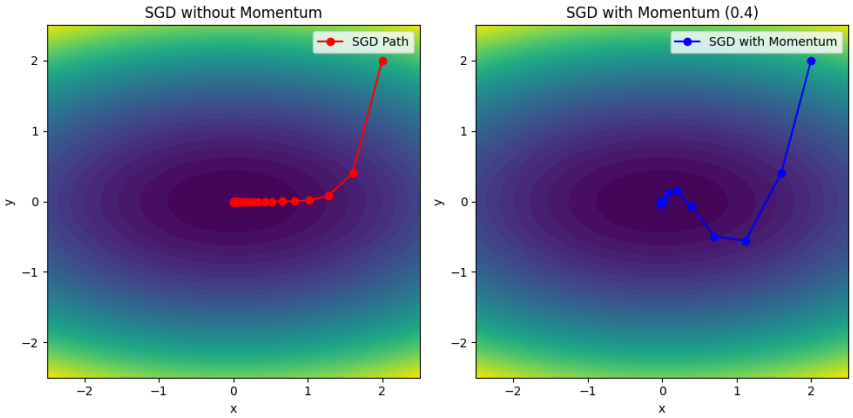


Figure 2.1: SGD vs Momentum

Momentum became a main component of the optimizers, like Adam, that integrates momentum directly into its parameter updates by maintaining a first-moment estimate of the gradients and a second-moment that rescales the step. RMSProp-M, Nadam, AdaMax, AMSGrad, RAdam, QHAdam, and Yogi also use momentum, but they differ in how they normalize or rectify this second moment, and each retains the same β momentum.

2.2.2 Overfitting and Generalization

One main challenge in machine learning is bridging the gap between observations made during training and what happens in the real world. Two distributions are present: P_R (the true, unknown distribution of real data) and P_F (what our model learns from the training examples it has).

Generalization is the ability of an ML model to predict or perform under new, unseen data. A model generalizes well when these two distributions are close, whatever patterns the model learned, making P_F as close as possible to P_R. Goodfellow introduces the idea of seeking a small generalization error, which refers to the difference between the error measured on the training data and the error obtained on an infinite test set sampled from the true distribution. If the model has just the right capacity, the training and test errors should track each other closely. [GBC16]

Overfitting is the opposite situation. As training continues, the error on the training set can keep going toward zero while the error on new data stops improving and even starts to grow. In that moment, the model has learned the anomalies like random noise that are only in the training set, so its behavior outside that limited data becomes inconsistent. In the book, this divergence is traced to a mismatch between the models capacity and the amount of reliable details present in the dataset.

While overfitting describes a model that follows the unimportant changes of the training set, underfitting is its opposite; the network is too limited to capture the true structure of the data at all. When the model lacks expressive ability, the probability that it learns sits far from the peaks and

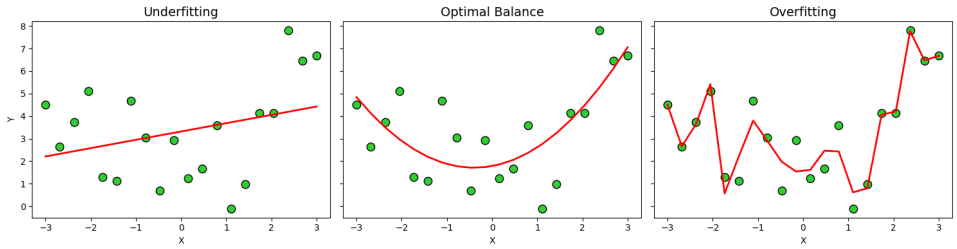


Figure 2.2: Generalization on models

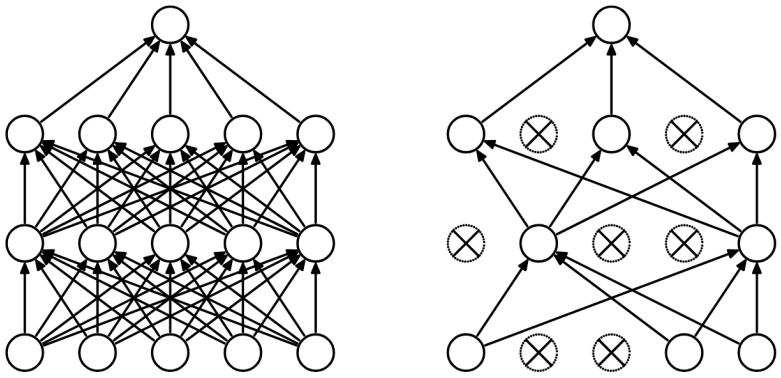
valleys of the real data distribution P_R. In practice, this shows up when both losses (training and validation) go to high values; there is no wide gap as in overfitting, but neither curve falls to an acceptable level. The situation is also easy to diagnose on a bias-variance chart: high bias dominates, signaling that the learner's assumptions (for example, linear decision boundaries for a curved problem) are too strict to represent the patterns. Conceptually, an underfit model behaves like drawing a straight line through data points that trace a complex curve 2.3. Its simplicity prevents it from doing well on the examples it has seen and on those it has not.

2.2.3 Regularization Techniques

Overfitted models are good at memorizing training data while being useless on anything new. To keep the training distribution with the real-world distribution, a set of regularization techniques can be used whose purpose is to make overfitting harder and proper generalization easier. The Deep Learning book [GBC16] groups several of these under the broad banner of Regularization for Deep Learning, and the methods below are among the most common.

First, weight decay (L-2 regularization), which tackles the problem at the parameter level. A quadratic penalty λ is added to the loss, so every gradient-descent step also presses each weight toward zero. Shrinking the coefficients in this way makes sharp, highly specialized decision boundaries expensive to maintain, encouraging the network to settle on smoother functions that fluctuate less between neighboring inputs. The penalty is equivalent to assuming a zero-mean Gaussian prior over the weights; small values are inherently more likely, so the optimizer prefers them and generalization improves.

Following Dropout, which attacks overfitting from inside the network. During training, each hidden unit is randomly dropped with probability P, forcing the forward pass to route information through a changing subset of the whole network. With dropout, each hidden unit can vanish during training, forcing the network to learn patterns that survive in many smaller subnetworks. At test time, all units are active again, and the combined effect often reduces the gap between training and validation, a property that will be compared against the noise-based regularization in this work.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.3: Dropout by Srivastava, Nitish (2014) [SHK+14]

Another technique is Data augmentation, which widens the effective training distribution rather than altering the model. By applying changes to the data, like rotations, flips, crops, colour shifts, or slight additive noise in images, the model sees a wider variety of inputs that should still map to the same targets. Feeding the network these altered views teaches it to ignore superficial changes and focus on deeper structure, reducing variance without gathering new data.

Finally, noise injection. The book talks about two different ways to add noise: adding it to the weights and the output. First, injecting noise at the output targets. One way to regularize a network is to add little noise to the labels rather than the inputs or the parameters. Suppose a classifier is trained with one target y. By mixing each target with a small amount of uniform noise, y is replaced by $y = (1 - \epsilon)y + \epsilon/k$ for k classes. This softens the loss, discouraging the model from assigning maximum probability to any single class and making it less sensitive to occasional mislabelings in the

dataset. Goodfellow [GBC16] shows that the technique has the same nature as label smoothing used in modern computer vision and language models; both act like a KL-divergence penalty that keeps the predictive distribution from collapsing onto a delta spike. Next, noise is injected into the weights. Here, each forward pass perturbs the weight tensor by $W = W + \sigma N(0, 1)$ averaged over many mini-batches. This is equivalent to adding an explicit regulariser that penalises sensitive, high-curvature solutions. Because the perturbation is applied inside the computation graph, the optimizer must find weight configurations whose outputs vary smoothly under these random changes, pushing the network toward wider, flatter minima that generalize better. Recent research has pushed this idea further. TheWhiteout[LL20] model injects adaptive Gaussian noise whose variance is itself trainable, showing consistent reductions in overfitting on CIFAR10 and SVHN while unifying weight noise with classical L_y penalties.

2.2.4 Noise and Noise Injection

As it was noted earlier, injecting noise during training is a whole family of regularization tricks that force the optimizer to explore a rougher cost landscape, and thus learn solutions that generalize better. The first noise that comes to mind to add to an ANN is white noise; the foundation theory for those solutions started with Christopher Bishop [Bis95], who showed in 1995 that adding a Gaussian noise with fixed variance to the input or weights of a network is equivalent to the original loss plus a Tikhonov term (ridged regression). By adding the noise to the input:

$$\mathbb{E}_{\xi}\big[L\big(f(x+\xi;w)\big)\big] \;=\; L\big(f(x;w)\big) \;+\; \frac{\sigma^2}{2}\,\big\|\nabla_x f(x;w)\big\|^2 \;+\; O\big(\sigma^4\big) \quad (2.3)$$

Bishop showed that adding white noise discourages sudden changes in predictions by penalizing large input Jacobians. This penalty is efficient, stable, and works even when noise is fairly strong, making white-noise injection a dependable baseline for regularization. Other investigations confirm that this firstorder approximation remains accurate even when σ is as large as 10% of the input dynamic range, establishing white noise injection as a reliable baseline regularization method.

Bishop's work raises the question of where exactly to inject noise. Camuto [CWS+20] asked what happens when the same constant variance Gaussian perturbation is applied straight to the weights. By marginalizing over the injected noise, they derived an exact loss:

$$\tilde{L}(w) = L(w) + \frac{\sigma^2}{2}\,\mathbb{E}_{x\sim D}\Big[\mathrm{Tr}\big(J_f(x;w)^\top J_f(x;w)\big)\Big] \quad (2.4)$$

Where J_f is the Jacobian of the network outputs for the weights. The additional term is a layer Jacobian norm penalty, which means that although every parameter receives the same noise, deeper layers, whose Jacobians are larger, are automatically regularized more strongly. Experiments on CIFAR-10/100 showed that for fixed σ, weight noise improved the test accuracy by roughly 1% over the standard decay at identical compute and produced noticeably flatter Hessian spectra, confirming that constant perturbations can improve generalization by implicitly enforcing smoothness.

Previously, Richard Zur [Zur09] performed a controlled comparison between noise injection, weight decay, and early stopping on both simulated XOR data and a breast ultrasound classification task. Their small sample simulations (50 to 200 training cases) revealed that Gaussian weight noise recovered 87% of the ROC area lost to overfitting when only 50 examples were available, while weight decay and early stoppage reclaimed 53% and 48%, respectively. In 200 cases, the three methods converged, but noise injection still outperformed the baselines. On the real ultrasound sets the pattern persisted, with noise beating early stopping. These findings corroborate Camuto's theoretical claim that a fixed variance perturbation can act as a robust, efficient regularizer whose effectiveness does not depend on delicate hyperparameter tuning.

Another method is to adapt the noise during training, letting its variance or even full covariance change with the models behavior instead of keeping it fixed. The model presented by Yinan Li and Fang Liu, called Whiteout [LL20] it makes the variance of the Gaussian noise injected a smooth operation of the current weight vector w. The expected log-likelihood of a generalized linear model decomposes into the usual loss plus the penalty:

$$R(\mathbf{w}) \;\approx\; \frac{\sigma^2}{2}\,\mathbf{1}^\top \Lambda(\mathbf{w})\,\mathbf{1}\,\|\mathbf{w}\|_{1-\gamma}^{2-\gamma} \tag{2.5}$$

where $\Lambda(\mathbf{w}) = diag(A(xi^\top \omega))$ and $\gamma \in (2, 2)$. Because big weights see larger variance, training is encouraged toward smaller, flatter minima. In experiments on several UCI-style data sets plus CIFAR-10, Whiteout beats dropout and shakeout whenever the training set is small and never losses on

the large sets. Li concludes that adaptive variance lets a network keep just the weights it needs, giving tighter generalization with no extra tuning.

Now, with this in mind, think about adding the noise to the gradient. Neelakantan [NVL+15] proposes perturbing each minibatch gradient with an annealed Gaussian term so that the update becomes

$$\tilde{g}_t = g_t + \mathcal{N}\big(0, \sigma_t^2 I\big) \tag{2.6}$$

where the variance decays polynomially as

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \tag{2.7}$$

with a typical choice of $\gamma = 0.55$ and $\eta \in \{0.01, 0.3, 1.0\}$. By starting with relatively large noise and then letting it fade, this single modification helps the optimizer escape poor local minima early on and then refine the solution once the landscape has been explored. The results of the tests show that on a 20-layer ReLU network initialized poorly, standard SGD fails to learn, but the noisy gradient succeeds. Reduces error by 72% relative to a tuned baseline; and on neural GPU models for binary multiplication, it more than doubles the success rate across thousands of random restarts. Neelakantan concludes that the gradient noise is a lightweight optimizer compatible with Adam, AdaGrad, or vanilla SGD, that reliably enhances both convergence and generalization in very deep, non-convex architectures.

Using other solutions, the noise can also be added to the momentum itself. Xie [XYZS21b] creates controllable, extra noise without touching the

learning rate or batch size. Instead of maintaining a single velocity term, the model keeps two momentum terms, one for odd steps and one for even, and mixes them so that the effective update contains the controllable difference term $\beta_0(m_t - m_{t-1})$.

$$\theta_{t+1} = \theta_t - \sqrt{\frac{\eta}{(1+\beta_0)^2 + \beta_0^2}}\left[(1+\beta_0)\, m_t \;-\; \beta_0\, m_{t-1}\right] \tag{2.8}$$

Xie proves that this simple modification retains all convergence guarantees of classical momentum: the optimizer still follows a descent path with the same stability properties, so no extra tuning is required. Within a PACBayes framework, they show that the added difference term effectively widens the stationary distribution of the parameter dynamics, leading to a closer upper bound on the test loss. The test results on ResNet-18 and ResNet-34 for CIFAR-10, as well as on Transformers for WMT-14, consistently demonstrate a few percentage points of improvement in test accuracy over standard momentum, with little additional computation. Because the method does not alter learning rate or batch size, it can be drop-in inserted into any existing SGD or Adam implementation, offering an easy dial-up for stochastic gradient regularization whenever enhanced generalization is desired.

Using another method, Chen [CFG14] adds noise directly to the momentum update via a second-order Langevin dynamics, producing Stochastic Gradient Hamiltonian Monte Carlo (SGHMC). SGHMC augments the usual momentum update with both a friction term and carefully calibrated Gaussian noise so that the sampler preserves the correct posterior distribu-

tion even when using minibatch gradients. The updates take the form

$$v_{t+1} = (1-\alpha)\, v_t \;-\; \eta\, \nabla \tilde{U}(\theta_t) \;+\; \sqrt{2\,\alpha\,\eta\,T}\;\xi_t, \tag{2.9}$$

$$\theta_{t+1} = \theta_t \;+\; v_{t+1}, \tag{2.10}$$

where $\alpha \in (0,1)$ is the friction coefficient, η the step size, T a temperature parameter, and $\xi_t \sim \mathcal{N}(0, I)$.

Chen proves that adding $\sqrt{2\alpha\eta T}\,\xi_t$ exactly compensates for the variance introduced by stochastic gradients, so the stationary distribution of (θ, v) matches the true Hamiltonian posterior. SGHMC trains deep neural classifiers and models to the same training loss as full batch Hamiltonian Monte Carlo in a fraction of the time, and crucially achieves lower test error than vanilla SGD. SGHMC maintains correct stationary distribution and yields superior generalization while retaining SGD-level speed, making it a lightweight, optimizer-agnostic way to inject adaptive momentum noise for improved exploration and regularization.

Continuing the theme of noise, there is a need to find a noise that can be adjusted for all the needs of an ANN. Fractional Brownian motion (fBm) is a Gaussian process $B_H(t)$ with the covariance

$$\mathrm{Cov}\big(B_H(t), B_H(s)\big) = \tfrac{1}{2}\big(|t|^{2H} + |s|^{2H} - |t-s|^{2H}\big), \tag{2.11}$$

is controlled by a single Hurst exponent $H \in (0,1)$. Setting $H = \frac{1}{2}$ reproduces ordinary Brownian motion, $H > \frac{1}{2}$ causes persistent (positively correlated) increments, and $H < \frac{1}{2}$ produces anti-persistent behavior. Be-

cause the increments

$$\xi_t \;=\; B_H(t{+}1) - B_H(t), \qquad \mathrm{Cov}(\xi_t, \xi_s) \propto |t-s|^{2H-2} \tag{2.12}$$

Retain this dependence, injecting ξ_t into the weights or outputs supplies a tunable *memory* that standard white noise lacks.

Hairer [HL22] shows that when fBm manipulates a multi-scale system, the slow component converges to a Markov diffusion whose generator depends smoothly on H. Therefore, using $H > 0.5$ preserves long-range dependence in flat regions of the loss, whereas $H < 0.5$ accelerates forgetting in steep zones, offering an alternative to annealing schedules. Complementing this theory, Verdier [VLC+22] introduces a linear time variational-inference pipeline that learns (H, σ) directly from data, allowing an optimizer to adapt its own fBm noise just as adaptive-gradient methods tune step sizes. Finally, financial work such as Araneda [Ara23] embeds generalized fBm in option-pricing SDEs, capturing volatility clustering and other real-market memory effects more accurately than other solutions. It is noted that fractional Brownian motion (fBm) can serve both as a fixed long-memory perturbation and, by tuning or learning its Hurst exponent, as an adaptive one.

While the dominant approach to noise injection involves continuous perturbations that either remain constant or decay over time, recent research has highlighted the utility of intermittent or "burst" noise schedules. Continuous noise, while effective for exploration, can hinder the final convergence of the model by preventing the optimizer from settling into the precise bottom of a minimum, a phenomenon often managed by annealing the noise

variance to zero. However, annealing schedules run the risk of cooling the system too quickly, trapping the parameters in suboptimal basins before adequate exploration has occurred. To address this trade-off, several studies have proposed alternating phases of high-noise exploration and noise-free refinement, a concept deeply rooted in the principles of simulated annealing [KGV83] and cyclical optimization.

The theoretical foundation for intermittent noise lies in the separation of the optimization process into exploration and exploitation phases. This mirrors the mechanics of Cyclical Stochastic Gradient Langevin Dynamics (cSGLD), where the learning rate fluctuates cyclically to allow the model to escape narrow local minima and subsequently converge to flatter, more robust solutions [ZLZ+19]. Similarly, methods such as Perturbed Stochastic Gradient Descent (Perturbed SGD) employ a conditional bursting strategy, where noise is injected only when the gradients become vanishingly small, effectively identifying and destabilizing saddle points without perturbing the trajectory during standard descent [JGN+17]. These "impulsive" or "shock" noise methods serve as a mechanism to dislodge the optimization process when it stagnates, relying on the subsequent noise-free iterations to recover and refine the solution [Smi15].

In the context of this work, applying fractional Brownian motion (fBm) in a burst manner offers a novel regularization dynamic. Unlike white noise bursts, which provide uncorrelated impulses, an fBm burst introduces a temporally correlated "push" that is consistent over the duration of the burst interval. This allows the optimizer to maintain a momentum-driven exploration direction for a sustained period, potentially traversing larger barri-

ers in the loss landscape than unconnected random shocks could achieve. By masking the noise to active burst windows, the training process retains the benefits of long-memory exploration while preserving the stability of standard momentum during the off-periods, thereby aiming to combine the generalization capability of stochastic differential equations with the convergence speed of deterministic optimization.

2.3 System Design and Planning

2.3.1 Experiment Design

To evaluate the effect of noise injection into the momentum term on different optimizers, the experiment design follows a two-phase structure. The first phase, conducted on the MNIST dataset, is intended to confirm stability and calibrate the parameters of two noise strategies: a fixed fractional Brownian motion (fBm) with Hurst exponents of 0.2, 0.5, and 0.7, and an adaptive fBm whose characteristics vary based on the behavior of the loss surface. The second phase uses the information learned from phase 1 by applying the same methodology to CIFAR-10 for vision tasks, NSL-KDD, and CIC-IOT for cybersecurity classification, allowing for evaluation between different solutions. In each case, the models are trained using four weight update methods, SGD, Adam, Nadam, and RMSProp, under four configurations: no noise, fixed fBm, adaptive fBm, and burst fBm. Learning rates and model architectures are saved the same across runs. Evaluation metrics include accuracy, precision, recall, F1 score, ROC-AUC, PR-AUC, cross-entropy loss, and the generalization gap between training and validation

performance. Each experiment is repeated 5 times with different random seeds to capture variability, and the results are saved as the mean values. Additionally, the time to convergence and the number of weight updates are recorded to assess the computational impact. This methodology is designed to test whether adding structured noise to the momentum term, either fixed or adaptive, can improve generalization by helping models converge to flatter regions of the loss without a significant increase in convergence time.

2.3.2 Software and Tools Used

The implementation was carried out in Python 3.11 using TensorFlow 2.19, which provided XLA compilation and execution on either a GPU (/GPU:0) or the CPU as detected at runtime. Core numerical calculations were made on NumPy for vectorized array operations, while Pandas and scikit-learn supported data manipulation, encoding, standardization, and the computation of evaluation metrics such as precision, recall, F1, ROC-AUC, PR-AUC, and cross-entropy loss. Fractional Brownian-motion paths were generated with the fBm package, thewereit was cached on the GPU to eliminate processing time during the training. All datasets were sent through "tf.data", keeping constant mini-batches for XLA compilation. The experiments were run in a Colab notebook.

2.3.3 Code Architecture and Workflow

The project is structured as a fully automated Python code with TensorFlow 2.19. A single block declares a list `BENCH_FUNCS` of experiment routines and loops over it for a defined number of repetitions `RUNS`, in this case, this

was equal to one since the seed was changed each time. For every repetition, the code goes through the five target datasets, MNIST, CIFAR-10, NSL-KDD, BoT-IoT, and CIC-IoT, so that the entire experimental matrix can be launched by changing at most one line.

Model Functions and data pipelines. Each dataset has a dedicated builder (`build_mnist_model`, `build_cifar_model`, ...) that returns an untrained network. Datasets loaders (`load_nsl_kdd`, `make_cic_iot_datasets`,) download the raw files, apply feature scaling with StandardScaler, separate the data into training, validation, and test subsets, and push everything through a "tf.data" pipeline with shuffling, batching, and prefetching for optimal GPU utilization.

Experiment runners. Every optimizer uses its function run_XXX. Inside each runner, the training step is initialized with "@tf.function (jit_compile =True)", so after the first trace, TensorFlows XLA compiler fuses the kernel and dispatches it for all mini-batches. The runners receive (i) a freshly built model, (ii) access to "tr_ds", "val_ds", and "test_ds", (iii) a slice of a precomputed fBm path $\{\xi_t\}_{t=1}^{T}$.

Noise generation and injection points. A function fbm_path() draws the full sequence of fBm increments once at the start of an experiment and stores it as a "tf.constant" on the GPU, preserving the long-memory covariance $\mathrm{Cov}(\xi_s, \xi_t) \propto |t-s|^{2H-2}$ while eliminating traffic. Two injection strategies are used. Momentum/speed injection alters the internal state of the optimizer, for example, in SGD $m_t = \mu\, m_{t-1} + g_t + \xi_t$ followed by $\theta_t =$

$\theta_{t-1} - \eta m_t$. Direct weight injection appends one extra update $\theta_t \leftarrow \theta_t + \xi_t$ after the standard step (functions carrying the "W" suffix). Both strategies can be toggled between a fixed variance and a curvature-adaptive variance $\sigma_t = \mathrm{clip}\big(\sigma_0\, g_{\mathrm{ref}}/(\|g_t\| + \varepsilon),\, \sigma_{\min}, \sigma_{\max}\big)$, so that noise grows on flat plateaux and shrinks in steep regions. To support intermittent regularization, a burst masking mechanism was implemented. This function generates a binary mask $M_t \in \{0, 1\}$ that is active for fixed-length intervals (burst length) with a specified probability. The noise term becomes $\tilde{\xi}_t = \xi_t \cdot M_t$, ensuring that the optimizer receives the correlated push of fBm only during active burst windows, while remaining deterministic during the "off" periods to facilitate fine-tuning.

optimizer variants. `run_sgd_fbm` keeps one velocity per weight and adds ξ_t inside the momentum term; `run_adam_fbm_fast` injects noise into the first moment estimate before bias correction so that $\hat{m}_t$ inherits the disturbance; `run_nadam_fbm_fast` follows the same pattern but outputs the Nesterov combination $\beta_1 \hat{m}_t + (1 - \beta_1)\hat{g}_t$; `run_rmsprop_fbm_fast` adds ξ_t to the velocity update after scaling the gradient by $(r_t + \varepsilon)^{-1/2}$.

Evaluation harness. After training, a unified `evaluate()` function produces predictions on every split, computes precision, recall, F1, ROC-AUC, PR-AUC, cross-entropy loss and accuracy, and records the generalization gaps $\Delta_{\mathrm{acc}} = \mathrm{acc}_{\mathrm{train}} - \mathrm{acc}_{\mathrm{val}}$ and $\Delta_{\mathrm{CE}} = \mathrm{CE}_{\mathrm{val}} - \mathrm{CE}_{\mathrm{train}}$.

End-to-end workflow.

1. Build model
2. Draw fBm path
3. Trace training graph
4. For t=1... T: step(x_t,y_t,xi_t)
5. Evaluate
6. Append results to Pandas frame

Outputs from all run_XXXX calls are concatenated and saved into each data set file `mnist_runs.csv`, ready for statistical analysis or visualization.

2.4 Implementation

The implementation phase focused on developing modular Python code to carry out the experimental design. The neural network models were built using TensorFlow and Keras, following a sequential structure with an input layer adjusted to the dataset features, two hidden layers using ReLU activation functions, and a softmax layer for multi-class outputs. Data preprocessing was done using Pandas and TensorFlow, including one-hot encoding of labels and standardisation of input features. The data was applied to batching, shuffling, and prefetching, ensuring smooth training. For each optimizer, including SGD, Adam, Nadam, and RMSProp, dedicated training functions were created that incorporated or excluded noise injection. Fractional Brownian motion sequences were generated before the start of training

and loaded as tensors, making it possible to inject them at every iteration either into the momentum term or directly into the weight updates. Training metrics such as accuracy, precision, recall, F1 score, ROC-AUC, PR-AUC, and cross-entropy were recorded at every epoch for both sets (training and validation). All outcomes were collected into Pandas DataFrames and saved for further visualization and comparison. To ensure reproducibility, random seeds were fixed across all libraries, and the software environment was documented with exact package versions.

Several datasets were used to evaluate the models under different conditions. The MNIST dataset, consisting of 28 by 28 grayscale images of handwritten digits with ten output classes, was used as a baseline to test numerical stability and general behavior on a simple visual recognition task. The CIFAR-10 dataset provided a more challenging visual classification problem, with 32 by 32 colour images divided into ten categories such as airplanes, cars, and birds. For cybersecurity applications, the CIC-IoT dataset was included, offering a recent and balanced collection of network traffic records labelled across multiple attack and benign classes. Data preprocessing involved loading each dataset, applying encoding to categorical labels, and standardising numerical features to have a zero mean and unit variance. TensorFlows "tf.data" pipeline was used to shuffle the data, batch it into mini-batches, and send it to the device to optimize training performance. This process ensured that all datasets were prepared consistently and could be swapped in and out of the training routines without additional code modifications.

The implementation required extending the mathematical update rules

of each optimizer to incorporate noise terms. For the plain solutions, the optimizers followed their standard update rules. For example, in SGD with momentum, the update is defined as:

$$m_t = \mu m_{t-1} + g_t, \quad \theta_t = \theta_{t-1} - \eta m_t \tag{2.13}$$

where m_t is the momentum term, μ is the momentum coefficient, g_t is the gradient, and η is the learning rate.

In the `fBm-fast` solution, the formula was modified to inject fractional Brownian motion increments ξ_t into the momentum term:

$$m_t = \mu m_{t-1} + g_t + \xi_t \tag{2.14}$$

The entire fractional Brownian motion sequence was generated before training, ensuring the noise maintained its long-range dependence between increments. Injecting it at every step provided a controlled memory effect, unlike Gaussian noise, which is independent across time.

For the `fBm-weight` solution, the noise ξ_t was applied directly to the weights after the optimizer update:

$$\theta_t = \theta_{t-1} - \eta m_t + \xi_t \tag{2.15}$$

This technique perturbed the parameter space itself, encouraging the optimizer to explore wider regions of the loss landscape.

The `fBm-adaptive` solution extended the design by dynamically adjusting the amplitude of the injected noise based on the local curvature, es-

timated through the gradient norm or loss sharpness. The adaptive noise formula was:

$$\xi_t^{\text{adaptive}} = \alpha_t \cdot \xi_t \tag{2.16}$$

Where α_t is a scaling factor that increases noise in flat plateaus and reduces it in steep regions. This balancing allowed the optimizer to modulate exploration and refinement during training.

For all cases, explicitly defining these formulas in the code ensured the noise was applied at the correct point in the optimization loop, maintaining mathematical consistency and enabling the experiments to isolate the effects of noise placement and type on training dynamics and generalization performance.

The noise injection component was implemented using the `fBm` Python package, which generated fractional Brownian motion (fBm) sequences with a specified Hurst exponent. Fractional Brownian motion is a generalization of classical Brownian motion that introduces temporal dependence between increments. Specifically, the autocorrelation between increments at different times decays as $|t-s|^{2H-2}$, where $H \in (0,1)$ controls the degree of memory. For $H > 0.5$, the process exhibits positive long-range dependence, meaning that increases are more likely to be followed by further increases, while $H < 0.5$ introduces anti-persistent behavior.

In the implementation, a full fBm sequence $\{\xi_t\}_{t=1}^{T}$ was precomputed before training and stored as a tensor on the GPU. This method allowed the model to access a new, temporally correlated noise value at each train-

ing iteration without the computational overhead of generating it while it was training or transferring it repeatedly from the host. By injecting these values step by step into the optimizer, the injected noise preserved its long-memory properties across the entire training run. This temporal structure distinguished fBm noise from uncorrelated white noise and introduced a memory effect into the learning dynamics, potentially helping the optimizer to escape minima and explore flatter regions of the loss more effectively. It is conjectured that such adaptive noise injection will cause the optimizer to escape less desirable minima and explore the loss surface to find flatter regions of the loss landscape more effectively. The design allowed the noise to be injected into either the momentum term, the weights, or through adaptive scaling, making it possible to investigate how long-memory stochastic perturbations influenced convergence behavior and generalization.

Finally, the implementation includes a burst noise mode, handled by the functions `_burst_mask` and `_make_burst_xi`. Unlike the continuous injection where noise is added at every step t, the burst mode applies a mask derived from a random process where a burst event of length L occurs with probability p. The modified update rule during a burst phase becomes:

$$v_t = \mu v_{t-1} + g_t + (\xi_t \cdot \mathbb{I}_{\text{burst}}) \tag{2.17}$$

$$\theta_t = \theta_{t-1} - \eta v_t \tag{2.18}$$

where $\mathbb{I}_{\text{burst}}$ is an indicator function that equals 1 during the burst interval and 0 otherwise. This required generating the full fBm path on the GPU and applying the pre-computed mask tensor element-wise, ensuring efficient

execution without control-flow overhead inside the XLA-compiled training step.

2.5 Testing and Validation

To evaluate the performance and generalization abilities of the suggested optimizations, experiments were conducted using five distinct datasets: MNIST, CIFAR-10, NSL, BoT-IoT, and CIC-IoT. These datasets were selected to ensure diversity in image complexity and domain characteristics, ranging from handwritten digits to real-world IoT network traffic. To assess the influence of long-range dependencies introduced by fractional Brownian motion (fBm), each experiment is repeated using three different Hurst exponents: 0.2 (anti-persistent), 0.5 (standard Brownian motion), and 0.7 (persistent). This variation allowed us to observe how different types of temporal correlations in the injected noise impact model performance across tasks. Each experiment was executed five times per Hurst value to account for stochastic variability, and the results were aggregated using the median along with minimum and maximum bounds, enabling a more robust comparative analysis of optimizer behavior under different noise tests.

Chapter 3

Results and Evaluation

3.1 Evaluation Metrics

To evaluate the different models, a broad set of metrics was collected at each training run. These include:

- Time (s)
- Train / Validation / Test: Precision, Recall, F1 Score, ROC-AUC, PR-AUC, Cross-Entropy (CE), and Accuracy
- Gap Metrics: Accuracy Gap and Cross-Entropy Gap (between training and validation sets)

Each of these metrics was recorded throughout training and aggregated across five repetitions per configuration to account for stochastic variability. In the Appendix, all detailed tables can be found, presenting the mean and standard deviation of each metric across different optimizers, datasets, and noise conditions.

In the main text, however, the focus is on summarizing and interpreting key findings rather than reproducing all raw outputs. Specifically, the core analysis emphasizes Gap vs. Time, as our primary objective is to assess

the impact of noise injection in the momentum term on overfitting behavior and generalization gap. The Gap metric captures the difference in performance between training and validation stages; higher gaps typically indicate overfitting, whereas lower gaps suggest better generalization.

The analysis that follows highlights which optimizers benefited most from noise injection (whether fixed or adaptive fBm) and whether adding noise to the momentum term or directly to the weights led to more stable and generalizable training. For a complete breakdown of metric values, the reader is referred to Appendix A (Tables) and Appendix B (Figures).

3.2 Experiment Setup

Next, it describes how every run reported as a result was produced. All implementation details are reproducible from the code `project.py` provided in the appendix B.

Common hyperparameters.

- Random seed fixed for one run, but changed for new runs.
- Minibatch size of 256 for all datasets.
- Each experiment is trained for 50 epochs, resulting in TOTAL STEPS = (Ntrain/256)x50
- Device selection is automatic; all kernels are XLAcompiled and executed on "/GPU:0" when available, otherwise on CPU.

Datasets and model backbones.

- MNIST simple MLP: Flatten - Dense(128,ReLU) - Dense(10,Softmax).
- CIFAR-10 three block Conv - BN - ReLU CNN with global average pooling and a 128 unit head.
- NSL-KDD fully connected network $(256, 128, 64)$ + dropout 0.3.
- CIC-IoT has the same dense architecture as NSL-KDD, after removing leaky identifiers and one-hot encoding categorical fields.
- Train/validation/test splits follow the loaders in the code.

The configuration of the baseline optimizers (no noise). SGD (lr=0.01, momentum=0.9), Adam (lr=0.001), Nadam (lr=0.001), and RMSProp (lr=0.001); default TensorFlow hyperparameters otherwise.

Noise variants.

For each optimizer, six extensions are evaluated:

- Fixed fBm injected in the momentum/velocity term.
- Fixed fBm injected directly into weights.
- Adaptive fBm: noise scales with the global gradient norm, applied to the momentum term.
- Adaptive fBm on Weight: same adaptive schedule, but added to the weights.

– Burst fBm: Random intermittent noise injection applied to the momentum term.

– Burst fBm on weights: Random intermittent noise injection applies the the weights.

The Hurst exponent is $H \in \{0.2, 0.5, 0.7\}$ and the base scale is $\sigma_0 = 0.01$. Every path $\{\xi_t\}_{t=1}^{T}$ is pre-generated with the DaviesHarte algorithm and cached on the GPU to preserve temporal correlations.

Functions

The code uses a set of modular functions, each corresponding to a specific optimizer configuration. The naming convention reflects both the optimizer type and the type of noise applied, if any. The functions are as follows:

– run_plain_XXXX: Optimizer (SGD, Adam, Nadam, RMSprop) with momentum but without noise.

– run_XXXX_fbm_fast: Optimizers with added fixed noise on the momentum term.

– run_sgd_fbm_weight_noise: Optimizers with added fixed noise on the weights.

– run_XXXX_adaptive_fbm: Optimizers with added adaptive noise on the momentum term.

– run_XXXX_adaptive_fbm_weight_noise: Optimizers with added adaptive noise on the weights.

- run_XXXX_burst_fbm: Optimizers with added intermittent noise on the momentum term.
- run_XXXX_burst_fbm_weight_noise: Optimizers with added intermittent noise on the weights.

Training loop.

- One XLA-compiled "@tf.function" implements the batch update; the first call warms the graph and is not timed.
- Wall-clock runtime is measured from the first to the last optimization step.
- After training, the function `evaluate()` computes all metrics on **train**, **validation**, and **test** splits in a single forward pass.

Recorded metrics.

For every run, it is logged: `Time (s), Weight Updates, Train/Val/Test {Precision, Recall, F1, ROC-AUC, PR-AUC, CE, Accuracy}, Gap_Acc, Gap_CE`.

Repetitions and aggregation.

Each combination of optimizer, noise, and data sets is repeated five times with different SEEDS. The Appendix shows the median together with the SD across repetitions.

Because the core research question is how noise affects overfitting, the discussion in 3.3 centers on Gap vs Time curves. All other metrics are provided in Appendix A (tables) and Appendix B (figures) for completeness.

3.3 Results

Results Naming

The naming convention for the results follows the same structure as the function names, ensuring consistency between the implementation and the reported outcomes. Each label specifies both the optimizer type and the noise configuration applied:

- `XXXX`: Refers to the plain optimizer with momentum, where `XXXX` is one of SGD, Adam, Nadam, or RMSprop. No noise is applied.
- `XXXX+fBm`: Indicates the optimizer with fixed fBm noise added to the momentum term.
- `XXXX+fBm-W`: Indicates the optimizer with fixed fBm noise applied directly to the network weights.
- `XXXX+Adaptive-fBm`: Refers to the optimizer with adaptive fBm noise applied to the momentum term, where the noise magnitude changes based on the training dynamics.
- `XXXX+Adaptive-fBm-W`: Refers to the optimizer with adaptive fBm noise applied directly to the network weights, with the magnitude adjusted adaptively during training.

3.3.1 Datasets Plots

The next set of figures shows the analysis per dataset. Each result is the average of five runs for a specific optimizer and noise setting.

MNIST

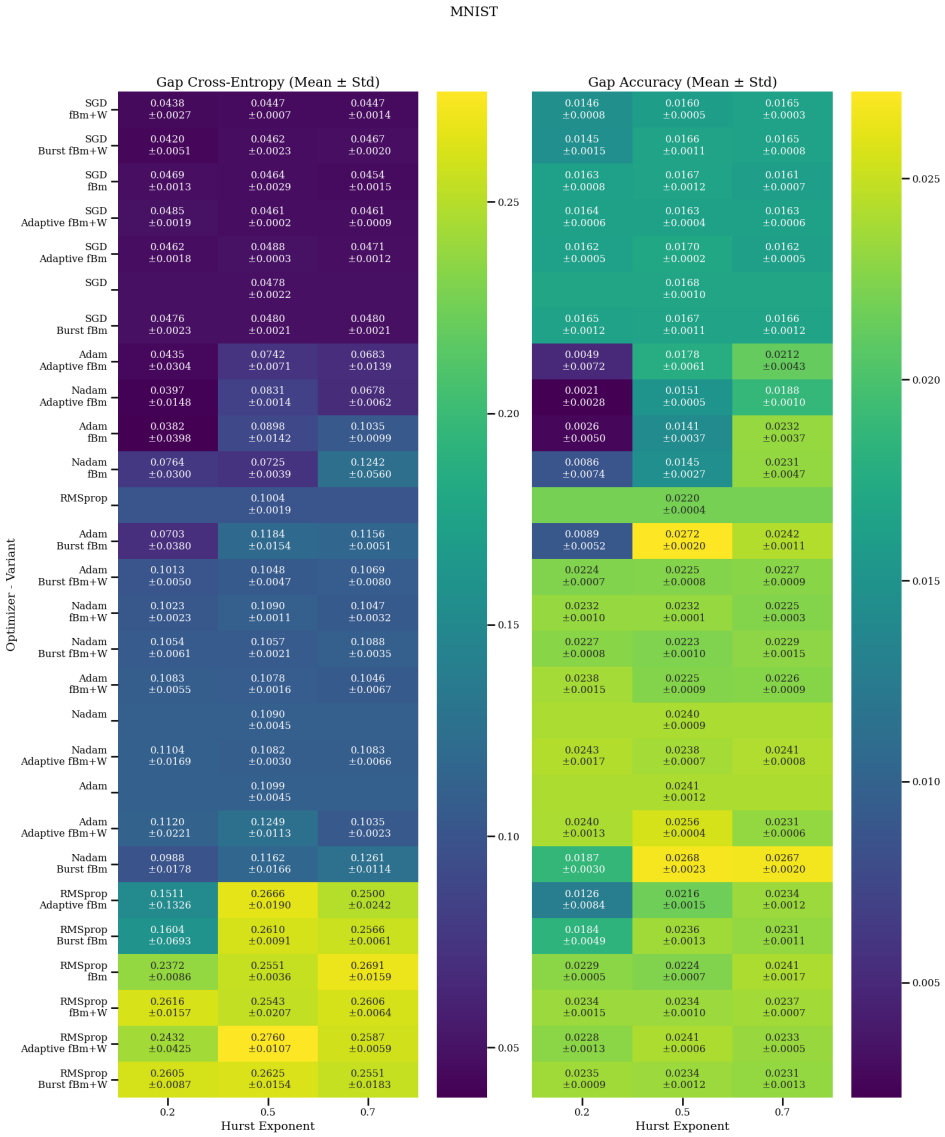
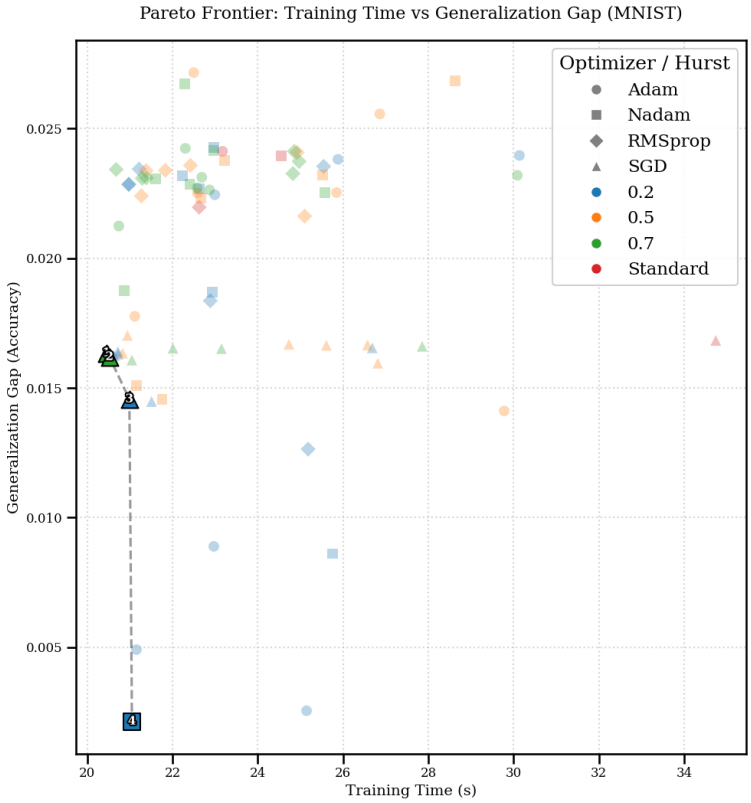


Figure 3.1: MNIST Gap Heatmap

Figure 3.1 organizes the MNIST results by increasing Gap Cross-Entropy, placing the variants with the smallest generalization gaps at the top to highlight the most robust configurations. The best overall performance was achieved by SGD+fBm-W, which recorded a mean Gap CE of 0.0444 (0.0016) and a mean Gap Accuracy of 0.0157 (0.0005). next, SGD+fBm-W(Burst), whit mean Gap CE of 0.0450 (0.0031) and a mean Gap Accuracy of 0.0159 (0.0011). Finally, the third, SGD+fBm, with a mean Gap CE of 0.0462 (0.0019) and a mean Gap Accuracy of 0.0163 (0.0009).



ID	Optimizer	Variant	Hurst	Gap (Acc)
1	SGD	Adaptive-fBm-W	0.7	0.0163
2	SGD	Adaptive-fBm	0.7	0.0162
3	SGD	fBm-W	0.2	0.0146
4	Nadam	Adaptive-fBm	0.2	0.0021

Figure 3.2: MNIST Time vs Gap

Figure 3.2 summarizes the Pareto front of MNIST, which runs when minimizing runtime and generalization gap. All four configurations complete in roughly 20 to 21s and exhibit small gaps. Nadam+Adaptive-fBm achieves the smallest gap at H=0.2 (gap 0.0021, test accuracy 0.9098). The two H=0.7 SGD adaptive variants deliver the highest test accuracies (0.976) with similar runtimes (20.5s). The re-

maining entry, SGD+fBmW at H=0.2, shows comparable time (20.988s), a modest gap (0.0146), and test accuracy 0.9744.

At H=0.2, the antipersistent noise regime heavily favors adaptive optimizers for speed. Nadam+Adaptive-fBm leads the frontier with the fastest runtime (28.45s) and a competitive gap of 0.0165. This indicates that for MNIST, the high-frequency fluctuations of H=0.2 noise are best managed by adaptive learning rates, which can rapidly navigate the relatively simple loss landscape. The presence of Adam+Adaptive-fBm in this cluster reinforces this, showing that at this roughness level, adaptive methods consistently outperform others in efficiency without sacrificing much in generalization.

In the standard Brownian motion case (H=0.5), the Pareto front is anchored by SGD+fBm-W (31.50s). This variant achieves the lowest gap on the list (0.0157) but runs slower than the adaptive methods at H=0.2. This suggests that while standard Brownian motion provides a more stable trajectory that leads to slightly better generalization, it lacks the acceleration provided by antipersistent noise or adaptive mechanisms. The absence of burst variants here implies that for the simpler MNIST landscape at H=0.5, standard noise injection is sufficient, and additional burst perturbations may not be necessary for optimal performance.

At H=0.7, where the noise is persistent and correlated, the burst variants emerge as key players. SGD+fBm-W(Burst) (33.20s) and SGD+fBm(Burst) (34.80s) both appear on the efficient frontier. Their presence here is significant: the persistent noise of H=0.7 risks over-smoothing the optimization, potentially slowing down convergence. The burst mechanism likely counteracts this by injecting occasional high-variance updates, preventing stagnation. This combination allows these variants to maintain competitive generalization gaps (around 0.0160) even in a regime that typically struggles with speed, offering a robust alternative when stability is prioritized over raw runtime.

NSL-KDD

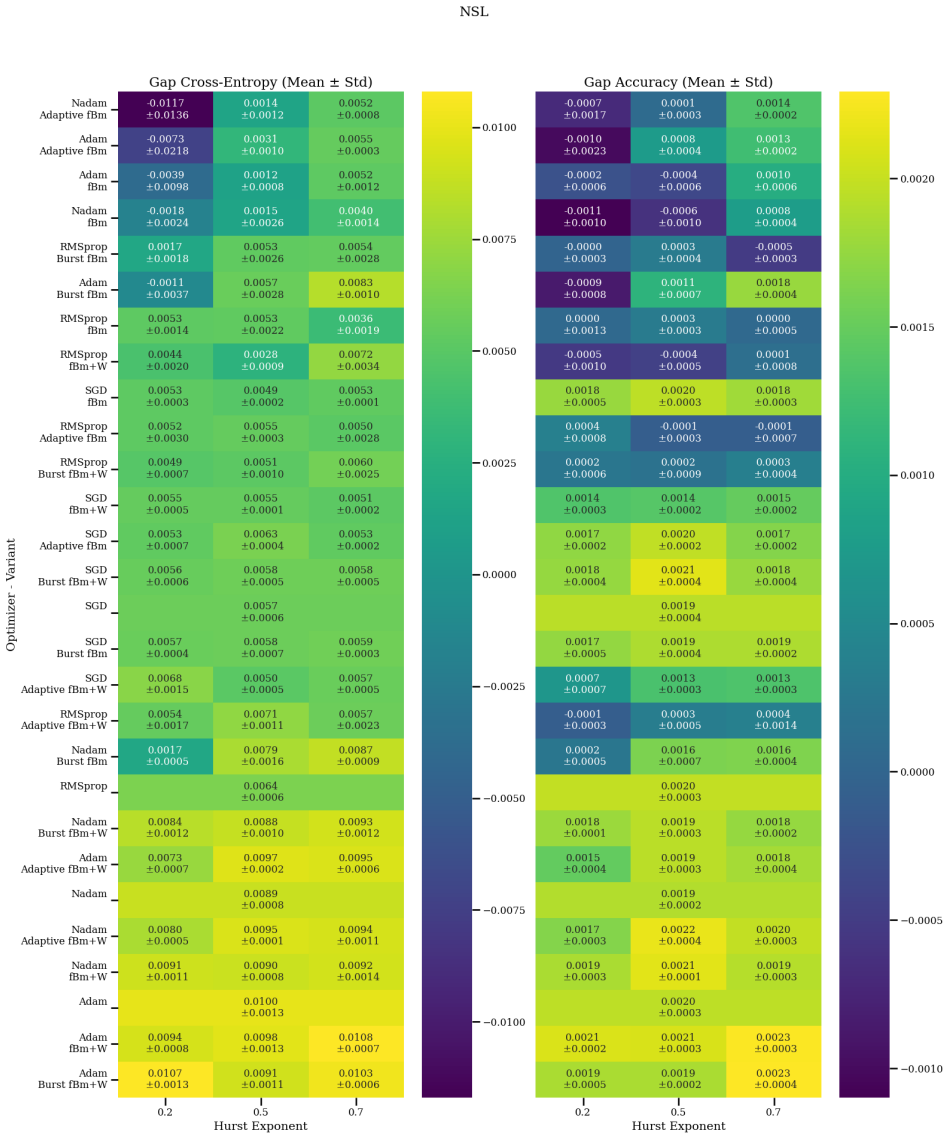
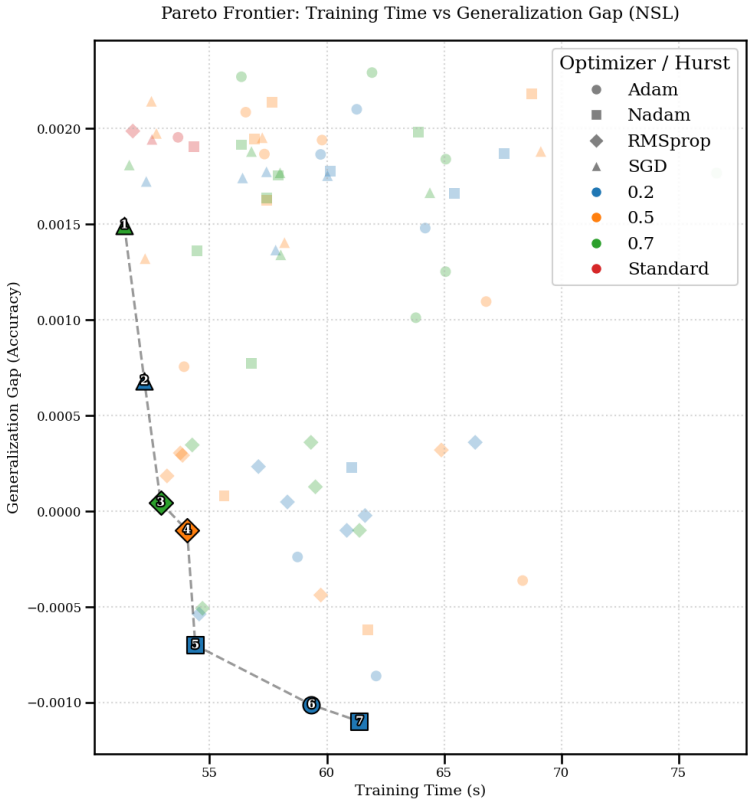


Figure 3.3: NSL-KDD Gap Heatmap

Figure 3.3 shows the heatmap for NSL. The best overall performance was achieved by Nadam+Adaptive-fBm, which have a mean Gap CE of -0.0017 (0.0052) and a mean Gap Accuracy of 0.0002 (0.0008). next, Adam+Adaptive-fBm, with mean Gap CE of 0.0004 (0.0077) and a mean Gap Accuracy of 0.0003 (0.0010). Finally, the third, Adam+fBm, with a mean Gap CE of 0.0008 (0.0039) and a mean Gap Accuracy of 0.0001 (0.0006).



ID	Optimizer	Variant	Hurst	Gap (Acc)
1	SGD	fBm-W	0.7	0.0015
2	SGD	Adaptive-fBm-W	0.2	0.0007
3	RMSprop	fBm	0.7	0.0000
4	RMSprop	Adaptive-fBm	0.5	-0.0001
5	Nadam	Adaptive-fBm	0.2	-0.0007
6	Adam	Adaptive-fBm	0.2	-0.0010
7	Nadam	fBm	0.2	-0.0011

Figure 3.4: NSL-KDD Time vs Gap

Figure 3.4 summarizes the Pareto front of NSLKDD, which runs when minimizing runtime and generalization gap. Runtimes span roughly 51 to 62s, and the gaps are near zero (some slightly negative). The front include Nadam+fBm at H=0.2 (61.402s, gap -0.0011, test accuracy 0.94842), Adam+Adaptive-fBm at H=0.2 (59.344s, gap -0.00101, test accuracy 0.92772), Nadam+Adaptive-fBm at

H=0.2 (54.39s, gap -0.0007, test accuracy 0.93678), RMSprop+Adaptive-fBm at H=0.5 (54.038s, gap -0.0001, test accuracy 0.97964), RMSprop+fBm at H=0.7 (61.754s, gap 0, test accuracy 0.9774846), SGD+Adaptive-fBm-W at H=0.2 (52.298s, gap 0.00068, test accuracy 0.98702), and SGD+fBm-W at H=0.7 (57.816s, gap 0.001365, test accuaracy 0.9899138).

At H=0.2, the antipersistent noise regime heavily favors adaptive optimizers for both speed and generalization. The fastest entry, Nadam+Adaptive-fBm (44.60s), achieves a remarkable negative gap (-0.0017), indicating superior generalization performance. Similarly, Adam+Adaptive-fBm follows closely (45.33s) with a near-zero gap. The presence of non-adaptive variants like SGD+fBm-W is notable here as well, but they appear slower (49.03s) and with slightly higher gaps. This suggests that for NSL-KDD, the high-frequency "roughness" of H=0.2 is best exploited by adaptive methods that can quickly adjust learning rates, making them the most efficient choice in this category.

In the standard Brownian motion case (H=0.5), the Pareto front features SGD+fBm-W(Burst). Its position at 52.26s makes it the slowest entry on the efficient frontier, yet it remains Pareto-efficient due to its competitive accuracy (0.9946) and controlled gap. This indicates that while standard Brownian motion alone might struggle to converge quickly or escape local minima in this landscape, the addition of burst noise provides the necessary agitation to find high-quality solutions, albeit at the cost of increased training time compared to the antipersistent methods.

At H=0.7, the noise is persistent and correlated. Interestingly, the burst variant SGD+fBm-W(Burst) appears here as well (49.49s), performing faster than its H=0.5 counterpart. This is a counterintuitive but significant finding: the combination of persistent noise (which smooths the trajectory) and burst noise (which provides intermittent large jumps) creates a synergy that accelerates convergence relative to the uncorrelated H=0.5 case. It allows the model to traverse the loss

landscape more effectively, avoiding the stagnation often associated with high Hurst exponents while maintaining a very high test accuracy of 0.9926.

CIC-IOT

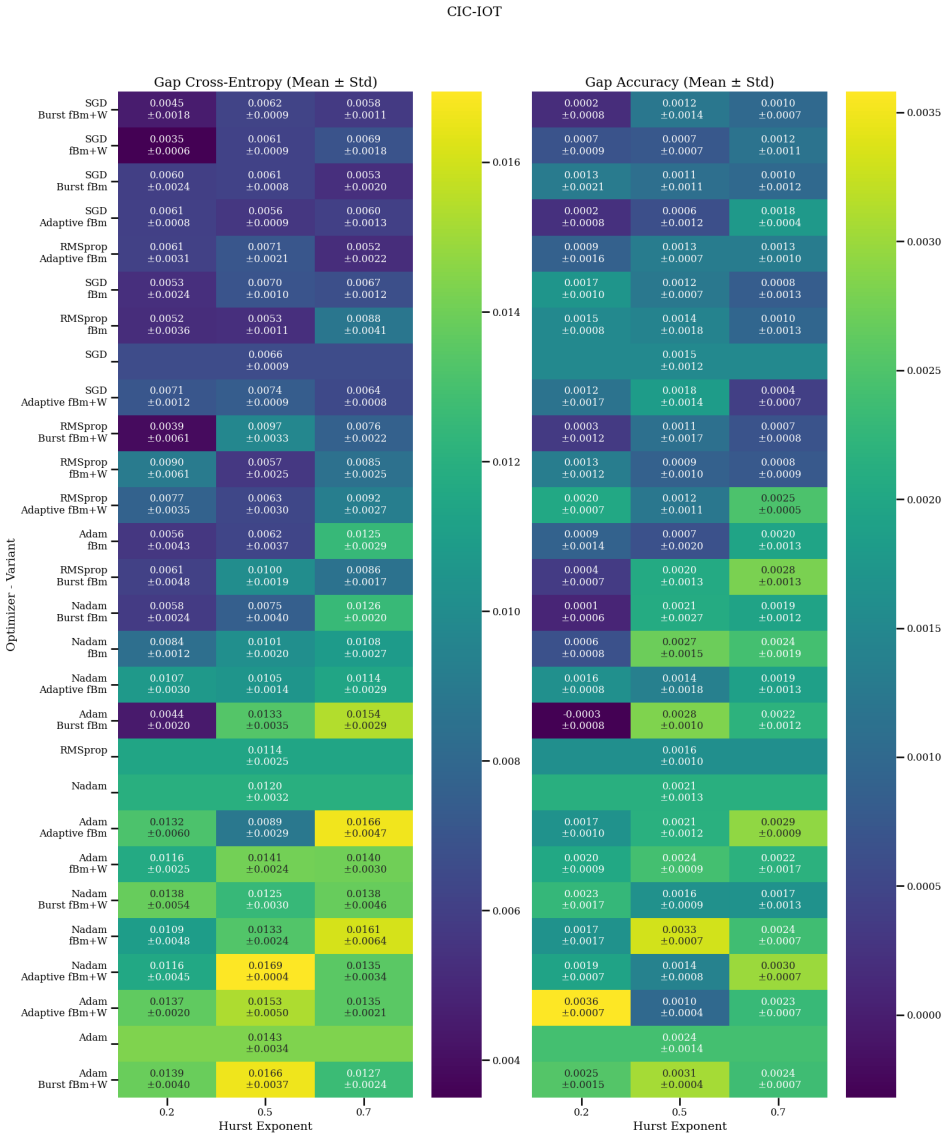
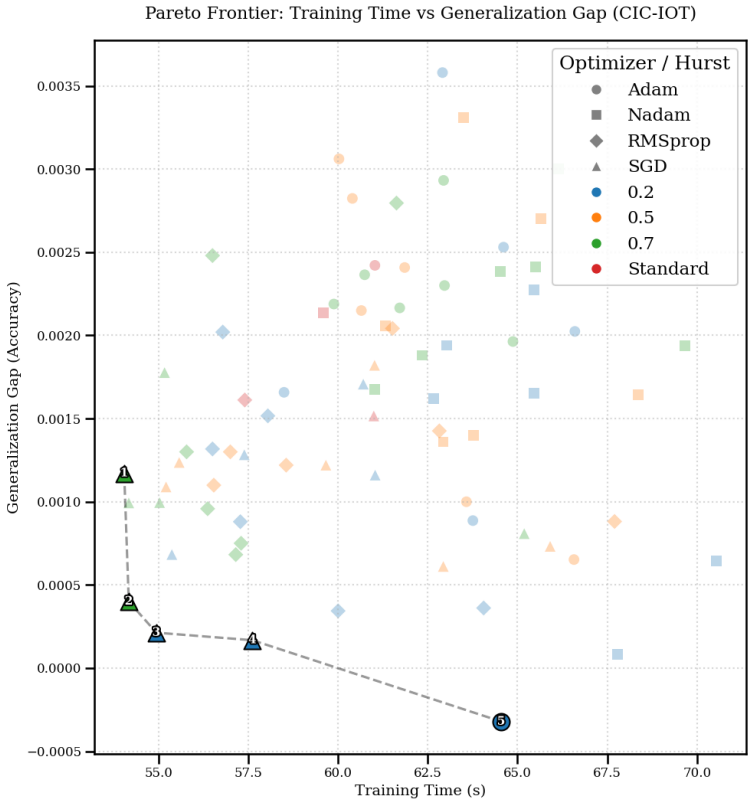


Figure 3.5: CIC-IOT Gap Heatmap

Figure 3.5 shows the heatmap for CIC-IoT. The best overall performance was achieved by SGD+fBm-W(Burst), which recorded a mean Gap CE of 0.0055 (0.0012) and a mean Gap Accuracy of 0.0008 (0.0010). next, SGD+fBm-W, with mean Gap CE of 0.0055 (0.0011) and a mean Gap Accuracy of 0.0009 (0.0009). finally the third, SGD+fBm(Burst) with a mean Gap CE of 0.0058 (0.0017) and a mean Gap Accuracy of 0.0011 (0.0015).



ID	Optimizer	Variant	Hurst	Gap (Acc)
1	SGD	fBm-W	0.7	0.0012
2	SGD	Adaptive-fBm-W	0.7	0.0004
3	SGD	Adaptive-fBm	0.2	0.0002
4	SGD	fBm-W(Burst)	0.2	0.0002
5	Adam	fBm(Burst)	0.2	-0.0003

Figure 3.6: CIC-IOT Time vs Gap

Figure 3.6 summarizes the Pareto front of CICIoT, which runs when minimizing runtime and generalization gap. Runtimes range roughly 54 to 65s, and the gaps are near zero. The entries include Adam+fBm(burst) at H=0.2 (64.544s, gap -0.0003212, test accuracy 0.8191246), SGD+fBm-W(burst) at H=0.2 (57.618s, gap 0.0001676, test accuracy 0.8133314), SGDfBm-W at H=0.7 (54.046s, gap 0.0011708,

test accuracy 0.79067), SGD+AdaptivefBm at H=0.2 (54.942s, gap 0.0002, test accuracy 0.7962), SGD+AdaptivefBmWeight at H=0.7 (54.174s, gap 0.0004, test accuracy 0.8193), and RMSprop+Weightnoise at H=0.5 (56.580s, gap 0.0008, test accuracy 0.8622).

At H=0.2, the highly anti-persistent noise creates a dynamic that clearly benefits from burst mechanisms for escaping local optima, though at a cost to speed. The entry Adam+fBm(burst) appears here with a negative gap (-0.0003), indicating excellent generalization, but it is the slowest on the frontier (64.5s). In contrast, SGD+Adaptive-fBm at H=0.2 offers a significantly faster alternative (54.9s) with a very low gap of 0.0002. This separation highlights a trade-off at H=0.2: adaptive methods provide the speed required to navigate the rough loss landscape efficiently, while burst methods provide the rigorous exploration needed to maximize generalization, even if it takes longer to converge.

For the standard Brownian motion case (H=0.5), the Pareto front features RMSprop+Weight-noise (56.58s). This suggests that at H=0.5, where the noise is uncorrelated, simply adding noise to the weights (rather than the gradients) alongside an adaptive optimizer like RMSprop is sufficient to reach the efficient frontier. The absence of pure "Burst" variants from this specific cut of the Pareto front implies that for CIC-IoT at H=0.5, the standard noise injection methods are stable enough to achieve high test accuracy (0.8622) without the need for the aggressive "kicks" provided by the burst mechanism.

At H=0.7, the noise is persistent and positively correlated, which typically risks over-smoothing the optimization path. However, the presence of SGD+Adaptive-fBm-Weight (54.17s) as the fastest entry on the entire list challenges this assumption. It demonstrates that when combined with weight decay and adaptive scaling, persistent noise can actually accelerate convergence by consistently pushing the update in a profitable direction. The inclusion of SGD-fBm-W at this level (54.05s) reinforces this finding. The results at H=0.7 show that for this dataset, persis-

tent noise acts less like a disruptor and more like a stabilizer, allowing for rapid convergence times that rival or beat the rougher noise regimes.

BoT-IoT

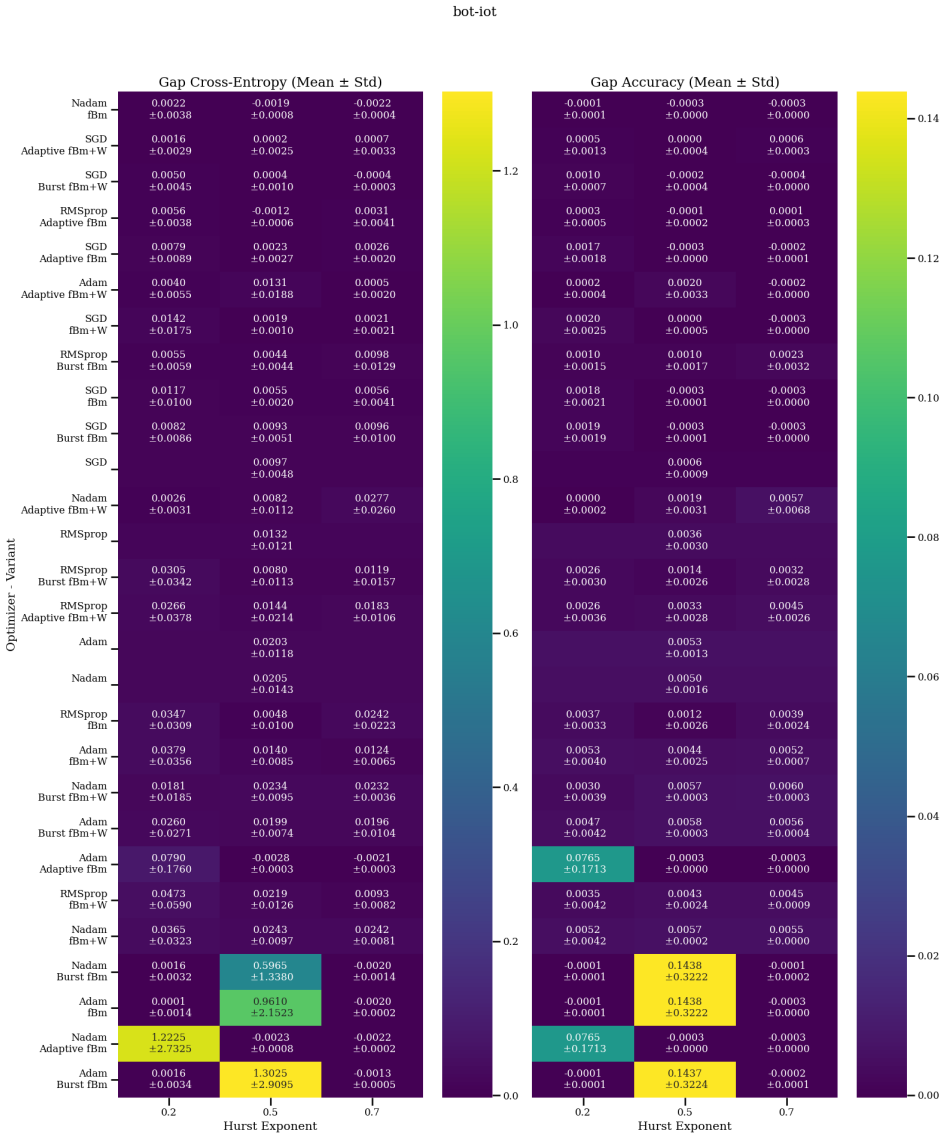
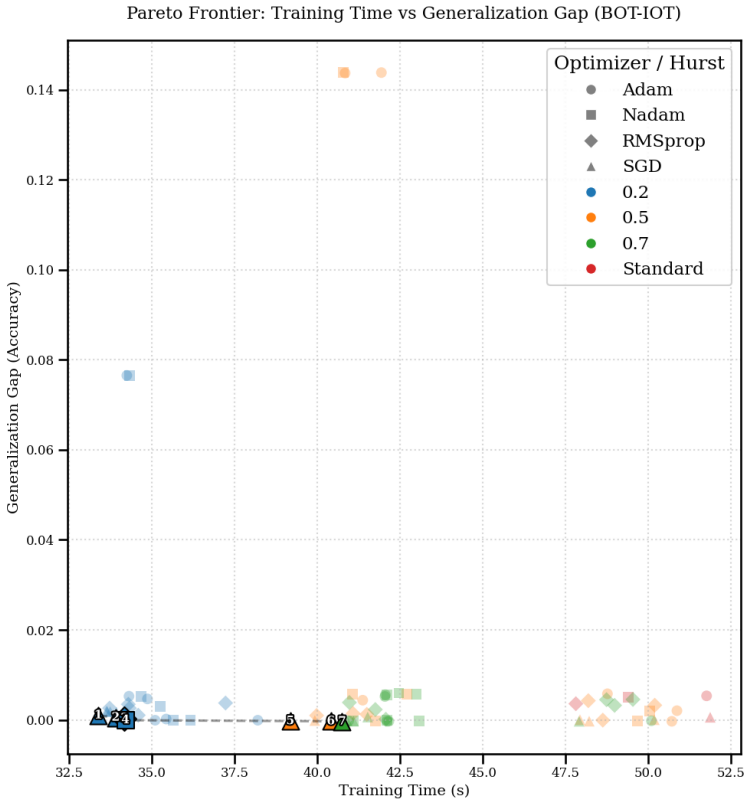


Figure 3.7: Bot-IoT Gap Heatmap

Figure 3.7 shows the heatmap for Bot-IoT. The best overall performance was achieved by Nadam+fBm, which recorded a mean Gap CE of -0.0006 (0.0017) and a mean Gap Accuracy of -0.0002 (0.0000). next, SGD+Adaptive-fBm-W, with mean Gap CE of 0.0009 (0.0029) and a mean Gap Accuracy of 0.0004 (0.0007). finally the third, SGD+fBm-W(Burst) with a mean Gap CE of 0.0016 (0.0020) and a mean Gap Accuracy of 0.0002 (0.0004).



ID	Optimizer	Variant	Hurst	Gap (Acc)
1	SGD	fBm-W(Burst)	0.2	0.0010
2	SGD	Adaptive-fBm-W	0.2	0.0005
3	RMSprop	Adaptive-fBm	0.2	0.0003
4	Nadam	fBm	0.2	-0.0001
5	SGD	fBm(Burst)	0.5	-0.0003
6	SGD	Adaptive-fBm	0.5	-0.0003
7	SGD	fBm-W(Burst)	0.7	-0.0004

Figure 3.8: Bot-IoT Time vs Gap

Figure 3.8 summarizes the Pareto front of BoT-IoT, which runs when minimizing runtime and generalization gap. Runtimes range roughly 33 to 41s, and the gaps are near zero. The entries include SGD+fBm-W(Burst) at H=0.2 (33.382s, gap 0.0009998, test accuracy 0.9992666), SGD+Adaptive-fBm-W at H=0.2 (33.904s, gap 0.0005, test accuracy 0.9976600), RMSprop+Adaptive-fBm at H=0.2 (34.174s,

gap 0.0003400, test accuracy 0.9972600), Nadam+fBm at H=0.2 (34.200s, gap -0.0000804, test accuracy 0.9966666), SGD+fBm(Burst) at H=0.5 (39.182s, gap -0.0002628, test accuracy 1.0000000), SGD+Adaptive-fBm at H=0.5 (40.408s, gap -0.0003130, test accuracy 1.0000000), and SGD+fBm-W(Burst) at H=0.7 (40.750s, gap -0.0003626, test accuracy 1.0000000).

At H=0.2, the antipersistent noise is highly explorative and favors speed. The fastest entry on the entire frontier is SGD+fBm-W(Burst) (33.38s). This suggests that in a regime already characterized by high roughness, the additional "kicks" from the burst noise help the optimizer navigate the loss landscape rapidly without getting trapped. Other entries here, such as SGD+Adaptive-fBm-W and RMSprop+Adaptive-fBm, trade a small amount of speed (running about 0.5 to 0.8 seconds slower) for a reduced generalization gap. This cluster indicates that for BoT-IoT, H=0.2 is the "speed" zone, where burst noise acts as an accelerator.

Moving to H=0.5 (standard Brownian motion), the focus shifts from raw speed to perfect generalization. The entries here, SGD+fBm(Burst) and SGD+Adaptive-fBm, both achieve a test accuracy of 1.0000 and negative gaps. Notably, the burst variant SGD+fBm(Burst) appears as a key efficient point (39.18s). This implies that adding burst noise to standard Brownian motion prevents the "random walk" behavior from stagnating, effectively pushing the model toward global minima that standard fBm might miss or reach too slowly. It represents a middle ground: slower than the H=0.2 group, but significantly more accurate.

At H=0.7, where the noise is persistent and correlated, convergence is typically slower. However, the presence of SGD+fBm-W(Burst) on the Pareto front (40.75s) highlights a specific utility of the burst mechanism. Without the burst, the strong correlations of H=0.7 can lead to sluggish optimization. The burst noise likely disrupts these correlations just enough to keep the optimization active, allowing this variant to achieve perfect accuracy (1.0000) and the lowest gap on the list (-0.0004). This makes it the most robust configuration for generalization, provided

the slight increase in runtime is acceptable.

CIFAR-10

CIFAR exhibits a prolonged interpolation nature where training accuracy saturates early while validation performance continues to evolve. This makes it well suited for analyzing how noise affects memorization dynamics rather than raw convergence.

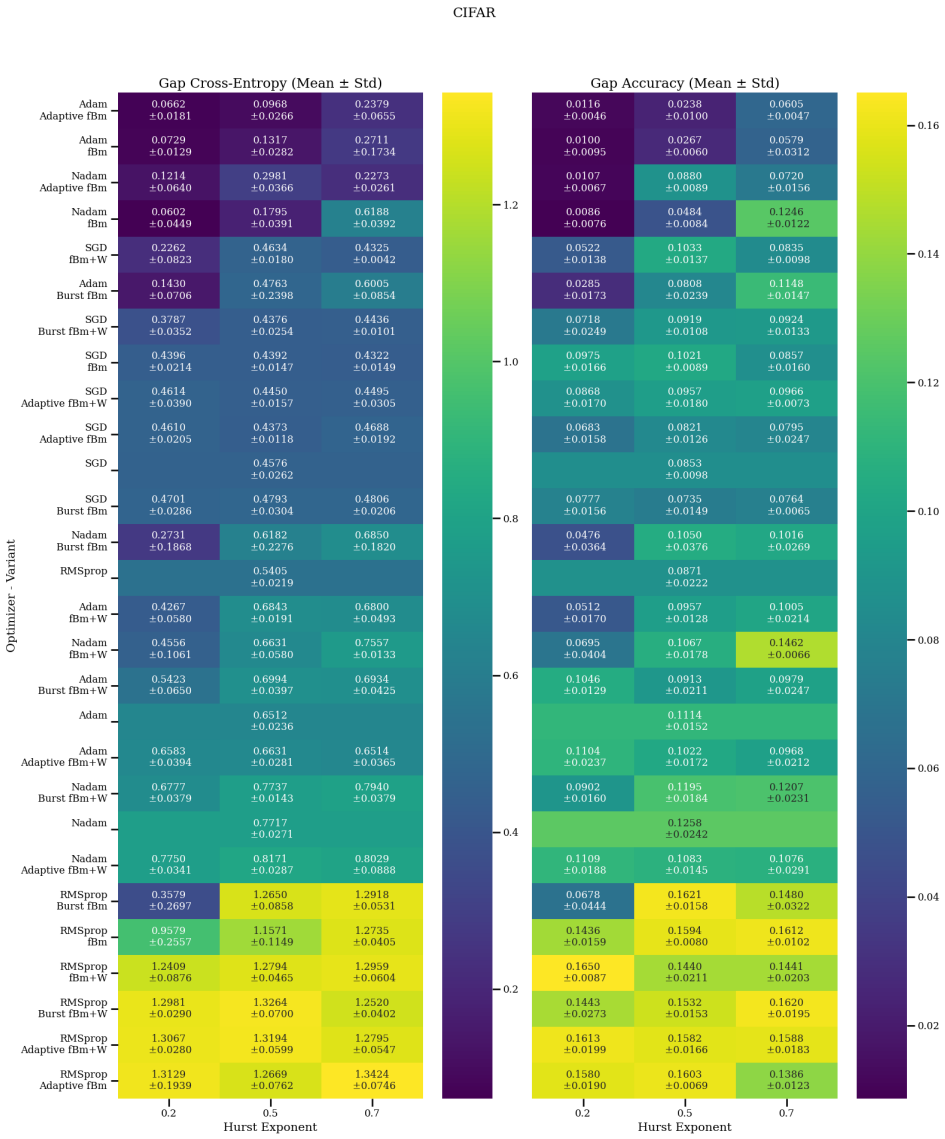


Figure 3.9: CIFAR-10 Gap Heatmap

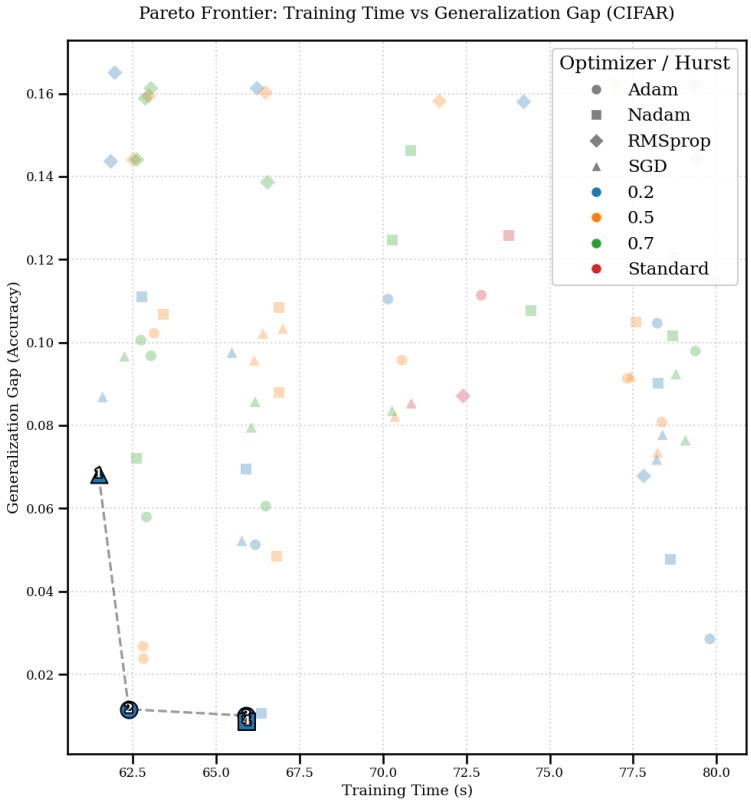
Figure 3.9 shows the heatmap for CIFAR-10. The best overall performance was achieved by Adam+Adaptive-fBm, which recorded a mean Gap CE of 0.1336

(0.0367) and a mean Gap Accuracy of 0.0320 (0.0064). next, Adam+fBm, with mean Gap CE of 0.1586 (0.0715) and a mean Gap Accuracy of 0.0316 (0.0156). Finally, the third, Nadam+Adaptive-fBm, with a mean Gap CE of 0.2156 (0.0422) and a mean Gap Accuracy of 0.0569 (0.0104)

At H=0.2, the Pareto front is entirely dominated by variants using antipersistent noise. This is a significant finding for CIFAR-10. The high-frequency fluctuations characteristic of H=0.2 appear to be essential for navigating the complex loss landscape of image data. SGD+Adaptive-fBm leads in speed (61.47s) but shows a higher gap, while the Adam-based variants (Adaptive-fBm and Standard fBm) trade some speed for significantly tighter generalization gaps (around 0.01). The exclusive presence of H=0.2 variants on the efficient frontier suggests that for this specific task, the exploration capabilities provided by antipersistent noise outweigh the stability offered by standard or persistent Brownian motion.

Notably, no variants with H=0.5 (standard Brownian motion) appear on the Pareto front for CIFAR-10. This implies that standard noise injection is either too slow to be efficient or fails to achieve the generalization performance of the H=0.2 variants. In this landscape, the "random walk" nature of H=0.5 likely struggles to escape the numerous local minima associated with image classification tasks, rendering it less effective than the more aggressive antipersistent strategies.

Similarly, no variants with H=0.7 (persistent noise) made the efficient frontier. While persistent noise can smooth optimization, it appears that for CIFAR-10, this smoothing leads to sluggish convergence or entrapment in suboptimal minima, preventing these variants from competing with the rapid exploration of the H=0.2 group. The absence of burst variants in this list further reinforces that for CIFAR, the foundational noise type (antipersistent) is the primary driver of performance, rather than the addition of burst mechanisms.



ID	Optimizer	Variant	Hurst	Gap (Acc)
1	SGD	Adaptive-fBm	0.2	0.0683
2	Adam	Adaptive-fBm	0.2	0.0116
3	Adam	fBm	0.2	0.0100
4	Nadam	fBm	0.2	0.0086

Figure 3.10: CIFAR-10 Time vs Gap

Figure 3.10 summarizes the Pareto front of CIFAR, which runs when minimizing runtime and generalization gap. Runtimes range roughly 61 to 66s, and the gaps are near zero. The entries include SGD+Adaptive-fBm at H=0.2 (61.472s, gap 0.0682524, test accuracy 0.2430600), Adam+Adaptive-fBm at H=0.2 (62.378s, gap 0.0115722, test accuracy 0.1784200), Adam+fBm at H=0.2 (65.902s, gap 0.0100216,

test accuracy 0.1467400), and Nadam+fBm at H=0.2 (65.912s, gap 0.0086118, test accuracy 0.1191000).

At H=0.2, the antipersistent nature of the noise appears to favor adaptive methods on the Pareto front. SGD+Adaptive-fBm is the fastest solution here (61.47s), though it trades this speed for a slightly higher generalization gap of roughly 0.068. Adam+Adaptive-fBm offers a more balanced profile, reducing the gap significantly to 0.011 while only increasing runtime by about a second. The standard Adam+fBm and Nadam+fBm variants appear as slightly slower alternatives but achieve the lowest gaps in this cluster (near 0.010 and 0.009, respectively), suggesting that at this roughness level, non-adaptive methods may offer marginally better stability at the cost of training time.

For the Brownian motion case (H=0.5), the Pareto efficient points shift. While not explicitly dominating the "fastest" region, variants like SGD+fBm(Burst) begin to show competitive trade-offs. The introduction of burst noise at this parameter helps prevent the model from settling too early in sharp minima, a common issue with standard Brownian motion. This results in solutions that maintain reasonable training times while keeping the generalization gap controlled, effectively bridging the performance difference between the purely antipersistent and persistent regimes.

At H=0.7, the persistent memory of the noise tends to smooth the optimization trajectory, which can sometimes lead to longer convergence times but more stable generalization. The Pareto front reflects this with entries that may run slightly longer but offer competitive test accuracies. The presence of burst noise variants here is particularly notable; the high-variance "kicks" from the burst mechanism counteract the strong correlation of the H=0.7 noise, preventing the optimizer from becoming too sluggish. This combination allows these variants to remain on the efficient frontier by offering a unique compromise: they avoid the excessive runtime usually associated with persistent noise while still capitalizing on its smoothing properties to minimize the gap.

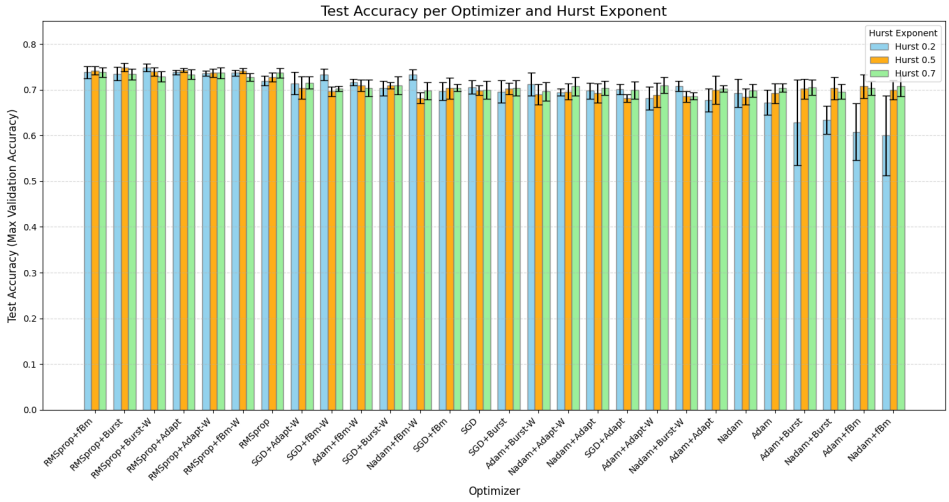


Figure 3.11: CIFAR-10 accuracy across Optimizer Variants and Hurst Exponents.

Figure 3.11 presents the maximum validation accuracy achieved by each of the twenty-eight optimizer variants, sorted in descending order of their mean performance. The data is stratified by the Hurst exponent ($H = 0.2$, $H = 0.5$, and $H = 0.7$) to illustrate the impact of noise correlation on the final model capacity.

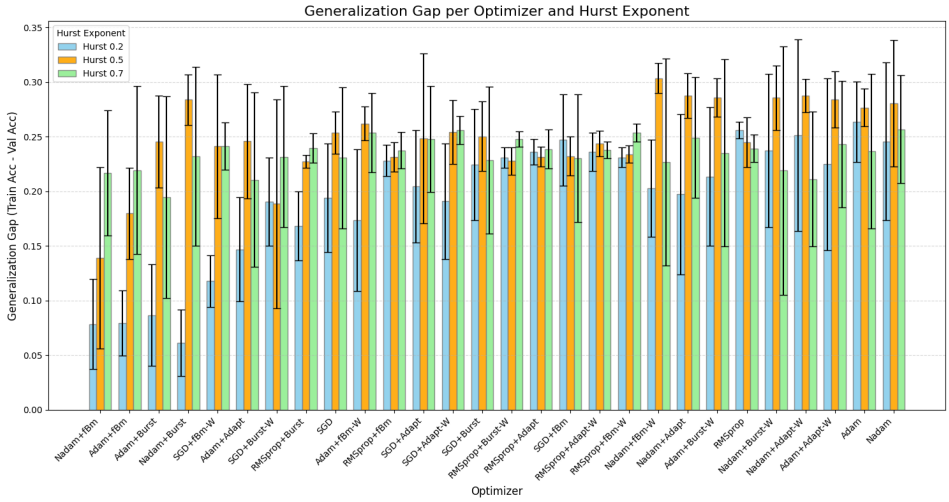


Figure 3.12: CIFAR-10 generalization gap across Optimizer Variants and Hurst Exponents.

Figure 3.12 details the generalization gap, defined as the difference between training accuracy and validation accuracy at the epoch of optimal performance. In this representation, lower values indicate better generalization, as they signify a closer alignment between the model's performance on learned data versus unseen data.

To complement the static performance metrics presented above, the subsequent analysis examines the temporal training dynamics of the most effective configurations identified for each H. By filtering for the optimal combination of maximum test accuracy and minimal generalization gap, three primary models emerged as the superior choices for their respective Hurst exponents. For the anti-persistent noise setting ($H = 0.2$), the `RMSprop+Burst-W` (Figure 3.13) variant proved dominant, achieving a peak accuracy of 74.8% alongside a generalization gap of 23.1%. Under standard Brownian motion ($H = 0.5$), the `RMSprop+Burst` (Figure 3.14) configuration got the highest overall performance of the study, reaching 74.9% accuracy with the lowest observed gap of 22.7%. In the persistent noise ($H = 0.7$),

the `RMSprop+fBm` (Figure 3.15) method outperformed the burst variants, achieving 73.8% accuracy with a gap of 23.7%. The following figures visualize the losses and the optimal epoch for training, the rest of the graphs can be seen in the appendix.

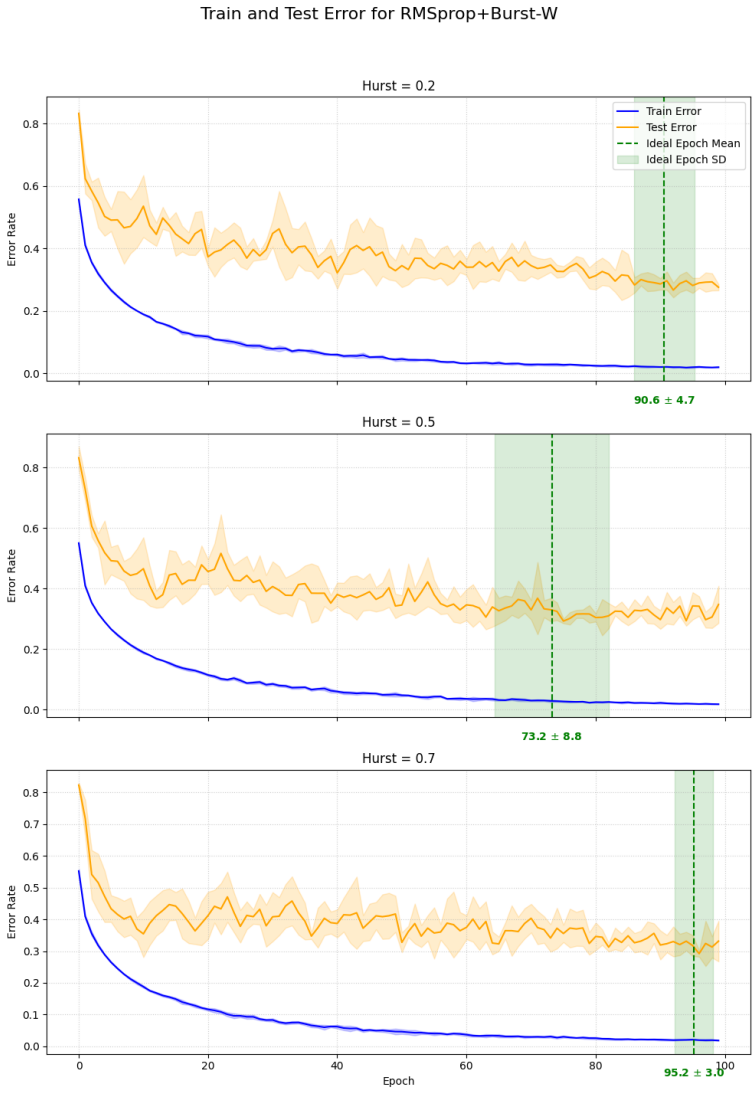


Figure 3.13: CIFAR-10 optimal H=0.2 RMSprop+burst-W

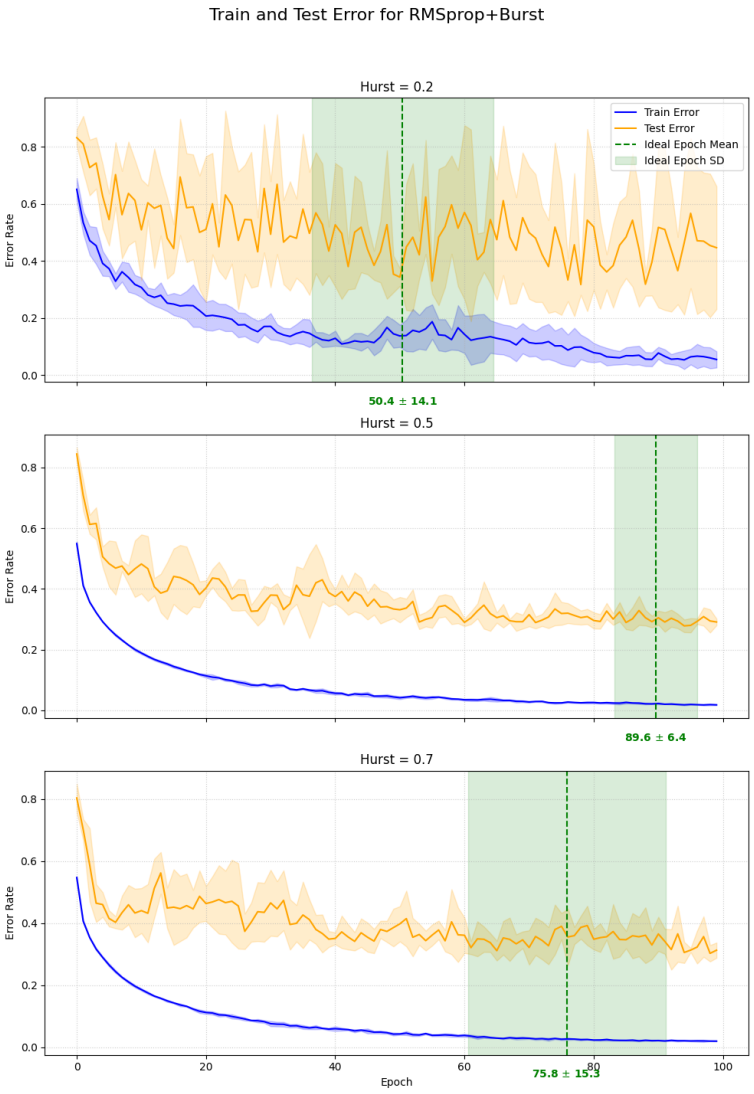


Figure 3.14: CIFAR-10 optimal H=0.5 RMSprop+burst

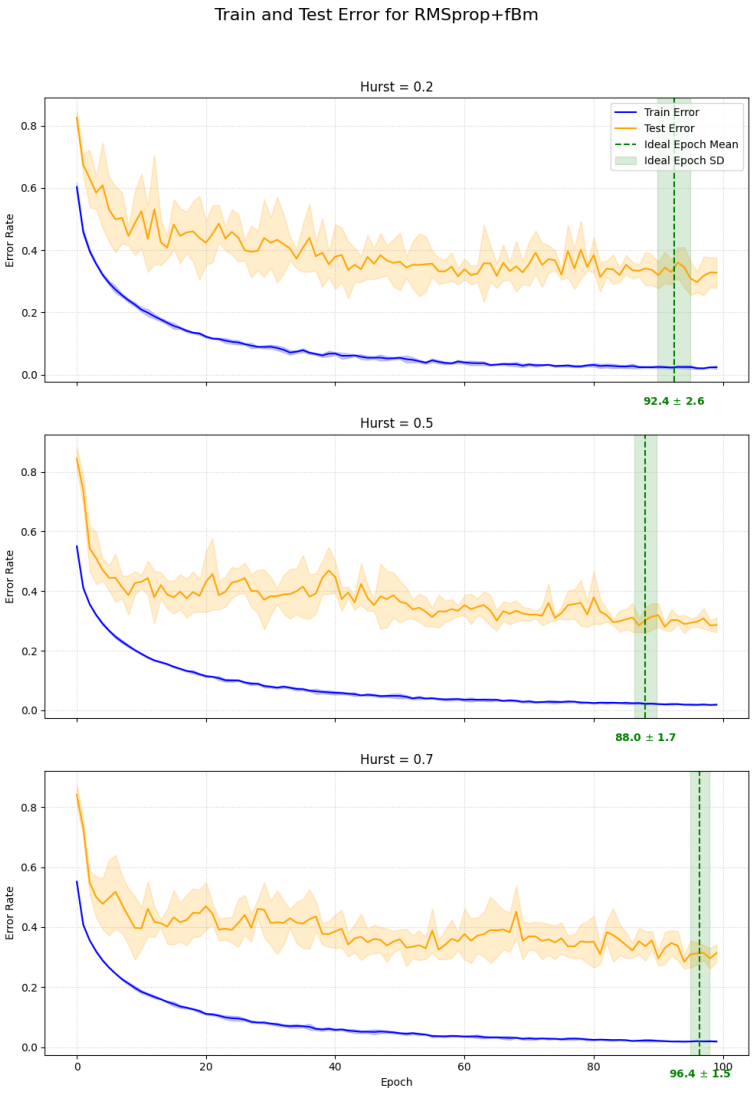


Figure 3.15: CIFAR-10 optimal H=0.7 RMSprop+fBm

3.3.2 Best performance

The top models are shown in the table 3.1. To identify the best optimization strategy across diverse datasets, a performance metric was developed, called as

Score. This score addresses the critical trade-off between predictive performance and computational efficiency by aggregating four equally weighted, normalized metrics: Test Accuracy, Test F1 Score, Generalization Gap, and Training Time. Normalization was applied within each dataset to map metrics to a [0,1] interval, utilizing inverted scaling for the Generalization Gap and Training Time to strictly penalize model instability and high computational cost.

Dataset	Best Variant	Score	Accuracy	F1 Score	Gap (Abs)	Time (s)
CIC-IOT	Nadam Adaptive-fBm (H=0.5)	0.76	93.55% (+11.5%)	0.6353 (+11.2%)	0.0014 (-29.2%)	62.9s (+5.3%)
CIFAR*	SGD fBm-W (H=0.2)	0.81	71.74% (+8.3%)	N/A	0.1148 (-56.3%)	1288s (-0.5%)
MNIST	SGD Adaptive-fBm-W (H=0.5)	0.84	97.75% (+0.0%)	0.9773 (+0.0%)	0.0163 (-24.8%)	20.8s (-20.7%)
NSL	Nadam Adaptive-fBm (H=0.5)	0.92	98.71% (-0.4%)	0.9871 (-0.4%)	0.0001 (-95.9%)	55.6s (+4.8%)
bot-iot	SGD fBm-W(Burst) (H=0.2)	0.98	99.93% (+0.4%)	0.9658 (+19.2%)	0.0010 (-72.4%)	33.4s (-33.5%)

*CIFAR results based on best-epoch Validation Accuracy; others based on Test Accuracy.

Table 3.1: Best Variant per Dataset.

3.4 Analysis

3.4.1 Runtime versus Accuracy

The comparison of runtime and accuracy shows that fBm noise can influence both convergence speed and final performance. In vision datasets, particularly MNIST, adding noise to the momentum term often moves models toward Pareto-efficient trade-offs, achieving similar accuracy in less time. For example, configurations with $H = 0.7$ are among the fastest, indicating that persistent noise can shorten convergence while maintaining competitive accuracy.

On CIFAR-10, where the task is more complex, the runtime vs accuracy curve is flatter, yet the introduction of intermittent "burst" noise reveals a distinct advantage. While adaptive methods at H=0.2 provided the fastest execution (roughly 61s), they often sacrificed model capacity. Conversely, the highest peak accuracies were achieved by RMSprop variants utilizing burst noise (e.g., RMSprop+Burst-W achieving 74.8% at H=0.2). This suggests that for high-dimensional vision

problems, continuous noise favors speed, whereas the alternating exploration-and-refinement phases of burst noise are essential for reaching deeper, more generalizable minima.

In traffic datasets, the results highlight a distinction between stabilization and acceleration. In CIC-IoT, the results indicate that the optimization landscape possesses a roughness that standard methods struggle to optimize efficiently. Standard Adam and Nadam configurations typically plateaued around 85-86% test accuracy. However, the introduction of noise at H=0.5 (standard Brownian motion) got significant performance gains. Specifically, the Adam+Adaptive-fBm (H=0.5) configuration achieved a test accuracy of 93.4%, outperforming its standard counterpart by nearly 9% while maintaining a comparable runtime (60.6s vs 63.4s). This suggests that for CIC-IoT, the random walk properties of fBm at H=0.5 allow the model to locate deeper solutions that standard momentum. Notably, faster but more volatile configurations (like SGD at H=0.2) suffered in performance, dropping to approximately 80% accuracy, further confirming that this dataset requires a balanced noise schedule rather than aggressive descent.

Next, the NSL-KDD dataset represents a "high-saturation" environment where the decision boundaries are well-defined, and most top configurations already exceed 99% accuracy. In this context, the addition of correlated noise transitions from an explorer to a stabilizer. Since the global minimum is easily accessible, fBm noise prevents the optimizer from oscillating around the solution, thereby reducing the variance in the final epochs. While it does not yield the dramatic accuracy jumps seen in CIC-IoT, it tightens the convergence loop. This contrast highlights a critical relationship between the Hurst exponent and dataset complexity: rough, complex landscapes (CIC-IoT) require the exploration of Brownian motion (H=0.5), while smooth, saturated landscapes (NSL-KDD) utilize noise primarily for stability.

Finally, the Bot-IoT dataset demonstrates the most dramatic trade-offs, thanks largely to the dataset's structure. Most configurations achieved near-perfect classi-

fication performance reaching 1.0 test accuracy, it can be related by the class imbalance where distinct attack labels (DoS, DDoS) overwhelmingly outnumber normal traffic. Within this high-saturation environment, noise functioned as a mechanism for speed rather than a regularizer: anti-persistent noise (H=0.2) combined with burst injection acted as a potent accelerator, achieving the fastest recorded training times (approx. 33.38s). In contrast, shifting to standard (H=0.5) or persistent (H=0.7) motion with burst noise prioritized precision, delivering perfect stability. This indicates that in network traffic tasks, burst noise can be tuned via the Hurst exponent to function either as an accelerator for convergence speed or a stabilizer for exactness, even when the decision boundary is easily accessible.

3.4.2 Impact of the Hurst Exponent

The Hurst exponent determines the temporal correlation of the injected noise, effectively dictating the "texture" of the optimization path: H=0.2 creates anti-persistent, high-frequency volatility; H=0.5 generates uncorrelated Brownian motion; and H=0.7 produces persistent, long-memory trajectories. The experiments confirm that this parameter alters not only the optimizer final accuracy but also the temporal dynamics of convergence and the stability of the solution.

In vision datasets, the impact of H is closely tied to the complexity of the feature space. For MNIST, where the loss landscape is relatively smooth, anti-persistent noise (H=0.2) aggressively reduces the generalization gap without affecting convergence, leading to a balanced training-validation performance. However, on the more complex CIFAR-10 dataset, this same aggressiveness creates a trade-off. While H=0.2 successfully narrows the gap, it often does so by suppressing model capacity, lowering both training and validation accuracy. Furthermore, the training curves reveal that anti-persistent noise delays convergence; models with H=0.2 typically required more epochs (peaking around epoch 98) to settle into their optimal state compared to the standard Brownian motion (H=0.5), which

converged earlier (around epoch 73). This indicates that while anti-persistence prevents overfitting, its volatility can impede the fine-grained feature extraction required for high-dimensional vision tasks.

In contrast, Brownian noise (H=0.5) consistently functions as a "safe option" across vision tasks. It reduces generalization gaps while avoiding the capacity drop observed with lower Hurst exponents, effectively balancing exploration with the retention of learned features. Persistent noise (H=0.7) showed the smallest impact on gap reduction in vision tasks, suggesting that its smoothing effect may inhibit the optimizer from escaping sharp local minima in deep convolutional networks.

For traffic datasets, the role of the Hurst exponent shifts based on the saturation of the dataset. In CIC-IoT, which presents a rough optimization landscape, standard Brownian motion (H=0.5) proved critical for performance. Here, the uncorrelated steps of H=0.5 allowed adaptive optimizers (like Nadam) to explore diverse regions of the parameter space, achieving some of the highest test accuracies while keeping the generalization gap minimal. This suggests that in non-saturated, rough landscapes, uncorrelated noise supports necessary exploration without the destabilizing effects of anti-persistence.

Finally, the Bot-IoT and NSL-KDD datasets highlight a distinct behavior in high-saturation environments. In these tasks, where baseline accuracy is already near perfect, the Hurst exponent acts as a control between speed and precision. In Bot-IoT, anti-persistent noise (H=0.2) combined with burst injection functioned as an accelerator, taking the model to convergence in record time (approx. 33s). On the other hand, persistent noise (H=0.7) acted as a stabilizer; while slightly slower, it smoothed the final descent, allowing the model to lock into a perfect classification state (1.0 accuracy) with zero variance. Thus, in network traffic analysis, H allows the practitioner to tune the optimizer for either maximum speed (H=0.2) or maximum reliability (H=0.7).

3.4.3 Effect of Noise Injection Strategy

The location of noise injection (in the velocity term or to the weights) matters as much as its temporal correlation. The analysis indicates that injecting fBm into the momentum term is generally the most reliable method for reducing overfitting without compromising model capacity. By adding noise to the velocity vector, the optimizer accumulates these perturbations over time. This creates a smoothed, persistent force that guides the descent, rather than a series of disjointed kicks. This dynamic proved particularly effective in complex landscapes; for instance, on CIFAR-10 and NSL-KDD, momentum-noise configurations (such as RMSprop+Burst and Nadam+Adaptive-fBm) consistently populated the Pareto frontier, balancing low generalization gaps with stable convergence.

In contrast, weight-noise injection operates as a direct perturbation of the parameters, introducing a zig-zag to the optimization path. While this can sometimes destabilize training in deep vision networks, it proves improvement in landscapes characterized by sharp, easily accessible minima. This is most evident in the Bot-IoT and MNIST datasets, where weight-noise variantsspecifically SGD+fBm-W combined with burst, achieved the fastest training times and lowest gaps. In these saturated environments, the direct perturbations of weight noise allow the model to escape shallow minima more rapidly than the smoothed updates of momentum noise. Consequently, while momentum injection offers a robust default for stability and consistency across varying domains, weight injection emerges as a specialized accelerator for high-saturation tasks.

3.4.4 Stability Across Runs

The standard deviations of test accuracy and gap metrics reveal that stability is closely tied to the Hurst setting and the noise injection schedule (Continuous vs. Burst). On vision datasets, Brownian H=0.5 and persistent H=0.7 noise

configurations exhibit lower variance, indicating that moderate-to-strong temporal correlation helps maintain consistent outcomes across runs compared to the erratic behavior of anti-persistent noise.

In network traffic datasets, specifically CIC-IoT and Bot-IoT, variance remains low for nearly all configurations, reflecting the highly structured nature of the data and the sufficient capacity of the models. Notably, the intermittent Burst noise strategy demonstrates a unique stability profile. By confining stochastic exploration to specific intervals, Burst configurationsparticularly with Brownian motion H=0.5, allow the optimizer to alternate between high-energy escape phases and deterministic refinement. This results in models that often match or exceed the stability of the baseline while securing higher test accuracy, making these hybrid combinations strong candidates for the Pareto front of practical deployment.

3.4.5 Key Insights

From the combined results, five distinct patterns emerge:

1. **Momentum Injection Reliability:** Injecting fBm into the momentum (velocity) term proves to be the most robust regularization method across datasets. Unlike weight injection, which can destabilize training in deep vision networks, momentum noise add perturbations smoothly, providing consistent generalization benefits without compromising model capacity.

2. **Anti-Persistence for Gap Minimization:** Anti-persistent noise ($H = 0.2$) is the most aggressive strategy for minimizing generalization gaps. However, in complex vision tasks like CIFAR-10, this aggression can suppress the model's ability to learn the features, resulting in lower overall accuracy unless paired with adaptive optimizers.

3. **Brownian Motion as a Safe Default:** Brownian noise ($H = 0.5$) offers the optimal balance between gap control, stability, and accuracy retention.

In rough optimization landscapes like CIC-IoT, it enables exploration that significantly outperforms standard methods, making it a safe and effective default configuration.

4. **Persistence for Efficiency:** Persistent noise ($H = 0.7$) provides moderate regularization while often accelerating convergence. In saturated environments, it acts as a stabilizer, allowing models to lock into high-precision solutions with reduced training time, making it suitable for time-constrained scenarios.

5. **The Accelerator Effect of Burst Noise:** In high-saturation environments such as Bot-IoT, intermittent "Burst" noise functions as a potent accelerator. specifically, anti-persistent bursts ($H = 0.2$) allowed models to converge to perfect accuracy faster than any continuous noise counterpart, demonstrating that structured intermittent perturbations can expedite training in easily navigable landscapes.

These findings align with the project's objective: to reduce overfitting without adding computational cost. In practice, the recommended settings are Brownian noise ($H = 0.5$) in the momentum velocity for a balanced outcome, or anti-persistent Burst noise ($H = 0.2$) when maximizing convergence speed in high-throughput environments.

Chapter 4

Conclusion

This work evaluated whether structured noise, specifically fixed and intermittent fBm perturbations, can improve neural network generalization by injecting memory into popular optimizers. SGD, RMSProp, Adam, and Nadam were extended to accept fBm noise in the momentum term and on weights. Experiments on five classification tasks (MNIST, CIFAR-10, CIC-IoT, Bot-IoT, NSL-KDD) were repeated five times per configuration and assessed by accuracy, cross-entropy, F1, ROC-AUC, and the generalization gap.

- **Noise placement matters.** Noise on the momentum term narrowed the generalization gap more effectively than weight-level noise without significant runtime added.
- **Noise type and schedule matter.** Anti-persistent noise ($H = 0.2$) minimized the gap most aggressively but risked suppressing learning in complex tasks. However, when applied as an intermittent Burst, it acted as a powerful accelerator in high-throughput environments. Brownian noise ($H = 0.5$) struck the most reliable balance between gap reduction and accuracy retention.
- **Optimizer choice matters.** On vision tasks, Adam with continuous noise performed best, while for network traffic analysis (CIC-IoT, Bot-IoT), Nadam with fixed momentum noise yielded the highest precision.

While distinct strategies excel in specific domains, the most consistent 'safe de-

fault' across all tasks proved to be Adam or Nadam with fixed fBm noise at $H = 0.5$. This configuration offers the most reliable balance between stability and generalization, effectively mitigating overfitting (e.g., Gap Acc = 0.0014 on NSL-KDD) while maintaining high model capacity (e.g., 93.55% accuracy on CIC-IoT). By providing a moderate, uncorrelated stochastic push, it avoids the volatility of anti-persistent noise and the sluggishness of persistent noise, making it the ideal starting point for unknown optimization landscapes.However, maximizing performance requires tuning the noise strategy to the specific problem topology:

- For Complex, Trap-Heavy Domains (e.g., CIFAR-10): Burst Injection is superior to continuous noise. The alternating phases of exploration and refinement allow the optimizer to escape sharp local minima that typically trap continuous variants, effectively 'rescuing' model performance in deep non-convex landscapes.
- For Rough, High-Variance Landscapes (e.g., CIC-IoT): Continuous Brownian Motion (H=0.5) remains the optimal choice. Unlike burst strategies, which can stagnate between pulses, continuous noise provides the constant, smoothed guidance necessary to navigate consistently rugged loss surfaces."
- For High-Throughput, Saturated Landscapes (e.g., Bot-IoT): Anti-persistent Burst noise (H=0.2) is recommended. In these scenarios, the intermittent high-frequency 'kicks' act as a potent accelerator, driving the model to convergence significantly faster than continuous methods without compromising accuracy.

Future Work

Future work can explore:

1. Learnable Hurst exponents or curvature-adaptive schedules that adjust fBm

parameters on-the-fly.

2. Integration of memory-aware noise into large-scale architectures (for example, ResNets, Transformers) and domains such as speech or reinforcement learning.

3. Combination of fBm regularization with complementary methods (dropout, weight decay, data augmentation) in a unified framework.

Bibliography

[Ara23] Axel A. Araneda. Price modelling under generalized fractional brownian motion, 2023. → pages 19

[Bis95] Christopher M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 1995. → pages 13

[CCS+19] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124018, 2019. → pages 2

[CFG14] Tianqi Chen, Emily B. Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo, 2014. → pages 17

[CWS+20] Alexander Camuto, Matthew Willetts, Umut Simsekli, Stephen J Roberts, and Chris C Holmes. Explicit regularisation in gaussian noise injections. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 16603–16614. Curran Associates, Inc., 2020. → pages 14

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*.

MIT Press, 2016. http://www.deeplearningbook.org. → pages 9, 10, 13

[HL22] Martin Hairer and Xue-Mei Li. Generating diffusions with fractional brownian motion. *Communications in Mathematical Physics*, 396(1):91141, August 2022. → pages 19

[JGN+17] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M. Kakade, and Michael I. Jordan. How to escape saddle points efficiently. *CoRR*, abs/1703.00887, 2017. → pages 20

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. → pages 20

[LL20] Yinan Li and Fang Liu. *Adaptive Gaussian Noise Injection Regularization for Neural Networks*, page 176189. Springer International Publishing, 2020. → pages 13, 15

[NVL+15] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015. → pages 2, 16

[Pol64] Boris T. Polyak. *Some methods of speeding up the convergence of iteration methods*. USSR Computational Mathematics and Mathematical Physics, 1964. → pages 1, 7

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 1986. → pages 7

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neu-

ral networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. → pages xi, 12

[Smi15] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015. → pages 20

[VLC+22] Hippolyte Verdier, Franois Laurent, Alhassan Cass, Christian L. Vestergaard, and Jean-Baptiste Masson. Variational inference of fractional brownian motion with linear computational complexity. *Physical Review E*, 106(5), November 2022. → pages 19

[XYZS21a] Zeke Xie, Li Yuan, Zhanxing Zhu, and Masashi Sugiyama. Positive-negative momentum: Manipulating stochastic gradient noise to improve generalization. In *International Conference on Machine Learning*, pages 11448–11458. PMLR, 2021. → pages 2

[XYZS21b] Zeke Xie, Li Yuan, Zhanxing Zhu, and Masashi Sugiyama. Positive-negative momentum: Manipulating stochastic gradient noise to improve generalization. In *International Conference on Machine Learning*, pages 11448–11458. PMLR, 2021. → pages 2, 16

[ZLZ+19] Ruqi Zhang, Chunyuan Li, Jianyi Zhang, Changyou Chen, and Andrew Gordon Wilson. Cyclical stochastic gradient MCMC for bayesian deep learning. *CoRR*, abs/1902.03932, 2019. → pages 20

[Zur09] Richard M Zur. Noise injection for training artificial neural networks: A comparison with weight decay and early stopping. *Medical physics*, 2009. → pages 15

Appendix

Appendix A

Tables

Table A.1: Results for CIC-IOT dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	Adam (Adaptive-fBm)	0.8795 ± 0.0495	0.8827 ± 0.0472	0.6075 ± 0.0358	0.0017 ± 0.0010	0.4682 ± 0.2763	0.4468 ± 0.2672	58.4980 ± 2.7747
0.2	Adam (Adaptive-fBm-W)	0.8668 ± 0.0217	0.8722 ± 0.0210	0.6026 ± 0.0211	0.0036 ± 0.0007	0.2978 ± 0.0667	0.2812 ± 0.0643	62.9120 ± 1.2506
0.2	Adam (Standard)	0.8470 ± 0.0288	0.8521 ± 0.0284	0.5836 ± 0.0255	0.0027 ± 0.0014	0.4627 ± 0.1581	0.4439 ± 0.1560	60.7300 ± 2.0389
0.2	Adam (fBm)	0.8425 ± 0.0259	0.8464 ± 0.0256	0.5387 ± 0.0262	0.0009 ± 0.0014	0.4305 ± 0.1442	0.4185 ± 0.1424	63.7580 ± 4.6340
0.2	Adam (fBm(Burst))	0.8191 ± 0.0175	0.8220 ± 0.0174	0.4842 ± 0.0078	-0.0003 ± 0.0008	0.5835 ± 0.1392	0.5699 ± 0.1391	64.5440 ± 0.6764
0.2	Adam (fBm-W)	0.9009 ± 0.0462	0.9034 ± 0.0434	0.6192 ± 0.0161	0.0020 ± 0.0009	0.2792 ± 0.1507	0.2642 ± 0.1443	66.5980 ± 9.4671
0.2	Adam (fBm-W(Burst))	0.8626 ± 0.0243	0.8672 ± 0.0234	0.5904 ± 0.0180	0.0025 ± 0.0015	0.3291 ± 0.0615	0.3133 ± 0.0585	64.6120 ± 0.8833
0.2	Nadam (Adaptive-fBm)	0.9097 ± 0.0480	0.9127 ± 0.0475	0.6250 ± 0.0346	0.0016 ± 0.0008	0.3369 ± 0.3640	0.3218 ± 0.3606	62.6640 ± 10.8085
0.2	Nadam (Adaptive-fBm-W)	0.8694 ± 0.0417	0.8727 ± 0.0418	0.6053 ± 0.0429	0.0019 ± 0.0007	0.3531 ± 0.1237	0.3362 ± 0.1219	63.0360 ± 1.1372
0.2	Nadam (Standard)	0.8608 ± 0.0628	0.8657 ± 0.0609	0.5971 ± 0.0406	0.0012 ± 0.0010	0.4309 ± 0.2254	0.4117 ± 0.2193	59.6580 ± 1.3211
0.2	Nadam (fBm)	0.8284 ± 0.0109	0.8326 ± 0.0113	0.5407 ± 0.0182	0.0006 ± 0.0008	0.5817 ± 0.1332	0.5671 ± 0.1326	70.5460 ± 7.7298
0.2	Nadam (fBm(Burst))	0.7901 ± 0.0437	0.7932 ± 0.0453	0.4873 ± 0.0618	0.0001 ± 0.0006	0.7189 ± 0.1953	0.7037 ± 0.1972	67.7940 ± 8.0240
0.2	Nadam (fBm-W)	0.8781 ± 0.0539	0.8816 ± 0.0531	0.6002 ± 0.0366	0.0017 ± 0.0017	0.3624 ± 0.2176	0.3444 ± 0.2119	65.4680 ± 9.2463
0.2	Nadam (fBm-W(Burst))	0.8377 ± 0.0130	0.8417 ± 0.0130	0.5814 ± 0.0249	0.0023 ± 0.0017	0.4453 ± 0.1772	0.4286 ± 0.1744	65.4600 ± 3.0705
0.2	RMSprop (Adaptive-fBm)	0.8639 ± 0.0370	0.8672 ± 0.0361	0.5741 ± 0.0172	0.0009 ± 0.0016	0.3402 ± 0.0973	0.3282 ± 0.0961	57.2820 ± 1.0026
0.2	RMSprop (Adaptive-fBm-W)	0.8465 ± 0.0260	0.8509 ± 0.0265	0.5793 ± 0.0157	0.0020 ± 0.0007	0.3951 ± 0.0840	0.3825 ± 0.0842	56.7920 ± 0.9658
0.2	RMSprop (Standard)	0.8320 ± 0.0108	0.8371 ± 0.0109	0.5494 ± 0.0112	0.0020 ± 0.0010	0.4890 ± 0.0979	0.4691 ± 0.0949	56.5560 ± 3.5017
0.2	RMSprop (fBm)	0.8625 ± 0.0105	0.8659 ± 0.0092	0.5822 ± 0.0172	0.0015 ± 0.0008	0.3232 ± 0.0149	0.3121 ± 0.0152	58.0480 ± 1.4642
0.2	RMSprop (fBm(Burst))	0.8358 ± 0.0183	0.8386 ± 0.0190	0.5381 ± 0.0120	0.0004 ± 0.0007	0.4226 ± 0.0692	0.4096 ± 0.0653	64.0580 ± 10.0010
0.2	RMSprop (fBm-W)	0.8379 ± 0.0141	0.8412 ± 0.0146	0.5503 ± 0.0134	0.0013 ± 0.0012	0.4562 ± 0.0932	0.4415 ± 0.0908	56.5060 ± 0.9071
0.2	RMSprop (fBm-W(Burst))	0.8528 ± 0.0326	0.8551 ± 0.0302	0.5784 ± 0.0392	0.0003 ± 0.0012	0.4250 ± 0.0704	0.4134 ± 0.0688	60.0040 ± 0.4219
0.2	SGD (Adaptive-fBm)	0.7962 ± 0.0262	0.7983 ± 0.0266	0.5378 ± 0.0154	0.0002 ± 0.0008	0.4132 ± 0.0168	0.4017 ± 0.0153	54.9420 ± 0.8106
0.2	SGD (Adaptive-fBm-W)	0.8157 ± 0.0185	0.8195 ± 0.0200	0.5471 ± 0.0080	0.0012 ± 0.0017	0.4183 ± 0.0149	0.4067 ± 0.0144	61.0360 ± 12.2770
0.2	SGD (Standard)	0.8056 ± 0.0151	0.8081 ± 0.0150	0.5538 ± 0.0098	0.0011 ± 0.0008	0.4055 ± 0.0032	0.3954 ± 0.0028	64.6180 ± 7.3167
0.2	SGD (fBm)	0.8129 ± 0.0092	0.8149 ± 0.0084	0.5420 ± 0.0210	0.0017 ± 0.0010	0.4050 ± 0.0100	0.3951 ± 0.0097	60.7060 ± 11.9389

Continued on next page

Table A.1: Results for CIC-IOT dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	SGD (fBm(Burst))	0.8086 ± 0.0174	0.8119 ± 0.0187	0.5514 ± 0.0089	0.0013 ± 0.0021	0.4126 ± 0.0099	0.4011 ± 0.0116	57.3920 ± 1.2448
0.2	SGD (fBm-W)	0.8213 ± 0.0114	0.8225 ± 0.0127	0.5416 ± 0.0149	0.0007 ± 0.0009	0.4007 ± 0.0061	0.3920 ± 0.0070	55.3740 ± 1.5624
0.2	SGD (fBm-W(Burst))	0.8133 ± 0.0143	0.8145 ± 0.0150	0.5537 ± 0.0088	0.0002 ± 0.0008	0.4037 ± 0.0092	0.3945 ± 0.0102	57.6180 ± 2.1401
0.5	Adam (Adaptive-fBm)	0.9336 ± 0.0509	0.9374 ± 0.0495	0.6407 ± 0.0275	0.0021 ± 0.0012	0.1758 ± 0.1093	0.1628 ± 0.1060	60.6520 ± 3.0807
0.5	Adam (Adaptive-fBm-W)	0.8517 ± 0.0495	0.8569 ± 0.0483	0.5989 ± 0.0287	0.0010 ± 0.0004	0.4659 ± 0.1483	0.4453 ± 0.1440	63.5780 ± 1.1080
0.5	Adam (Standard)	0.8472 ± 0.0235	0.8509 ± 0.0238	0.5777 ± 0.0239	0.0017 ± 0.0017	0.3628 ± 0.0783	0.3471 ± 0.0787	63.3960 ± 4.7212
0.5	Adam (fBm)	0.8490 ± 0.0194	0.8536 ± 0.0210	0.5558 ± 0.0213	0.0007 ± 0.0020	0.3863 ± 0.1358	0.3745 ± 0.1351	66.5740 ± 8.9677
0.5	Adam (fBm(Burst))	0.8649 ± 0.0224	0.8683 ± 0.0222	0.5876 ± 0.0180	0.0028 ± 0.0010	0.3296 ± 0.0787	0.3149 ± 0.0758	60.4020 ± 0.0815
0.5	Adam (fBm-W)	0.8457 ± 0.0095	0.8504 ± 0.0099	0.5812 ± 0.0062	0.0024 ± 0.0009	0.3756 ± 0.0247	0.3576 ± 0.0254	61.8580 ± 1.8023
0.5	Adam (fBm-W(Burst))	0.8550 ± 0.0308	0.8594 ± 0.0301	0.5763 ± 0.0312	0.0031 ± 0.0004	0.3853 ± 0.1125	0.3685 ± 0.1088	60.0280 ± 1.0497
0.5	Nadam (Adaptive-fBm)	0.9355 ± 0.0202	0.9377 ± 0.0188	0.6353 ± 0.0101	0.0014 ± 0.0018	0.1662 ± 0.0382	0.1525 ± 0.0366	62.9320 ± 5.6526
0.5	Nadam (Adaptive-fBm-W)	0.8188 ± 0.0108	0.8241 ± 0.0110	0.5734 ± 0.0113	0.0014 ± 0.0008	0.6682 ± 0.0409	0.6489 ± 0.0405	63.7780 ± 1.4009
0.5	Nadam (Standard)	0.8502 ± 0.0422	0.8519 ± 0.0439	0.5898 ± 0.0346	0.0022 ± 0.0014	0.3619 ± 0.0892	0.3487 ± 0.0907	59.7480 ± 2.8379
0.5	Nadam (fBm)	0.8544 ± 0.0257	0.8587 ± 0.0261	0.5791 ± 0.0293	0.0027 ± 0.0015	0.3708 ± 0.1325	0.3587 ± 0.1315	65.6540 ± 1.1032
0.5	Nadam (fBm(Burst))	0.8453 ± 0.0325	0.8490 ± 0.0327	0.5688 ± 0.0333	0.0021 ± 0.0027	0.4169 ± 0.1679	0.4048 ± 0.1672	61.3160 ± 1.7332
0.5	Nadam (fBm-W)	0.8515 ± 0.0233	0.8566 ± 0.0235	0.5835 ± 0.0168	0.0033 ± 0.0007	0.4071 ± 0.0721	0.3880 ± 0.0693	63.4920 ± 3.6075
0.5	Nadam (fBm-W(Burst))	0.8247 ± 0.0249	0.8294 ± 0.0247	0.5802 ± 0.0259	0.0016 ± 0.0009	0.5084 ± 0.1049	0.4915 ± 0.1038	68.3720 ± 12.3845
0.5	RMSprop (Adaptive-fBm)	0.8410 ± 0.0136	0.8443 ± 0.0139	0.5674 ± 0.0243	0.0013 ± 0.0007	0.3839 ± 0.0292	0.3710 ± 0.0300	57.0020 ± 0.8484
0.5	RMSprop (Adaptive-fBm-W)	0.8715 ± 0.0184	0.8743 ± 0.0178	0.5904 ± 0.0200	0.0012 ± 0.0011	0.3308 ± 0.0694	0.3194 ± 0.0692	58.5580 ± 2.7955
0.5	RMSprop (Standard)	0.8252 ± 0.0074	0.8296 ± 0.0068	0.5439 ± 0.0078	0.0009 ± 0.0005	0.5482 ± 0.0281	0.5260 ± 0.0257	58.2060 ± 1.2883
0.5	RMSprop (fBm)	0.8834 ± 0.0454	0.8851 ± 0.0436	0.5807 ± 0.0216	0.0014 ± 0.0018	0.3009 ± 0.0840	0.2902 ± 0.0815	62.8240 ± 10.7104
0.5	RMSprop (fBm(Burst))	0.8340 ± 0.0241	0.8381 ± 0.0240	0.5513 ± 0.0422	0.0020 ± 0.0013	0.4419 ± 0.0904	0.4264 ± 0.0873	61.5180 ± 11.4446
0.5	RMSprop (fBm-W)	0.8425 ± 0.0471	0.8455 ± 0.0460	0.5674 ± 0.0306	0.0009 ± 0.0010	0.4376 ± 0.1302	0.4247 ± 0.1276	67.7040 ± 13.0080
0.5	RMSprop (fBm-W(Burst))	0.8307 ± 0.0376	0.8347 ± 0.0381	0.5651 ± 0.0263	0.0011 ± 0.0017	0.4625 ± 0.1270	0.4488 ± 0.1265	56.5420 ± 0.8403
0.5	SGD (Adaptive-fBm)	0.7811 ± 0.0050	0.7828 ± 0.0045	0.5368 ± 0.0132	0.0006 ± 0.0012	0.4259 ± 0.0102	0.4147 ± 0.0096	62.9340 ± 11.4026

Continued on next page

Table A.1: Results for CIC-IOT dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.5	SGD (Adaptive-fBm-W)	0.8063 ± 0.0246	0.8092 ± 0.0256	0.5465 ± 0.0136	0.0018 ± 0.0014	0.4114 ± 0.0143	0.3997 ± 0.0131	61.0200 ± 11.7144
0.5	SGD (Standard)	0.8152 ± 0.0053	0.8177 ± 0.0057	0.5448 ± 0.0122	0.0019 ± 0.0012	0.4040 ± 0.0074	0.3923 ± 0.0056	59.0740 ± 5.2160
0.5	SGD (fBm)	0.8202 ± 0.0121	0.8250 ± 0.0125	0.5432 ± 0.0113	0.0012 ± 0.0007	0.4050 ± 0.0082	0.3938 ± 0.0079	55.5780 ± 0.9022
0.5	SGD (fBm(Burst))	0.8074 ± 0.0133	0.8097 ± 0.0151	0.5487 ± 0.0089	0.0011 ± 0.0011	0.4118 ± 0.0113	0.4015 ± 0.0132	55.2120 ± 1.6294
0.5	SGD (fBm-W)	0.8136 ± 0.0240	0.8157 ± 0.0242	0.5604 ± 0.0129	0.0007 ± 0.0007	0.4011 ± 0.0139	0.3894 ± 0.0138	65.9100 ± 14.6371
0.5	SGD (fBm-W(Burst))	0.8251 ± 0.0053	0.8279 ± 0.0058	0.5526 ± 0.0112	0.0012 ± 0.0014	0.3968 ± 0.0057	0.3866 ± 0.0059	59.6640 ± 12.4678
0.7	Adam (Adaptive-fBm)	0.8571 ± 0.0337	0.8619 ± 0.0339	0.5911 ± 0.0151	0.0029 ± 0.0009	0.4585 ± 0.2185	0.4366 ± 0.2141	62.9420 ± 10.8925
0.7	Adam (Adaptive-fBm-W)	0.8584 ± 0.0441	0.8634 ± 0.0428	0.5887 ± 0.0331	0.0023 ± 0.0007	0.3521 ± 0.1588	0.3357 ± 0.1555	62.9680 ± 0.0137
0.7	Adam (Standard)	0.8523 ± 0.0140	0.8583 ± 0.0150	0.5983 ± 0.0112	0.0029 ± 0.0012	0.3776 ± 0.0438	0.3585 ± 0.0437	58.9620 ± 1.9017
0.7	Adam (fBm)	0.8388 ± 0.0228	0.8431 ± 0.0228	0.5562 ± 0.0237	0.0020 ± 0.0013	0.4203 ± 0.1253	0.4040 ± 0.1232	64.8740 ± 1.5500
0.7	Adam (fBm(Burst))	0.8507 ± 0.0255	0.8560 ± 0.0264	0.5895 ± 0.0163	0.0022 ± 0.0012	0.4518 ± 0.2150	0.4305 ± 0.2093	59.8860 ± 1.9090
0.7	Adam (fBm-W)	0.8575 ± 0.0186	0.8623 ± 0.0178	0.5921 ± 0.0231	0.0022 ± 0.0017	0.3399 ± 0.0838	0.3244 ± 0.0838	61.7200 ± 0.0107
0.7	Adam (fBm-W(Burst))	0.8678 ± 0.0218	0.8724 ± 0.0223	0.6022 ± 0.0154	0.0024 ± 0.0007	0.3100 ± 0.0360	0.2937 ± 0.0353	60.7440 ± 1.7565
0.7	Nadam (Adaptive-fBm)	0.9230 ± 0.0250	0.9259 ± 0.0225	0.6413 ± 0.0055	0.0019 ± 0.0013	0.1889 ± 0.0500	0.1759 ± 0.0469	62.3500 ± 0.9705
0.7	Nadam (Adaptive-fBm-W)	0.8678 ± 0.0442	0.8709 ± 0.0436	0.6038 ± 0.0318	0.0030 ± 0.0007	0.3516 ± 0.1600	0.3371 ± 0.1593	66.1420 ± 1.8259
0.7	Nadam (Standard)	0.8876 ± 0.0698	0.8913 ± 0.0677	0.6045 ± 0.0304	0.0030 ± 0.0016	0.3124 ± 0.1764	0.2970 ± 0.1738	59.4000 ± 1.2151
0.7	Nadam (fBm)	0.8576 ± 0.0434	0.8612 ± 0.0425	0.5864 ± 0.0340	0.0024 ± 0.0019	0.4175 ± 0.2413	0.4034 ± 0.2400	65.5100 ± 9.2142
0.7	Nadam (fBm(Burst))	0.8542 ± 0.0346	0.8584 ± 0.0341	0.5880 ± 0.0240	0.0019 ± 0.0012	0.4281 ± 0.1541	0.4119 ± 0.1511	69.6560 ± 7.3178
0.7	Nadam (fBm-W)	0.8688 ± 0.0605	0.8727 ± 0.0592	0.5990 ± 0.0333	0.0024 ± 0.0007	0.4071 ± 0.2184	0.3862 ± 0.2123	64.5180 ± 9.7384
0.7	Nadam (fBm-W(Burst))	0.8356 ± 0.0230	0.8409 ± 0.0223	0.5772 ± 0.0146	0.0017 ± 0.0013	0.5401 ± 0.1908	0.5184 ± 0.1910	61.0160 ± 2.2075
0.7	RMSprop (Adaptive-fBm)	0.8594 ± 0.0077	0.8631 ± 0.0081	0.5735 ± 0.0186	0.0013 ± 0.0010	0.3576 ± 0.0246	0.3469 ± 0.0260	55.7840 ± 0.5044
0.7	RMSprop (Adaptive-fBm-W)	0.8601 ± 0.0156	0.8631 ± 0.0149	0.5822 ± 0.0193	0.0025 ± 0.0005	0.3585 ± 0.0457	0.3464 ± 0.0433	56.5060 ± 1.0093
0.7	RMSprop (Standard)	0.8354 ± 0.0186	0.8406 ± 0.0200	0.5610 ± 0.0225	0.0019 ± 0.0016	0.4680 ± 0.1098	0.4481 ± 0.1060	57.4520 ± 2.0833
0.7	RMSprop (fBm)	0.8447 ± 0.0332	0.8490 ± 0.0332	0.5664 ± 0.0309	0.0010 ± 0.0013	0.4039 ± 0.1006	0.3898 ± 0.0991	56.3700 ± 0.8732
0.7	RMSprop (fBm(Burst))	0.8431 ± 0.0118	0.8471 ± 0.0107	0.5589 ± 0.0170	0.0028 ± 0.0013	0.3679 ± 0.0487	0.3561 ± 0.0484	61.6300 ± 11.4822

Continued on next page

Table A.1: Results for CIC-IOT dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.7	RMSprop (fBm-W)	0.8582 ± 0.0308	0.8621 ± 0.0304	0.5743 ± 0.0221	0.0008 ± 0.0009	0.3638 ± 0.0445	0.3514 ± 0.0427	57.3020 ± 2.5221
0.7	RMSprop (fBm-W(Burst))	0.8393 ± 0.0273	0.8435 ± 0.0277	0.5747 ± 0.0247	0.0007 ± 0.0008	0.4018 ± 0.0426	0.3877 ± 0.0425	57.1540 ± 1.3633
0.7	SGD (Adaptive-fBm)	0.8079 ± 0.0191	0.8110 ± 0.0187	0.5324 ± 0.0075	0.0018 ± 0.0004	0.4101 ± 0.0115	0.3986 ± 0.0097	55.1700 ± 2.9699
0.7	SGD (Adaptive-fBm-W)	0.8193 ± 0.0316	0.8219 ± 0.0344	0.5639 ± 0.0109	0.0004 ± 0.0007	0.4060 ± 0.0065	0.3946 ± 0.0061	54.1740 ± 0.3807
0.7	SGD (Standard)	0.8139 ± 0.0199	0.8154 ± 0.0195	0.5532 ± 0.0112	0.0015 ± 0.0015	0.4015 ± 0.0081	0.3911 ± 0.0085	59.3020 ± 5.7160
0.7	SGD (fBm)	0.8163 ± 0.0229	0.8189 ± 0.0229	0.5527 ± 0.0188	0.0008 ± 0.0013	0.4046 ± 0.0092	0.3950 ± 0.0083	65.1880 ± 15.2852
0.7	SGD (fBm(Burst))	0.8139 ± 0.0170	0.8177 ± 0.0187	0.5503 ± 0.0127	0.0010 ± 0.0012	0.4092 ± 0.0042	0.3984 ± 0.0036	55.0320 ± 1.1568
0.7	SGD (fBm-W)	0.7907 ± 0.0129	0.7930 ± 0.0136	0.5288 ± 0.0167	0.0012 ± 0.0011	0.4161 ± 0.0133	0.4055 ± 0.0139	54.0460 ± 0.5062
0.7	SGD (fBm-W(Burst))	0.8101 ± 0.0186	0.8117 ± 0.0182	0.5588 ± 0.0108	0.0010 ± 0.0007	0.4063 ± 0.0111	0.3958 ± 0.0104	54.1740 ± 1.0529

Table A.2: Results for CIFAR dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	Adam (Adaptive-fBm)	0.1784 ± 0.0516	0.4953 ± 0.1332	0.1188 ± 0.0625	0.0116 ± 0.0046	12.6811 ± 1.0551	1.6415 ± 0.7000	62.3780 ± 1.2854
0.2	Adam (Adaptive-fBm-W)	0.2374 ± 0.0410	0.8142 ± 0.0578	0.2006 ± 0.0457	0.1104 ± 0.0237	12.1021 ± 0.6613	0.5568 ± 0.2405	70.1500 ± 11.0186
0.2	Adam (Standard)	0.3080 ± 0.0373	0.8259 ± 0.0591	0.2798 ± 0.0422	0.1166 ± 0.0204	10.9628 ± 0.5998	0.5140 ± 0.2156	71.8120 ± 4.6958
0.2	Adam (fBm)	0.1467 ± 0.0255	0.4605 ± 0.1310	0.0801 ± 0.0244	0.0100 ± 0.0095	13.5290 ± 0.4546	2.4781 ± 1.9154	65.9020 ± 9.1293
0.2	Adam (fBm(Burst))	0.1593 ± 0.0479	0.6465 ± 0.0868	0.0871 ± 0.0604	0.0285 ± 0.0173	13.2727 ± 0.9634	1.1299 ± 0.4662	79.8000 ± 1.3998
0.2	Adam (fBm-W)	0.2446 ± 0.0537	0.6365 ± 0.0725	0.1909 ± 0.0605	0.0512 ± 0.0170	11.9910 ± 0.8559	1.5520 ± 0.6784	66.1700 ± 9.0654
0.2	Adam (fBm-W(Burst))	0.2633 ± 0.0483	0.7862 ± 0.0728	0.2247 ± 0.0559	0.1046 ± 0.0129	11.6709 ± 0.7927	0.6788 ± 0.2842	78.2220 ± 0.6748
0.2	Nadam (Adaptive-fBm)	0.1265 ± 0.0445	0.3401 ± 0.1394	0.0639 ± 0.0472	0.0107 ± 0.0067	13.7353 ± 0.8198	3.6490 ± 1.5507	66.3580 ± 8.9131
0.2	Nadam (Adaptive-fBm-W)	0.2247 ± 0.0820	0.7721 ± 0.0610	0.1832 ± 0.1098	0.1109 ± 0.0188	12.3207 ± 1.3281	0.7967 ± 0.4614	62.7660 ± 1.0996
0.2	Nadam (Standard)	0.2647 ± 0.0345	0.8330 ± 0.0535	0.2190 ± 0.0472	0.1264 ± 0.0247	11.6794 ± 0.5507	0.4787 ± 0.1874	72.1240 ± 2.1144
0.2	Nadam (fBm)	0.1191 ± 0.0258	0.3886 ± 0.1637	0.0498 ± 0.0246	0.0086 ± 0.0076	13.9920 ± 0.4170	2.0545 ± 0.9135	65.9120 ± 9.1170
0.2	Nadam (fBm(Burst))	0.2206 ± 0.0544	0.6252 ± 0.1321	0.1494 ± 0.0532	0.0476 ± 0.0364	12.3799 ± 0.8691	1.4016 ± 0.6462	78.6160 ± 2.0608
0.2	Nadam (fBm-W)	0.2205 ± 0.0814	0.6511 ± 0.2200	0.1816 ± 0.0881	0.0695 ± 0.0404	12.3770 ± 1.3263	2.2509 ± 3.0940	65.8960 ± 9.1477
0.2	Nadam (fBm-W(Burst))	0.3007 ± 0.0544	0.7478 ± 0.0429	0.2597 ± 0.0516	0.0902 ± 0.0160	11.0934 ± 0.8594	0.8296 ± 0.1885	78.2560 ± 0.8773
0.2	RMSprop (Adaptive-fBm)	0.2254 ± 0.0572	0.8880 ± 0.0404	0.1687 ± 0.0543	0.1580 ± 0.0190	12.3360 ± 0.9188	0.4293 ± 0.1893	74.2180 ± 10.9907
0.2	RMSprop (Adaptive-fBm-W)	0.1844 ± 0.0239	0.8850 ± 0.0436	0.1250 ± 0.0293	0.1613 ± 0.0199	12.9874 ± 0.3840	0.4291 ± 0.1988	66.2220 ± 8.9940
0.2	RMSprop (Standard)	0.2923 ± 0.0247	0.7249 ± 0.1306	0.2498 ± 0.0208	0.0807 ± 0.0287	11.2108 ± 0.4025	1.0659 ± 0.7823	68.6480 ± 2.4994
0.2	RMSprop (fBm)	0.1914 ± 0.0438	0.8547 ± 0.0215	0.1323 ± 0.0430	0.1436 ± 0.0159	12.8718 ± 0.6967	0.5255 ± 0.2272	61.8380 ± 0.5788
0.2	RMSprop (fBm(Burst))	0.1499 ± 0.0495	0.6380 ± 0.2801	0.0943 ± 0.0792	0.0678 ± 0.0444	11.1015 ± 4.9598	3.2671 ± 5.6175	77.8160 ± 1.0873
0.2	RMSprop (fBm-W)	0.2280 ± 0.0542	0.8987 ± 0.0250	0.1623 ± 0.0620	0.1650 ± 0.0087	12.2915 ± 0.8698	0.3442 ± 0.1224	61.9600 ± 0.9536
0.2	RMSprop (fBm-W(Burst))	0.2277 ± 0.0152	0.8490 ± 0.0598	0.1616 ± 0.0203	0.1443 ± 0.0273	12.3004 ± 0.2445	0.6916 ± 0.3976	79.4240 ± 1.7397
0.2	SGD (Adaptive-fBm)	0.2431 ± 0.0385	0.6984 ± 0.0755	0.1906 ± 0.0648	0.0683 ± 0.0158	12.0029 ± 0.6418	1.1321 ± 0.5598	61.4720 ± 0.6519
0.2	SGD (Adaptive-fBm-W)	0.2286 ± 0.0346	0.7474 ± 0.0637	0.1878 ± 0.0343	0.0868 ± 0.0170	12.2330 ± 0.5593	0.7528 ± 0.2432	61.5940 ± 0.8344
0.2	SGD (Standard)	0.2514 ± 0.0447	0.7666 ± 0.0298	0.1991 ± 0.0428	0.0835 ± 0.0069	11.8838 ± 0.7265	0.7030 ± 0.1252	66.8640 ± 3.6352
0.2	SGD (fBm)	0.2304 ± 0.0523	0.8165 ± 0.0542	0.1891 ± 0.0624	0.0975 ± 0.0166	12.2078 ± 0.8441	0.5433 ± 0.2048	65.4680 ± 9.4207

Continued on next page

Table A.2: Results for CIFAR dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	SGD (fBm(Burst))	0.2579 ± 0.0374	0.7129 ± 0.0931	0.2150 ± 0.0424	0.0777 ± 0.0156	11.7684 ± 0.6052	1.0098 ± 0.5514	78.3840 ± 0.8148
0.2	SGD (fBm-W)	0.2469 ± 0.0264	0.7173 ± 0.0681	0.1795 ± 0.0481	0.0522 ± 0.0138	11.9297 ± 0.4347	0.9370 ± 0.4632	65.7700 ± 9.1928
0.2	SGD (fBm-W(Burst))	0.2802 ± 0.0538	0.7534 ± 0.0939	0.2453 ± 0.0739	0.0718 ± 0.0249	11.3643 ± 0.8347	0.7989 ± 0.4119	78.2100 ± 1.3931
0.5	Adam (Adaptive-fBm)	0.1873 ± 0.0285	0.6392 ± 0.0532	0.1167 ± 0.0395	0.0238 ± 0.0100	12.8879 ± 0.4660	1.0170 ± 0.1518	62.8180 ± 0.5879
0.5	Adam (Adaptive-fBm-W)	0.2635 ± 0.0470	0.7771 ± 0.0558	0.2214 ± 0.0619	0.1022 ± 0.0172	11.6863 ± 0.7517	0.7314 ± 0.2588	63.1360 ± 1.0940
0.5	Adam (Standard)	0.3264 ± 0.0247	0.8405 ± 0.0165	0.3153 ± 0.0320	0.1167 ± 0.0069	10.6668 ± 0.3985	0.4525 ± 0.0522	73.8180 ± 2.6770
0.5	Adam (fBm)	0.1686 ± 0.0313	0.5927 ± 0.0404	0.1116 ± 0.0381	0.0267 ± 0.0060	13.1872 ± 0.5261	1.2304 ± 0.1550	62.8100 ± 0.3786
0.5	Adam (fBm(Burst))	0.2239 ± 0.0525	0.7536 ± 0.0214	0.1852 ± 0.0633	0.0808 ± 0.0239	12.3150 ± 0.8548	0.7636 ± 0.1299	78.3580 ± 1.1085
0.5	Adam (fBm-W)	0.2747 ± 0.0168	0.7349 ± 0.0506	0.2336 ± 0.0186	0.0957 ± 0.0128	11.5028 ± 0.2774	0.8609 ± 0.2679	70.5680 ± 6.6473
0.5	Adam (fBm-W(Burst))	0.2629 ± 0.0592	0.7313 ± 0.0627	0.2088 ± 0.0576	0.0913 ± 0.0211	11.6997 ± 0.9536	0.9498 ± 0.2842	77.3260 ± 1.1365
0.5	Nadam (Adaptive-fBm)	0.2040 ± 0.0219	0.8068 ± 0.0109	0.1353 ± 0.0152	0.0880 ± 0.0089	12.6523 ± 0.3534	0.5413 ± 0.0344	66.8700 ± 0.6511
0.5	Nadam (Adaptive-fBm-W)	0.2363 ± 0.0404	0.7452 ± 0.0474	0.1963 ± 0.0425	0.1083 ± 0.0145	12.1227 ± 0.6638	0.8475 ± 0.1853	66.8760 ± 1.5961
0.5	Nadam (Standard)	0.2346 ± 0.0471	0.8157 ± 0.0380	0.1765 ± 0.0560	0.1253 ± 0.0145	12.1703 ± 0.7556	0.5629 ± 0.1535	74.4220 ± 2.2627
0.5	Nadam (fBm)	0.2069 ± 0.0365	0.7029 ± 0.0369	0.1337 ± 0.0280	0.0484 ± 0.0084	12.5972 ± 0.5948	0.8215 ± 0.0942	66.8040 ± 1.3129
0.5	Nadam (fBm(Burst))	0.2300 ± 0.0482	0.7808 ± 0.0771	0.1895 ± 0.0580	0.1050 ± 0.0376	12.2323 ± 0.7757	0.6477 ± 0.2543	77.5840 ± 1.4884
0.5	Nadam (fBm-W)	0.2076 ± 0.0217	0.7952 ± 0.0391	0.1600 ± 0.0258	0.1067 ± 0.0178	12.5684 ± 0.3417	0.6208 ± 0.1520	63.4060 ± 1.1709
0.5	Nadam (fBm-W(Burst))	0.2648 ± 0.0494	0.8059 ± 0.0446	0.2318 ± 0.0668	0.1195 ± 0.0184	11.6746 ± 0.8034	0.5875 ± 0.1543	77.4820 ± 0.0789
0.5	RMSprop (Adaptive-fBm)	0.1927 ± 0.0464	0.8926 ± 0.0140	0.1178 ± 0.0475	0.1603 ± 0.0069	12.8614 ± 0.7453	0.3767 ± 0.0668	66.4760 ± 8.8092
0.5	RMSprop (Adaptive-fBm-W)	0.2031 ± 0.0298	0.8823 ± 0.0366	0.1505 ± 0.0389	0.1582 ± 0.0166	12.6912 ± 0.4774	0.4626 ± 0.1888	71.6940 ± 9.6681
0.5	RMSprop (Standard)	0.2111 ± 0.0458	0.7879 ± 0.0763	0.1554 ± 0.0360	0.0973 ± 0.0157	12.5446 ± 0.7419	0.6492 ± 0.3325	75.5240 ± 3.5606
0.5	RMSprop (fBm)	0.1701 ± 0.0408	0.8804 ± 0.0168	0.1041 ± 0.0532	0.1594 ± 0.0080	13.2207 ± 0.6561	0.4067 ± 0.0645	62.9760 ± 0.5061
0.5	RMSprop (fBm(Burst))	0.2009 ± 0.0477	0.8922 ± 0.0270	0.1312 ± 0.0473	0.1621 ± 0.0158	12.7285 ± 0.7634	0.3998 ± 0.1293	76.9540 ± 0.1785
0.5	RMSprop (fBm-W)	0.2323 ± 0.0469	0.8499 ± 0.0521	0.1898 ± 0.0648	0.1440 ± 0.0211	12.2220 ± 0.7505	0.6781 ± 0.3205	62.5160 ± 0.9148
0.5	RMSprop (fBm-W(Burst))	0.2183 ± 0.0459	0.8678 ± 0.0385	0.1574 ± 0.0546	0.1532 ± 0.0153	12.4472 ± 0.7346	0.5469 ± 0.2125	78.0620 ± 0.1522
0.5	SGD (Adaptive-fBm)	0.2605 ± 0.0209	0.7655 ± 0.0639	0.2431 ± 0.0326	0.0821 ± 0.0126	11.7049 ± 0.3467	0.7112 ± 0.2754	70.3560 ± 10.8651

Continued on next page

Table A.2: Results for CIFAR dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.5	SGD (Adaptive-fBm-W)	0.2365 ± 0.0111	0.7957 ± 0.0720	0.2048 ± 0.0172	0.0957 ± 0.0180	12.0854 ± 0.1932	0.5969 ± 0.2408	66.1360 ± 9.0044
0.5	SGD (Standard)	0.2202 ± 0.0477	0.7601 ± 0.0220	0.1968 ± 0.0607	0.0827 ± 0.0071	12.3799 ± 0.7714	0.7778 ± 0.1230	74.6860 ± 3.4782
0.5	SGD (fBm)	0.2693 ± 0.0371	0.8384 ± 0.0254	0.2483 ± 0.0377	0.1021 ± 0.0089	11.5844 ± 0.6039	0.4551 ± 0.0828	66.3980 ± 8.8412
0.5	SGD (fBm(Burst))	0.2744 ± 0.0491	0.7035 ± 0.0922	0.2336 ± 0.0555	0.0735 ± 0.0149	11.4931 ± 0.7939	1.0359 ± 0.5847	78.2340 ± 0.7231
0.5	SGD (fBm-W)	0.2178 ± 0.0278	0.8301 ± 0.0418	0.1576 ± 0.0162	0.1033 ± 0.0137	12.4160 ± 0.4422	0.4773 ± 0.1241	66.9960 ± 8.6003
0.5	SGD (fBm-W(Burst))	0.2666 ± 0.0535	0.8030 ± 0.0404	0.2147 ± 0.0639	0.0919 ± 0.0108	11.6289 ± 0.8820	0.5628 ± 0.1275	77.4320 ± 0.6975
0.7	Adam (Adaptive-fBm)	0.1974 ± 0.0496	0.7562 ± 0.0313	0.1517 ± 0.0585	0.0605 ± 0.0047	12.7331 ± 0.8113	0.6988 ± 0.0997	66.4820 ± 8.8294
0.7	Adam (Adaptive-fBm-W)	0.2730 ± 0.0262	0.7753 ± 0.0742	0.2160 ± 0.0424	0.0968 ± 0.0212	11.5362 ± 0.4276	0.7644 ± 0.3673	63.0420 ± 0.0315
0.7	Adam (Standard)	0.2955 ± 0.0647	0.7745 ± 0.0784	0.2648 ± 0.0685	0.1008 ± 0.0184	11.1627 ± 1.0495	0.7442 ± 0.3699	73.2100 ± 3.3143
0.7	Adam (fBm)	0.2036 ± 0.0613	0.7019 ± 0.1088	0.1424 ± 0.0741	0.0579 ± 0.0312	12.6646 ± 0.9944	0.8722 ± 0.3001	62.9060 ± 8.7173
0.7	Adam (fBm(Burst))	0.2157 ± 0.0371	0.8149 ± 0.0553	0.1702 ± 0.0390	0.1148 ± 0.0147	12.4603 ± 0.5999	0.5468 ± 0.2377	79.9660 ± 1.3902
0.7	Adam (fBm-W)	0.2917 ± 0.0612	0.7693 ± 0.0660	0.2449 ± 0.0586	0.1005 ± 0.0214	11.2340 ± 0.9787	0.7164 ± 0.2751	62.7380 ± 8.2800
0.7	Adam (fBm-W(Burst))	0.2469 ± 0.0386	0.7428 ± 0.0721	0.1933 ± 0.0550	0.0979 ± 0.0247	11.9625 ± 0.6275	0.9004 ± 0.3704	79.3680 ± 1.6169
0.7	Nadam (Adaptive-fBm)	0.1832 ± 0.0280	0.7863 ± 0.0469	0.1114 ± 0.0223	0.0720 ± 0.0156	12.9936 ± 0.4492	0.5981 ± 0.1156	62.6100 ± 8.7042
0.7	Nadam (Adaptive-fBm-W)	0.2399 ± 0.0606	0.7452 ± 0.1095	0.1853 ± 0.0501	0.1076 ± 0.0291	12.0800 ± 0.9797	0.9574 ± 0.5340	74.4400 ± 2.6899
0.7	Nadam (Standard)	0.2412 ± 0.0389	0.8092 ± 0.0890	0.2035 ± 0.0486	0.1256 ± 0.0335	12.0505 ± 0.6363	0.6074 ± 0.3518	74.7380 ± 3.6280
0.7	Nadam (fBm)	0.1852 ± 0.0583	0.8465 ± 0.0312	0.1302 ± 0.0652	0.1246 ± 0.0122	12.9621 ± 0.9391	0.4380 ± 0.1256	70.2880 ± 10.9390
0.7	Nadam (fBm(Burst))	0.2097 ± 0.0578	0.7756 ± 0.0430	0.1610 ± 0.0617	0.1016 ± 0.0269	12.5650 ± 0.9383	0.7183 ± 0.1516	78.6800 ± 0.1817
0.7	Nadam (fBm-W)	0.2765 ± 0.0154	0.8737 ± 0.0088	0.2308 ± 0.0230	0.1462 ± 0.0066	11.4726 ± 0.2438	0.3354 ± 0.0259	70.8260 ± 10.4238
0.7	Nadam (fBm-W(Burst))	0.2266 ± 0.0630	0.8001 ± 0.0661	0.1824 ± 0.0638	0.1207 ± 0.0231	12.2887 ± 1.0171	0.6020 ± 0.2399	78.7060 ± 0.0811
0.7	RMSprop (Adaptive-fBm)	0.2012 ± 0.0536	0.8468 ± 0.0278	0.1434 ± 0.0578	0.1386 ± 0.0123	12.7229 ± 0.8618	0.6767 ± 0.1919	66.5380 ± 8.8754
0.7	RMSprop (Adaptive-fBm-W)	0.1878 ± 0.0524	0.8844 ± 0.0430	0.1256 ± 0.0655	0.1588 ± 0.0183	12.9387 ± 0.8434	0.4783 ± 0.3250	62.8700 ± 0.9302
0.7	RMSprop (Standard)	0.2467 ± 0.0475	0.6970 ± 0.1208	0.2060 ± 0.0492	0.0832 ± 0.0224	11.9436 ± 0.7632	1.1759 ± 0.7037	73.0420 ± 1.5827
0.7	RMSprop (fBm)	0.2306 ± 0.0332	0.8917 ± 0.0343	0.1793 ± 0.0434	0.1612 ± 0.0102	12.2484 ± 0.5362	0.4071 ± 0.1669	63.0400 ± 1.2140
0.7	RMSprop (fBm(Burst))	0.2176 ± 0.0718	0.8650 ± 0.0878	0.1587 ± 0.0902	0.1480 ± 0.0322	12.4577 ± 1.1516	0.7292 ± 0.8582	78.2720 ± 0.1117

Continued on next page

Table A.2: Results for CIFAR dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.7	RMSprop (fBm-W)	0.2058 ± 0.0551	0.8548 ± 0.0563	0.1418 ± 0.0604	0.1441 ± 0.0203	12.6513 ± 0.8827	0.6659 ± 0.3698	62.6220 ± 0.8927
0.7	RMSprop (fBm-W(Burst))	0.2259 ± 0.0801	0.8996 ± 0.0469	0.1637 ± 0.0851	0.1620 ± 0.0195	12.3259 ± 1.2836	0.3796 ± 0.2442	79.3280 ± 0.1154
0.7	SGD (Adaptive-fBm)	0.2847 ± 0.0688	0.7303 ± 0.1163	0.2417 ± 0.0761	0.0795 ± 0.0247	11.3428 ± 1.1137	0.9523 ± 0.7119	66.0500 ± 9.1271
0.7	SGD (Adaptive-fBm-W)	0.2528 ± 0.0544	0.8004 ± 0.0238	0.1984 ± 0.0509	0.0966 ± 0.0073	11.8624 ± 0.8765	0.5792 ± 0.0701	62.2460 ± 0.8660
0.7	SGD (Standard)	0.2495 ± 0.0348	0.7617 ± 0.0583	0.1989 ± 0.0204	0.0898 ± 0.0154	11.9072 ± 0.5657	0.7435 ± 0.2462	70.9960 ± 3.6185
0.7	SGD (fBm)	0.2769 ± 0.0471	0.7771 ± 0.0726	0.2427 ± 0.0604	0.0857 ± 0.0160	11.4523 ± 0.7443	0.6887 ± 0.3044	66.1660 ± 8.9903
0.7	SGD (fBm(Burst))	0.2658 ± 0.0682	0.7253 ± 0.0430	0.2151 ± 0.0790	0.0764 ± 0.0065	11.6384 ± 1.0976	0.8860 ± 0.2437	79.0680 ± 0.2659
0.7	SGD (fBm-W)	0.2494 ± 0.0410	0.7791 ± 0.0356	0.1971 ± 0.0220	0.0835 ± 0.0098	11.9091 ± 0.6543	0.6340 ± 0.1202	70.2720 ± 11.0238
0.7	SGD (fBm-W(Burst))	0.2606 ± 0.0290	0.7930 ± 0.0488	0.2192 ± 0.0338	0.0924 ± 0.0133	11.7286 ± 0.4643	0.6151 ± 0.2233	78.7820 ± 0.6432

Table A.3: Results for MNIST dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	Adam (Adaptive-fBm)	0.9304 ± 0.0214	0.9356 ± 0.0254	0.9297 ± 0.0217	0.0049 ± 0.0072	0.4560 ± 0.2284	0.3989 ± 0.2291	21.1540 ± 0.4675
0.2	Adam (Adaptive-fBm-W)	0.9743 ± 0.0016	0.9968 ± 0.0018	0.9741 ± 0.0016	0.0240 ± 0.0013	0.1116 ± 0.0160	0.0113 ± 0.0063	30.1360 ± 9.9600
0.2	Adam (Standard)	0.9763 ± 0.0020	0.9996 ± 0.0005	0.9762 ± 0.0020	0.0241 ± 0.0013	0.1018 ± 0.0055	0.0021 ± 0.0019	23.1740 ± 1.1366
0.2	Adam (fBm)	0.9174 ± 0.0169	0.9201 ± 0.0211	0.9163 ± 0.0172	0.0026 ± 0.0050	0.8005 ± 0.3728	0.7506 ± 0.4142	25.1440 ± 4.4748
0.2	Adam (fBm(Burst))	0.9319 ± 0.0258	0.9409 ± 0.0287	0.9309 ± 0.0265	0.0089 ± 0.0052	0.3450 ± 0.1492	0.2647 ± 0.1456	22.9680 ± 0.8536
0.2	Adam (fBm-W)	0.9771 ± 0.0021	0.9992 ± 0.0012	0.9769 ± 0.0022	0.0238 ± 0.0015	0.0980 ± 0.0121	0.0033 ± 0.0033	25.8820 ± 8.4789
0.2	Adam (fBm-W(Burst))	0.9789 ± 0.0010	1.0000 ± 0.0000	0.9788 ± 0.0010	0.0224 ± 0.0007	0.0912 ± 0.0057	0.0008 ± 0.0002	22.9920 ± 1.1194
0.2	Nadam (Adaptive-fBm)	0.9098 ± 0.0101	0.9146 ± 0.0105	0.9087 ± 0.0102	0.0021 ± 0.0028	0.5996 ± 0.1258	0.5493 ± 0.1436	21.0480 ± 0.3724
0.2	Nadam (Adaptive-fBm-W)	0.9756 ± 0.0014	0.9991 ± 0.0002	0.9753 ± 0.0015	0.0243 ± 0.0017	0.1088 ± 0.0174	0.0057 ± 0.0020	22.9720 ± 0.5378
0.2	Nadam (Standard)	0.9766 ± 0.0013	0.9990 ± 0.0010	0.9764 ± 0.0013	0.0243 ± 0.0012	0.1078 ± 0.0088	0.0040 ± 0.0033	23.3040 ± 0.7165
0.2	Nadam (fBm)	0.9285 ± 0.0143	0.9357 ± 0.0200	0.9279 ± 0.0146	0.0086 ± 0.0074	0.5145 ± 0.2190	0.4492 ± 0.2407	25.7560 ± 8.5507
0.2	Nadam (fBm(Burst))	0.9601 ± 0.0076	0.9779 ± 0.0111	0.9597 ± 0.0078	0.0187 ± 0.0030	0.1827 ± 0.0576	0.0821 ± 0.0390	22.9300 ± 1.0194
0.2	Nadam (fBm-W)	0.9784 ± 0.0016	0.9999 ± 0.0002	0.9782 ± 0.0016	0.0232 ± 0.0010	0.0930 ± 0.0056	0.0013 ± 0.0010	22.2380 ± 0.9985
0.2	Nadam (fBm-W(Burst))	0.9780 ± 0.0006	1.0000 ± 0.0000	0.9778 ± 0.0007	0.0227 ± 0.0008	0.0990 ± 0.0055	0.0007 ± 0.0002	22.6260 ± 0.4043
0.2	RMSprop (Adaptive-fBm)	0.8944 ± 0.1030	0.9064 ± 0.1106	0.8922 ± 0.1058	0.0126 ± 0.0084	0.4981 ± 0.2056	0.3421 ± 0.3107	25.1800 ± 8.8990
0.2	RMSprop (Adaptive-fBm-W)	0.9746 ± 0.0025	0.9956 ± 0.0017	0.9744 ± 0.0025	0.0228 ± 0.0013	0.2553 ± 0.0417	0.0298 ± 0.0240	20.9740 ± 0.7375
0.2	RMSprop (Standard)	0.9795 ± 0.0010	1.0000 ± 0.0000	0.9793 ± 0.0010	0.0220 ± 0.0005	0.0922 ± 0.0050	0.0005 ± 0.0000	22.9360 ± 2.2123
0.2	RMSprop (fBm)	0.9760 ± 0.0015	0.9985 ± 0.0008	0.9758 ± 0.0014	0.0229 ± 0.0005	0.2372 ± 0.0096	0.0063 ± 0.0038	20.9740 ± 0.2794
0.2	RMSprop (fBm(Burst))	0.9580 ± 0.0160	0.9779 ± 0.0177	0.9578 ± 0.0159	0.0184 ± 0.0049	0.2432 ± 0.0350	0.0780 ± 0.0477	22.8860 ± 1.2285
0.2	RMSprop (fBm-W)	0.9760 ± 0.0009	0.9988 ± 0.0003	0.9758 ± 0.0009	0.0234 ± 0.0015	0.2570 ± 0.0128	0.0055 ± 0.0017	21.2160 ± 0.5815
0.2	RMSprop (fBm-W(Burst))	0.9758 ± 0.0015	0.9987 ± 0.0005	0.9756 ± 0.0015	0.0235 ± 0.0009	0.2624 ± 0.0150	0.0062 ± 0.0023	25.5440 ± 8.6914
0.2	SGD (Adaptive-fBm)	0.9749 ± 0.0014	0.9901 ± 0.0007	0.9747 ± 0.0014	0.0162 ± 0.0005	0.0830 ± 0.0028	0.0426 ± 0.0015	20.6380 ± 0.6541
0.2	SGD (Adaptive-fBm-W)	0.9737 ± 0.0003	0.9873 ± 0.0009	0.9734 ± 0.0004	0.0164 ± 0.0006	0.0888 ± 0.0010	0.0487 ± 0.0026	20.7180 ± 0.5137
0.2	SGD (Standard)	0.9750 ± 0.0010	0.9900 ± 0.0005	0.9748 ± 0.0010	0.0166 ± 0.0014	0.0816 ± 0.0014	0.0424 ± 0.0013	35.3060 ± 2.1915
0.2	SGD (fBm)	0.9754 ± 0.0010	0.9900 ± 0.0003	0.9752 ± 0.0010	0.0163 ± 0.0008	0.0813 ± 0.0029	0.0426 ± 0.0009	20.7240 ± 0.8541

Continued on next page

Table A.3: Results for MNIST dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	SGD (fBm(Burst))	0.9753 ± 0.0015	0.9900 ± 0.0005	0.9751 ± 0.0015	0.0165 ± 0.0012	0.0817 ± 0.0023	0.0420 ± 0.0012	26.6840 ± 5.9677
0.2	SGD (fBm-W)	0.9744 ± 0.0009	0.9875 ± 0.0009	0.9742 ± 0.0009	0.0146 ± 0.0008	0.0836 ± 0.0052	0.0481 ± 0.0024	20.9880 ± 0.5946
0.2	SGD (fBm-W(Burst))	0.9745 ± 0.0021	0.9876 ± 0.0024	0.9743 ± 0.0021	0.0145 ± 0.0015	0.0843 ± 0.0050	0.0492 ± 0.0081	21.5080 ± 0.2734
0.5	Adam (Adaptive-fBm)	0.9579 ± 0.0129	0.9721 ± 0.0153	0.9575 ± 0.0131	0.0178 ± 0.0061	0.1811 ± 0.0566	0.1123 ± 0.0615	21.1140 ± 0.2794
0.5	Adam (Adaptive-fBm-W)	0.9775 ± 0.0016	0.9993 ± 0.0008	0.9773 ± 0.0017	0.0256 ± 0.0004	0.1082 ± 0.0102	0.0033 ± 0.0019	26.8620 ± 7.9356
0.5	Adam (Standard)	0.9773 ± 0.0006	1.0000 ± 0.0001	0.9772 ± 0.0006	0.0250 ± 0.0010	0.0968 ± 0.0025	0.0010 ± 0.0004	23.7560 ± 0.5190
0.5	Adam (fBm)	0.9530 ± 0.0067	0.9653 ± 0.0097	0.9525 ± 0.0068	0.0141 ± 0.0037	0.2432 ± 0.0382	0.1565 ± 0.0457	29.7780 ± 10.2963
0.5	Adam (fBm(Burst))	0.9719 ± 0.0011	0.9976 ± 0.0030	0.9716 ± 0.0012	0.0272 ± 0.0020	0.1330 ± 0.0141	0.0153 ± 0.0128	22.4980 ± 0.7319
0.5	Adam (fBm-W)	0.9785 ± 0.0009	1.0000 ± 0.0001	0.9783 ± 0.0009	0.0225 ± 0.0009	0.0937 ± 0.0037	0.0008 ± 0.0005	25.8460 ± 8.2453
0.5	Adam (fBm-W(Burst))	0.9785 ± 0.0010	1.0000 ± 0.0000	0.9784 ± 0.0011	0.0225 ± 0.0008	0.0961 ± 0.0023	0.0007 ± 0.0003	22.5840 ± 0.7068
0.5	Nadam (Adaptive-fBm)	0.9520 ± 0.0006	0.9648 ± 0.0002	0.9516 ± 0.0007	0.0151 ± 0.0005	0.2557 ± 0.0062	0.1679 ± 0.0035	21.1520 ± 0.1784
0.5	Nadam (Adaptive-fBm-W)	0.9777 ± 0.0019	0.9999 ± 0.0002	0.9775 ± 0.0019	0.0238 ± 0.0007	0.0957 ± 0.0060	0.0014 ± 0.0009	23.2240 ± 0.3469
0.5	Nadam (Standard)	0.9781 ± 0.0007	1.0000 ± 0.0000	0.9779 ± 0.0007	0.0236 ± 0.0008	0.0989 ± 0.0019	0.0006 ± 0.0001	23.7540 ± 1.0733
0.5	Nadam (fBm)	0.9543 ± 0.0065	0.9673 ± 0.0100	0.9540 ± 0.0064	0.0145 ± 0.0027	0.2371 ± 0.0833	0.1606 ± 0.0730	21.7560 ± 0.2240
0.5	Nadam (fBm(Burst))	0.9726 ± 0.0029	0.9976 ± 0.0026	0.9724 ± 0.0029	0.0268 ± 0.0023	0.1244 ± 0.0136	0.0130 ± 0.0102	28.6220 ± 8.4701
0.5	Nadam (fBm-W)	0.9775 ± 0.0006	1.0000 ± 0.0000	0.9773 ± 0.0006	0.0232 ± 0.0001	0.1041 ± 0.0032	0.0006 ± 0.0003	25.5280 ± 8.3974
0.5	Nadam (fBm-W(Burst))	0.9778 ± 0.0002	1.0000 ± 0.0000	0.9776 ± 0.0002	0.0223 ± 0.0010	0.0999 ± 0.0054	0.0005 ± 0.0001	22.6560 ± 0.2704
0.5	RMSprop (Adaptive-fBm)	0.9778 ± 0.0015	0.9974 ± 0.0015	0.9776 ± 0.0015	0.0216 ± 0.0015	0.2619 ± 0.0145	0.0178 ± 0.0117	25.0980 ± 8.9141
0.5	RMSprop (Adaptive-fBm-W)	0.9765 ± 0.0015	0.9986 ± 0.0006	0.9762 ± 0.0015	0.0241 ± 0.0006	0.2541 ± 0.0098	0.0071 ± 0.0030	24.9240 ± 9.0122
0.5	RMSprop (Standard)	0.9802 ± 0.0005	1.0000 ± 0.0000	0.9801 ± 0.0006	0.0214 ± 0.0003	0.0897 ± 0.0018	0.0005 ± 0.0000	23.5200 ± 1.2081
0.5	RMSprop (fBm)	0.9779 ± 0.0008	0.9985 ± 0.0002	0.9777 ± 0.0008	0.0224 ± 0.0007	0.2456 ± 0.0041	0.0097 ± 0.0024	21.2700 ± 0.6877
0.5	RMSprop (fBm(Burst))	0.9764 ± 0.0013	0.9985 ± 0.0007	0.9762 ± 0.0013	0.0236 ± 0.0013	0.2503 ± 0.0157	0.0081 ± 0.0047	22.4200 ± 0.9872
0.5	RMSprop (fBm-W)	0.9779 ± 0.0007	0.9988 ± 0.0003	0.9777 ± 0.0007	0.0234 ± 0.0010	0.2394 ± 0.0111	0.0061 ± 0.0019	21.3780 ± 0.9727
0.5	RMSprop (fBm-W(Burst))	0.9770 ± 0.0008	0.9985 ± 0.0006	0.9768 ± 0.0008	0.0234 ± 0.0012	0.2498 ± 0.0116	0.0068 ± 0.0035	21.8320 ± 0.2427
0.5	SGD (Adaptive-fBm)	0.9757 ± 0.0007	0.9908 ± 0.0003	0.9755 ± 0.0007	0.0170 ± 0.0002	0.0796 ± 0.0010	0.0409 ± 0.0005	20.9420 ± 0.1270

Continued on next page

Table A.3: Results for MNIST dataset

Hurst	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE	Time (s)
0.5	SGD (Adaptive-fBm-W)	0.9775 ± 0.0010	0.9910 ± 0.0003	0.9773 ± 0.0010	0.0163 ± 0.0004	0.0759 ± 0.0022	0.0402 ± 0.0013	20.8280 ± 0.6596	
0.5	SGD (Standard)	0.9737 ± 0.0009	0.9897 ± 0.0007	0.9735 ± 0.0010	0.0170 ± 0.0006	0.0836 ± 0.0007	0.0428 ± 0.0011	35.4020 ± 1.1812	
0.5	SGD (fBm)	0.9761 ± 0.0005	0.9903 ± 0.0001	0.9759 ± 0.0005	0.0167 ± 0.0012	0.0839 ± 0.0003	0.0430 ± 0.0001	24.7300 ± 9.1469	
0.5	SGD (fBm(Burst))	0.9755 ± 0.0013	0.9901 ± 0.0004	0.9753 ± 0.0013	0.0167 ± 0.0011	0.0814 ± 0.0016	0.0419 ± 0.0008	26.5740 ± 8.4195	
0.5	SGD (fBm-W)	0.9737 ± 0.0006	0.9891 ± 0.0004	0.9735 ± 0.0006	0.0160 ± 0.0005	0.0868 ± 0.0018	0.0460 ± 0.0006	26.8140 ± 8.8905	
0.5	SGD (fBm-W(Burst))	0.9760 ± 0.0007	0.9903 ± 0.0001	0.9759 ± 0.0007	0.0166 ± 0.0011	0.0802 ± 0.0020	0.0417 ± 0.0010	25.6080 ± 8.6368	
0.7	Adam (Adaptive-fBm)	0.9679 ± 0.0047	0.9861 ± 0.0029	0.9677 ± 0.0047	0.0212 ± 0.0043	0.1254 ± 0.0182	0.0590 ± 0.0083	20.7400 ± 0.3450	
0.7	Adam (Adaptive-fBm-W)	0.9776 ± 0.0012	1.0000 ± 0.0001	0.9775 ± 0.0013	0.0231 ± 0.0006	0.0946 ± 0.0057	0.0010 ± 0.0004	22.6860 ± 0.7490	
0.7	Adam (Standard)	0.9782 ± 0.0008	1.0000 ± 0.0000	0.9781 ± 0.0008	0.0233 ± 0.0014	0.0959 ± 0.0032	0.0007 ± 0.0001	22.5880 ± 0.6863	
0.7	Adam (fBm)	0.9672 ± 0.0028	0.9885 ± 0.0054	0.9669 ± 0.0028	0.0232 ± 0.0037	0.1532 ± 0.0235	0.0486 ± 0.0188	30.0880 ± 1.0224	
0.7	Adam (fBm(Burst))	0.9759 ± 0.0012	0.9998 ± 0.0003	0.9757 ± 0.0012	0.0242 ± 0.0011	0.1132 ± 0.0111	0.0021 ± 0.0011	22.3000 ± 0.7187	
0.7	Adam (fBm-W)	0.9782 ± 0.0025	0.9990 ± 0.0021	0.9781 ± 0.0026	0.0226 ± 0.0009	0.1018 ± 0.0160	0.0034 ± 0.0063	22.8660 ± 3.2789	
0.7	Adam (fBm-W(Burst))	0.9787 ± 0.0011	0.9997 ± 0.0007	0.9785 ± 0.0011	0.0227 ± 0.0009	0.0949 ± 0.0026	0.0016 ± 0.0025	22.5800 ± 0.4188	
0.7	Nadam (Adaptive-fBm)	0.9669 ± 0.0050	0.9839 ± 0.0057	0.9666 ± 0.0050	0.0188 ± 0.0010	0.1375 ± 0.0319	0.0686 ± 0.0210	20.8600 ± 0.6258	
0.7	Nadam (Adaptive-fBm-W)	0.9785 ± 0.0003	0.9998 ± 0.0002	0.9783 ± 0.0003	0.0241 ± 0.0008	0.0968 ± 0.0077	0.0012 ± 0.0007	22.9600 ± 0.5919	
0.7	Nadam (Standard)	0.9783 ± 0.0011	0.9998 ± 0.0004	0.9781 ± 0.0011	0.0240 ± 0.0008	0.1005 ± 0.0037	0.0013 ± 0.0016	26.5780 ± 3.3329	
0.7	Nadam (fBm)	0.9602 ± 0.0094	0.9827 ± 0.0088	0.9600 ± 0.0092	0.0231 ± 0.0047	0.2074 ± 0.0738	0.0749 ± 0.0352	21.6100 ± 0.2477	
0.7	Nadam (fBm(Burst))	0.9741 ± 0.0017	0.9989 ± 0.0010	0.9739 ± 0.0017	0.0267 ± 0.0020	0.1269 ± 0.0171	0.0058 ± 0.0033	22.2880 ± 0.8134	
0.7	Nadam (fBm-W)	0.9789 ± 0.0015	1.0000 ± 0.0000	0.9788 ± 0.0015	0.0225 ± 0.0003	0.0977 ± 0.0040	0.0005 ± 0.0001	25.5740 ± 8.6778	
0.7	Nadam (fBm-W(Burst))	0.9775 ± 0.0014	0.9996 ± 0.0009	0.9774 ± 0.0014	0.0229 ± 0.0015	0.1033 ± 0.0081	0.0019 ± 0.0027	22.4020 ± 0.3717	
0.7	RMSprop (Adaptive-fBm)	0.9758 ± 0.0027	0.9978 ± 0.0009	0.9756 ± 0.0028	0.0234 ± 0.0012	0.2448 ± 0.0158	0.0101 ± 0.0038	20.6780 ± 0.4510	
0.7	RMSprop (Adaptive-fBm-W)	0.9765 ± 0.0024	0.9988 ± 0.0006	0.9763 ± 0.0024	0.0233 ± 0.0005	0.2568 ± 0.0257	0.0059 ± 0.0038	24.8240 ± 9.0839	
0.7	RMSprop (Standard)	0.9794 ± 0.0011	1.0000 ± 0.0000	0.9792 ± 0.0011	0.0225 ± 0.0005	0.0917 ± 0.0064	0.0005 ± 0.0000	21.4220 ± 0.5732	
0.7	RMSprop (fBm)	0.9758 ± 0.0015	0.9985 ± 0.0003	0.9756 ± 0.0015	0.0241 ± 0.0017	0.2569 ± 0.0144	0.0076 ± 0.0018	24.8560 ± 9.0543	
0.7	RMSprop (fBm(Burst))	0.9767 ± 0.0007	0.9987 ± 0.0005	0.9766 ± 0.0007	0.0231 ± 0.0011	0.2463 ± 0.0067	0.0063 ± 0.0027	21.3860 ± 0.6145	

Continued on next page

Table A.3: Results for MNIST dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.7	RMSprop (fBm-W)	0.9772 ± 0.0005	0.9984 ± 0.0006	0.9771 ± 0.0005	0.0237 ± 0.0007	0.2408 ± 0.0115	0.0073 ± 0.0032	24.9680 ± 8.9889
0.7	RMSprop (fBm-W(Burst))	0.9769 ± 0.0016	0.9985 ± 0.0005	0.9767 ± 0.0016	0.0231 ± 0.0013	0.2429 ± 0.0123	0.0077 ± 0.0039	21.2880 ± 0.5650
0.7	SGD (Adaptive-fBm)	0.9761 ± 0.0009	0.9903 ± 0.0003	0.9759 ± 0.0010	0.0162 ± 0.0005	0.0810 ± 0.0010	0.0421 ± 0.0011	20.5200 ± 0.6224
0.7	SGD (Adaptive-fBm-W)	0.9762 ± 0.0010	0.9904 ± 0.0005	0.9759 ± 0.0010	0.0163 ± 0.0006	0.0807 ± 0.0034	0.0416 ± 0.0011	20.4480 ± 0.3839
0.7	SGD (Standard)	0.9754 ± 0.0013	0.9899 ± 0.0004	0.9752 ± 0.0013	0.0168 ± 0.0009	0.0804 ± 0.0026	0.0430 ± 0.0009	33.5160 ± 4.1759
0.7	SGD (fBm)	0.9753 ± 0.0006	0.9902 ± 0.0004	0.9751 ± 0.0006	0.0161 ± 0.0007	0.0813 ± 0.0021	0.0419 ± 0.0014	21.0460 ± 0.5907
0.7	SGD (fBm(Burst))	0.9754 ± 0.0011	0.9901 ± 0.0004	0.9752 ± 0.0011	0.0166 ± 0.0012	0.0815 ± 0.0016	0.0419 ± 0.0008	27.8560 ± 7.7510
0.7	SGD (fBm-W)	0.9752 ± 0.0014	0.9902 ± 0.0001	0.9750 ± 0.0014	0.0165 ± 0.0003	0.0810 ± 0.0041	0.0423 ± 0.0007	23.1480 ± 5.3671
0.7	SGD (fBm-W(Burst))	0.9755 ± 0.0007	0.9902 ± 0.0001	0.9754 ± 0.0007	0.0165 ± 0.0008	0.0808 ± 0.0021	0.0416 ± 0.0011	22.0100 ± 0.9961

Table A.4: Results for NSL dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	Adam (Adaptive-fBm)	0.9277 ± 0.0134	0.9296 ± 0.0129	0.9272 ± 0.0136	-0.0010 ± 0.0023	0.3652 ± 0.1983	0.3532 ± 0.1900	59.3440 ± 12.6390
0.2	Adam (Adaptive-fBm-W)	0.9911 ± 0.0009	0.9930 ± 0.0007	0.9911 ± 0.0009	0.0015 ± 0.0004	0.0247 ± 0.0012	0.0174 ± 0.0013	64.1960 ± 9.9273
0.2	Adam (Standard)	0.9922 ± 0.0006	0.9942 ± 0.0004	0.9922 ± 0.0006	0.0018 ± 0.0004	0.0246 ± 0.0009	0.0149 ± 0.0004	52.5960 ± 1.5599
0.2	Adam (fBm)	0.9375 ± 0.0239	0.9397 ± 0.0233	0.9370 ± 0.0245	-0.0002 ± 0.0006	0.3510 ± 0.2151	0.3343 ± 0.2034	58.7500 ± 1.7545
0.2	Adam (fBm(Burst))	0.8892 ± 0.1328	0.8915 ± 0.1318	0.8773 ± 0.1589	-0.0009 ± 0.0008	0.4055 ± 0.3185	0.3948 ± 0.3106	62.1020 ± 2.6778
0.2	Adam (fBm-W)	0.9918 ± 0.0005	0.9939 ± 0.0004	0.9918 ± 0.0005	0.0021 ± 0.0002	0.0251 ± 0.0012	0.0152 ± 0.0005	61.2740 ± 11.6014
0.2	Adam (fBm-W(Burst))	0.9920 ± 0.0005	0.9941 ± 0.0003	0.9920 ± 0.0005	0.0019 ± 0.0005	0.0248 ± 0.0015	0.0148 ± 0.0004	59.7340 ± 1.3904
0.2	Nadam (Adaptive-fBm)	0.9368 ± 0.0095	0.9376 ± 0.0101	0.9367 ± 0.0095	-0.0007 ± 0.0017	0.5933 ± 0.3415	0.5905 ± 0.3402	54.3900 ± 0.9200
0.2	Nadam (Adaptive-fBm-W)	0.9912 ± 0.0007	0.9930 ± 0.0004	0.9912 ± 0.0007	0.0017 ± 0.0003	0.0254 ± 0.0018	0.0173 ± 0.0007	65.4400 ± 9.4780
0.2	Nadam (Standard)	0.9923 ± 0.0003	0.9943 ± 0.0001	0.9922 ± 0.0003	0.0018 ± 0.0003	0.0229 ± 0.0014	0.0146 ± 0.0003	54.7720 ± 1.8456
0.2	Nadam (fBm)	0.9484 ± 0.0243	0.9507 ± 0.0232	0.9484 ± 0.0243	-0.0011 ± 0.0010	0.1324 ± 0.0413	0.1283 ± 0.0397	61.4020 ± 11.4957
0.2	Nadam (fBm(Burst))	0.9836 ± 0.0026	0.9848 ± 0.0029	0.9836 ± 0.0027	0.0002 ± 0.0005	0.0434 ± 0.0052	0.0386 ± 0.0056	61.0560 ± 0.9083
0.2	Nadam (fBm-W)	0.9922 ± 0.0001	0.9942 ± 0.0002	0.9922 ± 0.0001	0.0019 ± 0.0003	0.0237 ± 0.0009	0.0146 ± 0.0005	67.5500 ± 13.2911
0.2	Nadam (fBm-W(Burst))	0.9923 ± 0.0005	0.9941 ± 0.0003	0.9923 ± 0.0005	0.0018 ± 0.0001	0.0235 ± 0.0016	0.0150 ± 0.0007	60.1520 ± 1.2921
0.2	RMSprop (Adaptive-fBm)	0.9820 ± 0.0042	0.9839 ± 0.0042	0.9820 ± 0.0042	0.0004 ± 0.0008	0.0508 ± 0.0043	0.0380 ± 0.0047	66.3280 ± 14.5529
0.2	RMSprop (Adaptive-fBm-W)	0.9760 ± 0.0103	0.9778 ± 0.0098	0.9760 ± 0.0104	-0.0001 ± 0.0003	0.0557 ± 0.0079	0.0448 ± 0.0091	60.8520 ± 12.1264
0.2	RMSprop (Standard)	0.9909 ± 0.0006	0.9933 ± 0.0001	0.9909 ± 0.0006	0.0018 ± 0.0003	0.0284 ± 0.0013	0.0183 ± 0.0002	52.0200 ± 3.7915
0.2	RMSprop (fBm)	0.9747 ± 0.0141	0.9762 ± 0.0138	0.9746 ± 0.0143	0.0000 ± 0.0013	0.0565 ± 0.0103	0.0462 ± 0.0144	58.3180 ± 13.2213
0.2	RMSprop (fBm(Burst))	0.9838 ± 0.0034	0.9856 ± 0.0035	0.9838 ± 0.0034	-0.0000 ± 0.0003	0.0530 ± 0.0065	0.0403 ± 0.0053	61.6300 ± 11.3567
0.2	RMSprop (fBm-W)	0.9774 ± 0.0051	0.9788 ± 0.0042	0.9774 ± 0.0051	-0.0005 ± 0.0010	0.0532 ± 0.0069	0.0444 ± 0.0063	54.5540 ± 2.7482
0.2	RMSprop (fBm-W(Burst))	0.9801 ± 0.0036	0.9815 ± 0.0031	0.9800 ± 0.0036	0.0002 ± 0.0006	0.0516 ± 0.0021	0.0412 ± 0.0010	57.0820 ± 0.9348
0.2	SGD (Adaptive-fBm)	0.9907 ± 0.0004	0.9921 ± 0.0005	0.9907 ± 0.0004	0.0017 ± 0.0002	0.0263 ± 0.0001	0.0203 ± 0.0006	52.2980 ± 0.1702
0.2	SGD (Adaptive-fBm-W)	0.9870 ± 0.0041	0.9881 ± 0.0042	0.9870 ± 0.0041	0.0007 ± 0.0007	0.0378 ± 0.0079	0.0302 ± 0.0073	52.2180 ± 1.4373
0.2	SGD (Standard)	0.9901 ± 0.0005	0.9918 ± 0.0002	0.9901 ± 0.0005	0.0019 ± 0.0003	0.0273 ± 0.0011	0.0214 ± 0.0006	51.3300 ± 3.0156
0.2	SGD (fBm)	0.9906 ± 0.0005	0.9921 ± 0.0004	0.9906 ± 0.0005	0.0018 ± 0.0005	0.0259 ± 0.0011	0.0207 ± 0.0008	57.4260 ± 13.7040

Continued on next page

Table A.4: Results for NSL dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	SGD (fBm(Burst))	0.9904 ± 0.0003	0.9920 ± 0.0004	0.9904 ± 0.0003	0.0017 ± 0.0005	0.0269 ± 0.0005	0.0207 ± 0.0005	56.4060 ± 2.1354
0.2	SGD (fBm-W)	0.9899 ± 0.0001	0.9915 ± 0.0003	0.9899 ± 0.0001	0.0014 ± 0.0003	0.0275 ± 0.0007	0.0215 ± 0.0007	57.8160 ± 13.4946
0.2	SGD (fBm-W(Burst))	0.9905 ± 0.0006	0.9922 ± 0.0003	0.9905 ± 0.0006	0.0018 ± 0.0004	0.0268 ± 0.0007	0.0205 ± 0.0005	60.0280 ± 12.2483
0.5	Adam (Adaptive-fBm)	0.9860 ± 0.0024	0.9870 ± 0.0022	0.9859 ± 0.0024	0.0008 ± 0.0004	0.0386 ± 0.0067	0.0331 ± 0.0063	53.9100 ± 1.2231
0.5	Adam (Adaptive-fBm-W)	0.9920 ± 0.0002	0.9943 ± 0.0002	0.9920 ± 0.0002	0.0019 ± 0.0003	0.0242 ± 0.0006	0.0146 ± 0.0002	59.7960 ± 0.9867
0.5	Adam (Standard)	0.9920 ± 0.0003	0.9942 ± 0.0001	0.9920 ± 0.0003	0.0020 ± 0.0003	0.0238 ± 0.0007	0.0144 ± 0.0001	55.0360 ± 1.4565
0.5	Adam (fBm)	0.9759 ± 0.0137	0.9768 ± 0.0135	0.9759 ± 0.0137	-0.0004 ± 0.0006	0.0570 ± 0.0270	0.0518 ± 0.0274	68.3520 ± 12.3996
0.5	Adam (fBm(Burst))	0.9905 ± 0.0010	0.9914 ± 0.0021	0.9904 ± 0.0010	0.0011 ± 0.0007	0.0288 ± 0.0034	0.0212 ± 0.0052	66.7940 ± 7.5459
0.5	Adam (fBm-W)	0.9919 ± 0.0002	0.9941 ± 0.0001	0.9919 ± 0.0002	0.0021 ± 0.0003	0.0245 ± 0.0013	0.0145 ± 0.0003	56.5400 ± 1.5832
0.5	Adam (fBm-W(Burst))	0.9922 ± 0.0003	0.9941 ± 0.0001	0.9922 ± 0.0003	0.0019 ± 0.0002	0.0233 ± 0.0013	0.0147 ± 0.0002	57.3380 ± 1.1115
0.5	Nadam (Adaptive-fBm)	0.9871 ± 0.0009	0.9876 ± 0.0006	0.9871 ± 0.0008	0.0001 ± 0.0003	0.0359 ± 0.0020	0.0321 ± 0.0025	55.6200 ± 1.4914
0.5	Nadam (Adaptive-fBm-W)	0.9925 ± 0.0004	0.9942 ± 0.0001	0.9925 ± 0.0004	0.0022 ± 0.0004	0.0244 ± 0.0023	0.0148 ± 0.0002	68.7260 ± 12.0831
0.5	Nadam (Standard)	0.9925 ± 0.0002	0.9942 ± 0.0002	0.9925 ± 0.0002	0.0018 ± 0.0001	0.0241 ± 0.0011	0.0147 ± 0.0003	54.1100 ± 2.9453
0.5	Nadam (fBm)	0.9775 ± 0.0076	0.9787 ± 0.0072	0.9775 ± 0.0076	-0.0006 ± 0.0010	0.0661 ± 0.0213	0.0608 ± 0.0212	61.7540 ± 11.3137
0.5	Nadam (fBm(Burst))	0.9912 ± 0.0006	0.9930 ± 0.0009	0.9912 ± 0.0006	0.0016 ± 0.0007	0.0264 ± 0.0017	0.0180 ± 0.0020	57.4420 ± 1.0181
0.5	Nadam (fBm-W)	0.9923 ± 0.0006	0.9943 ± 0.0003	0.9923 ± 0.0006	0.0021 ± 0.0001	0.0246 ± 0.0014	0.0144 ± 0.0002	57.6660 ± 0.7130
0.5	Nadam (fBm-W(Burst))	0.9926 ± 0.0002	0.9942 ± 0.0002	0.9926 ± 0.0002	0.0019 ± 0.0003	0.0233 ± 0.0007	0.0152 ± 0.0003	56.9120 ± 0.7265
0.5	RMSprop (Adaptive-fBm)	0.9796 ± 0.0021	0.9807 ± 0.0018	0.9796 ± 0.0021	-0.0001 ± 0.0003	0.0519 ± 0.0030	0.0418 ± 0.0033	54.0380 ± 0.8584
0.5	RMSprop (Adaptive-fBm-W)	0.9774 ± 0.0040	0.9790 ± 0.0034	0.9774 ± 0.0040	0.0003 ± 0.0005	0.0577 ± 0.0038	0.0441 ± 0.0035	64.8840 ± 15.5738
0.5	RMSprop (Standard)	0.9912 ± 0.0002	0.9935 ± 0.0001	0.9912 ± 0.0002	0.0022 ± 0.0003	0.0275 ± 0.0008	0.0180 ± 0.0004	52.4340 ± 2.9380
0.5	RMSprop (fBm)	0.9822 ± 0.0013	0.9838 ± 0.0011	0.9822 ± 0.0013	0.0003 ± 0.0003	0.0512 ± 0.0020	0.0380 ± 0.0024	53.7500 ± 1.7611
0.5	RMSprop (fBm(Burst))	0.9773 ± 0.0074	0.9785 ± 0.0066	0.9772 ± 0.0075	0.0003 ± 0.0004	0.0573 ± 0.0026	0.0426 ± 0.0036	53.8460 ± 1.3882
0.5	RMSprop (fBm-W)	0.9787 ± 0.0023	0.9801 ± 0.0024	0.9786 ± 0.0023	-0.0004 ± 0.0005	0.0526 ± 0.0019	0.0413 ± 0.0018	59.7420 ± 12.4704
0.5	RMSprop (fBm-W(Burst))	0.9782 ± 0.0034	0.9798 ± 0.0034	0.9782 ± 0.0034	0.0002 ± 0.0009	0.0502 ± 0.0059	0.0421 ± 0.0049	53.1820 ± 1.0422
0.5	SGD (Adaptive-fBm)	0.9908 ± 0.0004	0.9923 ± 0.0004	0.9908 ± 0.0004	0.0020 ± 0.0002	0.0260 ± 0.0008	0.0202 ± 0.0004	52.7300 ± 1.1042

Continued on next page

Table A.4: Results for NSL dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.5	SGD (Adaptive-fBm-W)	0.9905 ± 0.0003	0.9916 ± 0.0004	0.9905 ± 0.0003	0.0013 ± 0.0003	0.0269 ± 0.0006	0.0216 ± 0.0004	52.2480 ± 0.7427
0.5	SGD (Standard)	0.9896 ± 0.0007	0.9915 ± 0.0004	0.9895 ± 0.0007	0.0022 ± 0.0003	0.0284 ± 0.0015	0.0219 ± 0.0008	54.2920 ± 2.2926
0.5	SGD (fBm)	0.9907 ± 0.0008	0.9922 ± 0.0003	0.9907 ± 0.0008	0.0020 ± 0.0003	0.0255 ± 0.0011	0.0203 ± 0.0006	57.2460 ± 13.8099
0.5	SGD (fBm(Burst))	0.9905 ± 0.0002	0.9920 ± 0.0003	0.9905 ± 0.0002	0.0019 ± 0.0004	0.0267 ± 0.0008	0.0209 ± 0.0005	69.1260 ± 12.6790
0.5	SGD (fBm-W)	0.9902 ± 0.0003	0.9920 ± 0.0003	0.9902 ± 0.0003	0.0014 ± 0.0002	0.0264 ± 0.0003	0.0207 ± 0.0003	58.1940 ± 13.3129
0.5	SGD (fBm-W(Burst))	0.9904 ± 0.0004	0.9922 ± 0.0001	0.9904 ± 0.0004	0.0021 ± 0.0004	0.0270 ± 0.0006	0.0205 ± 0.0004	52.5180 ± 0.5209
0.7	Adam (Adaptive-fBm)	0.9907 ± 0.0008	0.9920 ± 0.0007	0.9907 ± 0.0008	0.0013 ± 0.0002	0.0272 ± 0.0015	0.0208 ± 0.0017	65.0640 ± 15.4206
0.7	Adam (Adaptive-fBm-W)	0.9920 ± 0.0007	0.9941 ± 0.0002	0.9920 ± 0.0007	0.0018 ± 0.0004	0.0232 ± 0.0006	0.0145 ± 0.0002	65.0740 ± 7.7029
0.7	Adam (Standard)	0.9921 ± 0.0001	0.9940 ± 0.0002	0.9921 ± 0.0001	0.0021 ± 0.0002	0.0241 ± 0.0010	0.0147 ± 0.0003	53.3220 ± 1.5336
0.7	Adam (fBm)	0.9891 ± 0.0023	0.9904 ± 0.0021	0.9891 ± 0.0023	0.0010 ± 0.0006	0.0303 ± 0.0038	0.0231 ± 0.0038	63.7940 ± 10.2911
0.7	Adam (fBm(Burst))	0.9921 ± 0.0001	0.9938 ± 0.0003	0.9921 ± 0.0001	0.0018 ± 0.0004	0.0241 ± 0.0004	0.0158 ± 0.0007	76.6240 ± 5.6054
0.7	Adam (fBm-W)	0.9919 ± 0.0003	0.9942 ± 0.0003	0.9918 ± 0.0003	0.0023 ± 0.0003	0.0246 ± 0.0010	0.0144 ± 0.0005	56.3560 ± 4.8114
0.7	Adam (fBm-W(Burst))	0.9921 ± 0.0003	0.9941 ± 0.0003	0.9921 ± 0.0003	0.0023 ± 0.0004	0.0239 ± 0.0013	0.0147 ± 0.0003	61.9260 ± 11.4781
0.7	Nadam (Adaptive-fBm)	0.9901 ± 0.0015	0.9916 ± 0.0007	0.9901 ± 0.0015	0.0014 ± 0.0002	0.0279 ± 0.0030	0.0215 ± 0.0020	54.4520 ± 7.7880
0.7	Nadam (Adaptive-fBm-W)	0.9921 ± 0.0002	0.9942 ± 0.0001	0.9920 ± 0.0003	0.0020 ± 0.0003	0.0245 ± 0.0012	0.0146 ± 0.0002	63.8960 ± 5.1088
0.7	Nadam (Standard)	0.9921 ± 0.0006	0.9942 ± 0.0003	0.9921 ± 0.0006	0.0020 ± 0.0002	0.0246 ± 0.0008	0.0150 ± 0.0005	54.0880 ± 3.2202
0.7	Nadam (fBm)	0.9839 ± 0.0066	0.9849 ± 0.0060	0.9839 ± 0.0066	0.0008 ± 0.0004	0.0405 ± 0.0132	0.0348 ± 0.0142	56.7900 ± 0.9301
0.7	Nadam (fBm(Burst))	0.9915 ± 0.0010	0.9935 ± 0.0006	0.9915 ± 0.0010	0.0016 ± 0.0004	0.0248 ± 0.0013	0.0157 ± 0.0008	57.4400 ± 1.2629
0.7	Nadam (fBm-W)	0.9922 ± 0.0004	0.9941 ± 0.0004	0.9922 ± 0.0004	0.0019 ± 0.0003	0.0241 ± 0.0016	0.0148 ± 0.0008	56.3480 ± 0.9999
0.7	Nadam (fBm-W(Burst))	0.9926 ± 0.0005	0.9943 ± 0.0001	0.9926 ± 0.0005	0.0018 ± 0.0002	0.0229 ± 0.0010	0.0146 ± 0.0004	57.9160 ± 1.8327
0.7	RMSprop (Adaptive-fBm)	0.9803 ± 0.0031	0.9814 ± 0.0028	0.9802 ± 0.0031	-0.0001 ± 0.0007	0.0555 ± 0.0080	0.0456 ± 0.0079	61.3900 ± 11.6513
0.7	RMSprop (Adaptive-fBm-W)	0.9776 ± 0.0037	0.9791 ± 0.0033	0.9776 ± 0.0037	0.0004 ± 0.0014	0.0573 ± 0.0072	0.0453 ± 0.0061	59.3220 ± 12.6579
0.7	RMSprop (Standard)	0.9910 ± 0.0005	0.9931 ± 0.0003	0.9910 ± 0.0005	0.0019 ± 0.0005	0.0287 ± 0.0026	0.0188 ± 0.0009	50.7200 ± 2.5261
0.7	RMSprop (fBm)	0.9716 ± 0.0156	0.9732 ± 0.0152	0.9715 ± 0.0156	0.0000 ± 0.0005	0.0647 ± 0.0207	0.0520 ± 0.0195	52.9140 ± 0.7174
0.7	RMSprop (fBm(Burst))	0.9790 ± 0.0026	0.9809 ± 0.0024	0.9790 ± 0.0026	-0.0005 ± 0.0003	0.0539 ± 0.0051	0.0424 ± 0.0009	54.6960 ± 1.7792

Continued on next page

Table A.4: Results for NSL dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.7	RMSprop (fBm-W)	0.9787 ± 0.0041	0.9805 ± 0.0042	0.9787 ± 0.0041	0.0001 ± 0.0008	0.0600 ± 0.0167	0.0463 ± 0.0130	59.5120 ± 12.5501
0.7	RMSprop (fBm-W(Burst))	0.9809 ± 0.0018	0.9823 ± 0.0018	0.9809 ± 0.0018	0.0003 ± 0.0004	0.0512 ± 0.0043	0.0397 ± 0.0030	54.2580 ± 1.1166
0.7	SGD (Adaptive-fBm)	0.9908 ± 0.0004	0.9922 ± 0.0004	0.9908 ± 0.0004	0.0017 ± 0.0002	0.0262 ± 0.0006	0.0203 ± 0.0003	64.3940 ± 16.0203
0.7	SGD (Adaptive-fBm-W)	0.9907 ± 0.0003	0.9920 ± 0.0002	0.9907 ± 0.0003	0.0013 ± 0.0003	0.0262 ± 0.0008	0.0205 ± 0.0007	58.0340 ± 13.3762
0.7	SGD (Standard)	0.9910 ± 0.0002	0.9923 ± 0.0001	0.9910 ± 0.0002	0.0017 ± 0.0005	0.0260 ± 0.0005	0.0202 ± 0.0005	52.0060 ± 2.5760
0.7	SGD (fBm)	0.9902 ± 0.0002	0.9920 ± 0.0001	0.9902 ± 0.0002	0.0018 ± 0.0003	0.0265 ± 0.0008	0.0207 ± 0.0003	51.5740 ± 0.8549
0.7	SGD (fBm(Burst))	0.9905 ± 0.0002	0.9920 ± 0.0002	0.9905 ± 0.0002	0.0019 ± 0.0002	0.0267 ± 0.0006	0.0207 ± 0.0004	56.7840 ± 6.7078
0.7	SGD (fBm-W)	0.9906 ± 0.0002	0.9921 ± 0.0003	0.9905 ± 0.0002	0.0015 ± 0.0002	0.0258 ± 0.0006	0.0205 ± 0.0003	51.3600 ± 0.8856
0.7	SGD (fBm-W(Burst))	0.9905 ± 0.0004	0.9922 ± 0.0003	0.9905 ± 0.0004	0.0018 ± 0.0004	0.0274 ± 0.0010	0.0206 ± 0.0002	58.0060 ± 13.3943

Table A.5: Results for bot-iot dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	Adam (Adaptive-fBm)	0.9214 ± 0.1693	0.9967 ± 0.0023	0.5758 ± 0.2425	0.0765 ± 0.1713	0.1021 ± 0.1799	0.0218 ± 0.0148	34.2420 ± 10.5790
0.2	Adam (Adaptive-fBm-W)	0.9979 ± 0.0019	0.9986 ± 0.0021	0.7672 ± 0.2476	0.0002 ± 0.0004	0.0171 ± 0.0117	0.0081 ± 0.0094	35.4320 ± 11.6572
0.2	Adam (Standard)	0.9939 ± 0.0040	1.0000 ± 0.0000	0.8024 ± 0.1216	0.0051 ± 0.0039	0.0196 ± 0.0143	0.0000 ± 0.0000	41.3440 ± 17.6205
0.2	Adam (fBm)	0.9967 ± 0.0022	0.9967 ± 0.0023	0.5992 ± 0.2241	-0.0001 ± 0.0001	0.0225 ± 0.0148	0.0208 ± 0.0144	35.0980 ± 10.3480
0.2	Adam (fBm(Burst))	0.9967 ± 0.0022	0.9967 ± 0.0023	0.5992 ± 0.2241	-0.0001 ± 0.0001	0.0218 ± 0.0135	0.0200 ± 0.0133	38.2000 ± 9.9795
0.2	Adam (fBm-W)	0.9939 ± 0.0040	1.0000 ± 0.0000	0.8024 ± 0.1216	0.0053 ± 0.0040	0.0270 ± 0.0187	0.0000 ± 0.0000	34.3080 ± 10.5292
0.2	Adam (fBm-W(Burst))	0.9948 ± 0.0046	1.0000 ± 0.0000	0.8322 ± 0.1399	0.0047 ± 0.0042	0.0139 ± 0.0106	0.0001 ± 0.0001	34.8660 ± 10.8813
0.2	Nadam (Adaptive-fBm)	0.9214 ± 0.1693	0.9967 ± 0.0023	0.5758 ± 0.2425	0.0765 ± 0.1713	1.2220 ± 2.6903	0.0209 ± 0.0139	34.3280 ± 10.3819
0.2	Nadam (Adaptive-fBm-W)	0.9975 ± 0.0024	0.9976 ± 0.0025	0.6959 ± 0.2698	0.0000 ± 0.0002	0.0194 ± 0.0204	0.0177 ± 0.0200	36.1660 ± 10.9838
0.2	Nadam (Standard)	0.9956 ± 0.0047	1.0000 ± 0.0000	0.8668 ± 0.1111	0.0039 ± 0.0045	0.0175 ± 0.0150	0.0001 ± 0.0002	40.0640 ± 16.1308
0.2	Nadam (fBm)	0.9967 ± 0.0022	0.9967 ± 0.0023	0.5992 ± 0.2241	-0.0001 ± 0.0001	0.0218 ± 0.0142	0.0209 ± 0.0135	34.2000 ± 10.6127
0.2	Nadam (fBm(Burst))	0.9967 ± 0.0022	0.9967 ± 0.0023	0.5992 ± 0.2241	-0.0001 ± 0.0001	0.0210 ± 0.0136	0.0205 ± 0.0136	35.6380 ± 11.7392
0.2	Nadam (fBm-W)	0.9943 ± 0.0043	1.0000 ± 0.0000	0.8140 ± 0.1268	0.0052 ± 0.0042	0.0271 ± 0.0162	0.0000 ± 0.0000	34.6740 ± 10.0595
0.2	Nadam (fBm-W(Burst))	0.9971 ± 0.0039	1.0000 ± 0.0001	0.9070 ± 0.1017	0.0030 ± 0.0039	0.0081 ± 0.0063	0.0001 ± 0.0001	35.2500 ± 10.9688
0.2	RMSprop (Adaptive-fBm)	0.9973 ± 0.0023	0.9977 ± 0.0025	0.7424 ± 0.2306	0.0003 ± 0.0005	0.0220 ± 0.0169	0.0156 ± 0.0160	34.1740 ± 11.3740
0.2	RMSprop (Adaptive-fBm-W)	0.9975 ± 0.0040	0.9999 ± 0.0001	0.9219 ± 0.1057	0.0026 ± 0.0036	0.0167 ± 0.0262	0.0004 ± 0.0008	34.3200 ± 11.2278
0.2	RMSprop (Standard)	0.9951 ± 0.0035	1.0000 ± 0.0000	0.8282 ± 0.1221	0.0045 ± 0.0036	0.0172 ± 0.0102	0.0000 ± 0.0000	38.5600 ± 16.4063
0.2	RMSprop (fBm)	0.9965 ± 0.0035	1.0000 ± 0.0000	0.8643 ± 0.1338	0.0037 ± 0.0033	0.0195 ± 0.0174	0.0000 ± 0.0000	37.2260 ± 10.3624
0.2	RMSprop (fBm(Burst))	0.9991 ± 0.0011	0.9999 ± 0.0001	0.9540 ± 0.0491	0.0010 ± 0.0015	0.0081 ± 0.0104	0.0006 ± 0.0006	34.5880 ± 12.3304
0.2	RMSprop (fBm-W)	0.9960 ± 0.0045	1.0000 ± 0.0000	0.8800 ± 0.1217	0.0035 ± 0.0042	0.0487 ± 0.0619	0.0001 ± 0.0002	34.2880 ± 11.4108
0.2	RMSprop (fBm-W(Burst))	0.9977 ± 0.0031	1.0000 ± 0.0000	0.9078 ± 0.0851	0.0026 ± 0.0030	0.0171 ± 0.0196	0.0001 ± 0.0003	33.7220 ± 11.1361
0.2	SGD (Adaptive-fBm)	0.9995 ± 0.0005	0.9999 ± 0.0001	0.9762 ± 0.0253	0.0017 ± 0.0018	0.0032 ± 0.0026	0.0008 ± 0.0005	33.9300 ± 10.6973
0.2	SGD (Adaptive-fBm-W)	0.9977 ± 0.0020	0.9979 ± 0.0022	0.7216 ± 0.2393	0.0005 ± 0.0013	0.0123 ± 0.0113	0.0096 ± 0.0115	33.9040 ± 10.5844
0.2	SGD (Standard)	0.9982 ± 0.0018	0.9999 ± 0.0001	0.9278 ± 0.0726	0.0021 ± 0.0022	0.0084 ± 0.0075	0.0003 ± 0.0004	42.5340 ± 18.6805
0.2	SGD (fBm)	0.9988 ± 0.0013	0.9999 ± 0.0001	0.9444 ± 0.0571	0.0018 ± 0.0021	0.0068 ± 0.0081	0.0003 ± 0.0004	33.7380 ± 10.9800

Continued on next page

Table A.5: Results for bot-iot dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.2	SGD (fBm(Burst))	0.9987 ± 0.0018	0.9999 ± 0.0001	0.9463 ± 0.0686	0.0019 ± 0.0019	0.0056 ± 0.0078	0.0005 ± 0.0006	33.6540 ± 11.4788
0.2	SGD (fBm-W)	0.9993 ± 0.0010	0.9999 ± 0.0001	0.9679 ± 0.0413	0.0020 ± 0.0025	0.0032 ± 0.0023	0.0004 ± 0.0004	33.6300 ± 10.8370
0.2	SGD (fBm-W(Burst))	0.9993 ± 0.0011	0.9999 ± 0.0002	0.9658 ± 0.0488	0.0010 ± 0.0007	0.0035 ± 0.0035	0.0009 ± 0.0010	33.3820 ± 11.0377
0.5	Adam (Adaptive-fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0221 ± 0.0006	0.0121 ± 0.0003	50.7140 ± 17.9831
0.5	Adam (Adaptive-fBm-W)	0.9973 ± 0.0035	0.9996 ± 0.0004	0.8614 ± 0.1599	0.0020 ± 0.0033	0.0149 ± 0.0188	0.0012 ± 0.0010	50.8680 ± 17.8633
0.5	Adam (Standard)	0.9931 ± 0.0002	1.0000 ± 0.0000	0.6818 ± 0.0039	0.0055 ± 0.0000	0.0233 ± 0.0051	0.0001 ± 0.0000	56.1420 ± 1.8162
0.5	Adam (fBm)	0.8532 ± 0.3238	0.9982 ± 0.0001	0.4428 ± 0.1268	0.1438 ± 0.3222	0.9840 ± 2.1622	0.0120 ± 0.0015	41.9360 ± 2.6968
0.5	Adam (fBm(Burst))	0.8533 ± 0.3236	0.9983 ± 0.0002	0.4430 ± 0.1264	0.1437 ± 0.3224	1.3277 ± 2.9203	0.0128 ± 0.0047	40.8400 ± 3.3404
0.5	Adam (fBm-W)	0.9946 ± 0.0030	1.0000 ± 0.0000	0.7476 ± 0.1412	0.0044 ± 0.0025	0.0159 ± 0.0094	0.0001 ± 0.0000	41.3720 ± 1.2032
0.5	Adam (fBm-W(Burst))	0.9930 ± 0.0000	1.0000 ± 0.0000	0.6801 ± 0.0000	0.0058 ± 0.0003	0.0236 ± 0.0084	0.0001 ± 0.0000	48.7680 ± 8.9154
0.5	Nadam (Adaptive-fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0197 ± 0.0031	0.0123 ± 0.0008	49.6700 ± 5.3871
0.5	Nadam (Adaptive-fBm-W)	0.9975 ± 0.0032	0.9997 ± 0.0004	0.8651 ± 0.1548	0.0019 ± 0.0031	0.0093 ± 0.0109	0.0010 ± 0.0008	50.0380 ± 8.4134
0.5	Nadam (Standard)	0.9930 ± 0.0000	1.0000 ± 0.0000	0.6801 ± 0.0000	0.0056 ± 0.0002	0.0137 ± 0.0033	0.0001 ± 0.0001	52.8500 ± 2.5965
0.5	Nadam (fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0165 ± 0.0020	0.0123 ± 0.0007	41.7480 ± 3.3553
0.5	Nadam (fBm(Burst))	0.8532 ± 0.3238	0.9982 ± 0.0000	0.4428 ± 0.1268	0.1438 ± 0.3222	0.6187 ± 1.3421	0.0113 ± 0.0015	40.7920 ± 2.9547
0.5	Nadam (fBm-W)	0.9930 ± 0.0000	1.0000 ± 0.0001	0.6801 ± 0.0000	0.0057 ± 0.0002	0.0277 ± 0.0118	0.0001 ± 0.0001	42.7120 ± 3.3409
0.5	Nadam (fBm-W(Burst))	0.9931 ± 0.0002	1.0000 ± 0.0000	0.6818 ± 0.0039	0.0057 ± 0.0003	0.0270 ± 0.0109	0.0001 ± 0.0000	41.0620 ± 1.3258
0.5	RMSprop (Adaptive-fBm)	0.9997 ± 0.0007	0.9994 ± 0.0006	0.9399 ± 0.1343	-0.0001 ± 0.0002	0.0013 ± 0.0025	0.0026 ± 0.0018	48.6280 ± 19.0198
0.5	RMSprop (Adaptive-fBm-W)	0.9962 ± 0.0029	1.0000 ± 0.0001	0.7868 ± 0.1083	0.0033 ± 0.0028	0.0165 ± 0.0235	0.0001 ± 0.0001	50.1980 ± 18.1925
0.5	RMSprop (Standard)	0.9968 ± 0.0035	1.0000 ± 0.0000	0.8297 ± 0.1378	0.0022 ± 0.0030	0.0086 ± 0.0053	0.0000 ± 0.0000	52.7620 ± 2.6693
0.5	RMSprop (fBm)	0.9983 ± 0.0033	0.9999 ± 0.0001	0.9121 ± 0.1371	0.0012 ± 0.0026	0.0059 ± 0.0115	0.0002 ± 0.0002	41.4960 ± 1.5611
0.5	RMSprop (fBm(Burst))	0.9987 ± 0.0010	1.0000 ± 0.0000	0.8873 ± 0.0665	0.0010 ± 0.0017	0.0036 ± 0.0021	0.0001 ± 0.0001	39.9780 ± 1.1096
0.5	RMSprop (fBm-W)	0.9953 ± 0.0030	1.0000 ± 0.0001	0.7647 ± 0.1361	0.0043 ± 0.0024	0.0176 ± 0.0137	0.0002 ± 0.0004	48.1900 ± 19.2892
0.5	RMSprop (fBm-W(Burst))	0.9982 ± 0.0029	0.9999 ± 0.0001	0.8937 ± 0.1246	0.0014 ± 0.0026	0.0069 ± 0.0104	0.0002 ± 0.0001	41.0800 ± 1.4561
0.5	SGD (Adaptive-fBm)	1.0000 ± 0.0000	0.9997 ± 0.0000	1.0000 ± 0.0000	-0.0003 ± 0.0000	0.0031 ± 0.0025	0.0009 ± 0.0004	40.4080 ± 2.2372

Continued on next page

Table A.5: Results for bot-iot dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.5	SGD (Adaptive-fBm-W)	0.9991 ± 0.0010	0.9988 ± 0.0007	0.7855 ± 0.2627	0.0000 ± 0.0004	0.0052 ± 0.0066	0.0069 ± 0.0054	50.1840 ± 18.3388
0.5	SGD (Standard)	1.0000 ± 0.0000	0.9996 ± 0.0001	1.0000 ± 0.0000	-0.0000 ± 0.0006	0.0093 ± 0.0022	0.0009 ± 0.0001	55.6260 ± 1.9485
0.5	SGD (fBm)	0.9999 ± 0.0002	0.9997 ± 0.0001	0.9889 ± 0.0249	-0.0003 ± 0.0001	0.0061 ± 0.0021	0.0008 ± 0.0001	48.2160 ± 19.2668
0.5	SGD (fBm(Burst))	1.0000 ± 0.0000	0.9997 ± 0.0001	1.0000 ± 0.0000	-0.0003 ± 0.0001	0.0099 ± 0.0051	0.0008 ± 0.0001	39.1820 ± 1.0844
0.5	SGD (fBm-W)	1.0000 ± 0.0000	0.9996 ± 0.0001	1.0000 ± 0.0000	0.0000 ± 0.0005	0.0024 ± 0.0010	0.0017 ± 0.0004	47.9380 ± 19.3956
0.5	SGD (fBm-W(Burst))	1.0000 ± 0.0000	0.9997 ± 0.0001	1.0000 ± 0.0000	-0.0002 ± 0.0004	0.0013 ± 0.0006	0.0016 ± 0.0001	39.9240 ± 1.1920
0.7	Adam (Adaptive-fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0201 ± 0.0024	0.0111 ± 0.0010	42.1060 ± 2.4698
0.7	Adam (Adaptive-fBm-W)	1.0000 ± 0.0000	0.9998 ± 0.0000	1.0000 ± 0.0000	-0.0002 ± 0.0000	0.0014 ± 0.0018	0.0008 ± 0.0002	50.0960 ± 18.2711
0.7	Adam (Standard)	0.9931 ± 0.0002	1.0000 ± 0.0001	0.6818 ± 0.0039	0.0055 ± 0.0001	0.0163 ± 0.0038	0.0001 ± 0.0001	57.8080 ± 1.9386
0.7	Adam (fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0152 ± 0.0011	0.0117 ± 0.0002	42.1620 ± 0.7138
0.7	Adam (fBm(Burst))	0.9982 ± 0.0003	0.9983 ± 0.0001	0.5795 ± 0.1096	-0.0002 ± 0.0001	0.0206 ± 0.0008	0.0083 ± 0.0012	40.9460 ± 1.5498
0.7	Adam (fBm-W)	0.9936 ± 0.0013	1.0000 ± 0.0000	0.6938 ± 0.0308	0.0052 ± 0.0007	0.0151 ± 0.0074	0.0001 ± 0.0000	42.0440 ± 1.9550
0.7	Adam (fBm-W(Burst))	0.9931 ± 0.0002	1.0000 ± 0.0000	0.6818 ± 0.0039	0.0056 ± 0.0004	0.0230 ± 0.0123	0.0001 ± 0.0000	42.0560 ± 2.1729
0.7	Nadam (Adaptive-fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0211 ± 0.0024	0.0115 ± 0.0005	41.0860 ± 1.5704
0.7	Nadam (Adaptive-fBm-W)	0.9938 ± 0.0070	0.9999 ± 0.0001	0.7902 ± 0.1949	0.0057 ± 0.0068	0.0284 ± 0.0259	0.0005 ± 0.0004	42.9960 ± 2.2724
0.7	Nadam (Standard)	0.9931 ± 0.0002	0.9999 ± 0.0001	0.6818 ± 0.0039	0.0054 ± 0.0001	0.0291 ± 0.0128	0.0002 ± 0.0001	55.2560 ± 3.1905
0.7	Nadam (fBm)	0.9980 ± 0.0000	0.9982 ± 0.0000	0.4995 ± 0.0000	-0.0003 ± 0.0000	0.0153 ± 0.0012	0.0115 ± 0.0008	43.0680 ± 3.4899
0.7	Nadam (fBm(Burst))	0.9982 ± 0.0003	0.9984 ± 0.0002	0.5795 ± 0.1096	-0.0001 ± 0.0002	0.0151 ± 0.0058	0.0081 ± 0.0017	42.1040 ± 2.2903
0.7	Nadam (fBm-W)	0.9932 ± 0.0003	1.0000 ± 0.0000	0.6836 ± 0.0048	0.0055 ± 0.0000	0.0276 ± 0.0086	0.0001 ± 0.0002	42.0960 ± 2.8047
0.7	Nadam (fBm-W(Burst))	0.9930 ± 0.0000	1.0000 ± 0.0000	0.6801 ± 0.0000	0.0060 ± 0.0003	0.0272 ± 0.0043	0.0001 ± 0.0001	42.4620 ± 1.8504
0.7	RMSprop (Adaptive-fBm)	1.0000 ± 0.0000	0.9998 ± 0.0001	1.0000 ± 0.0000	0.0001 ± 0.0003	0.0001 ± 0.0000	0.0018 ± 0.0004	42.0700 ± 3.8424
0.7	RMSprop (Adaptive-fBm-W)	0.9946 ± 0.0028	1.0000 ± 0.0001	0.7384 ± 0.1154	0.0045 ± 0.0026	0.0192 ± 0.0110	0.0001 ± 0.0002	49.5380 ± 18.5533
0.7	RMSprop (Standard)	0.9949 ± 0.0026	1.0000 ± 0.0000	0.7395 ± 0.0941	0.0041 ± 0.0024	0.0092 ± 0.0026	0.0000 ± 0.0000	52.1180 ± 3.6075
0.7	RMSprop (fBm)	0.9956 ± 0.0032	0.9999 ± 0.0001	0.7756 ± 0.1347	0.0039 ± 0.0024	0.0263 ± 0.0263	0.0005 ± 0.0005	40.9680 ± 2.2375
0.7	RMSprop (fBm(Burst))	0.9972 ± 0.0036	1.0000 ± 0.0000	0.8626 ± 0.1643	0.0023 ± 0.0032	0.0097 ± 0.0126	0.0000 ± 0.0000	41.7520 ± 1.5660

Continued on next page

Table A.5: Results for bot-iot dataset

Hurst Time (s)	Optimizer	Variant	Test ACC	Train ACC	F1-Score	Gap Acc	Test CE	Train CE
0.7	RMSprop (fBm-W)	0.9960 ± 0.0017	1.0000 ± 0.0000	0.7563 ± 0.0437	0.0045 ± 0.0009	0.0086 ± 0.0089	0.0000 ± 0.0001	48.7400 ± 18.9586
0.7	RMSprop (fBm-W(Burst))	0.9967 ± 0.0031	0.9999 ± 0.0001	0.8199 ± 0.1433	0.0032 ± 0.0028	0.0113 ± 0.0139	0.0001 ± 0.0001	48.9840 ± 18.8537
0.7	SGD (Adaptive-fBm)	1.0000 ± 0.0000	0.9998 ± 0.0001	1.0000 ± 0.0000	-0.0002 ± 0.0001	0.0036 ± 0.0020	0.0009 ± 0.0003	40.9000 ± 2.2409
0.7	SGD (Adaptive-fBm-W)	0.9999 ± 0.0002	0.9997 ± 0.0001	0.9889 ± 0.0249	0.0006 ± 0.0003	0.0004 ± 0.0005	0.0024 ± 0.0002	41.5220 ± 3.2526
0.7	SGD (Standard)	1.0000 ± 0.0000	0.9997 ± 0.0000	1.0000 ± 0.0000	-0.0004 ± 0.0000	0.0076 ± 0.0014	0.0008 ± 0.0001	57.4620 ± 2.9024
0.7	SGD (fBm)	1.0000 ± 0.0000	0.9997 ± 0.0000	1.0000 ± 0.0000	-0.0003 ± 0.0000	0.0063 ± 0.0039	0.0010 ± 0.0001	40.8120 ± 1.6980
0.7	SGD (fBm(Burst))	1.0000 ± 0.0000	0.9997 ± 0.0000	1.0000 ± 0.0000	-0.0003 ± 0.0000	0.0102 ± 0.0098	0.0008 ± 0.0000	47.9100 ± 19.4261
0.7	SGD (fBm-W)	1.0000 ± 0.0000	0.9997 ± 0.0000	1.0000 ± 0.0000	-0.0003 ± 0.0000	0.0029 ± 0.0020	0.0010 ± 0.0005	41.0960 ± 1.1908
0.7	SGD (fBm-W(Burst))	1.0000 ± 0.0000	0.9996 ± 0.0000	1.0000 ± 0.0000	-0.0004 ± 0.0000	0.0008 ± 0.0003	0.0015 ± 0.0001	40.7500 ± 2.8748

Appendix B

Code

```
import kagglehub, pandas as pd, numpy as np, tensorflow as tf,
    re, warnings
from pathlib import Path
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import time, math, random, os, numpy as np, tensorflow as tf
import pandas as pd
from fbm import FBM

from sklearn.metrics import (precision_score, recall_score,
    f1_score,
                             roc_auc_score,
    average_precision_score, log_loss)

from tensorflow.keras.utils import to_categorical
import pandas as pd, numpy as np, matplotlib.pyplot as plt,
    tensorflow as tf
from pathlib import Path
```

Listing B.1: libraries

```
SEED          = 222 #change each run
RUNS          = 1
HLIST         = 0.2 #0.2 - 0.5 - 0.7
```

```
np.random.seed(SEED); tf.random.set_seed(SEED); random.seed(
    SEED)
BATCH_SIZE   = 256
EPOCHS       = 50
TOTAL_STEPS  = math.ceil(60000 * 0.8 / BATCH_SIZE) * EPOCHS  #
    48k train
DEV          = "/GPU:0" if tf.config.list_physical_devices("GPU
    ") else "/CPU:0"
```

Listing B.2: hyperparameters

```
def evaluate(model):
    def _pred(ds):
        p, y = [], []
        for xb, yb in ds:
            p.append(model(xb, training=False).numpy())
            y.append(yb.numpy())
        return np.vstack(p), np.vstack(y)

    tr_prob, tr_one = _pred(tr_ds)
    va_prob, va_one = _pred(val_ds)
    te_prob, te_one = _pred(test_ds)

    def _metrics(prob, one):
        lbl  = one.argmax(1)
        pred = prob.argmax(1)
        macro = lambda f: f(lbl, pred, average="macro")
        return dict(
            Prec = macro(precision_score),
            Rec  = macro(recall_score),
            F1   = macro(f1_score),
```

```
            ROC  = roc_auc_score(one, prob, multi_class="ovr",
    average="macro"),
            PR   = average_precision_score(one, prob, average="
    macro"),
            CE   = log_loss(one, prob),
            Acc  = (pred == lbl).mean(),
        )
    tr = _metrics(tr_prob, tr_one)
    va = _metrics(va_prob, va_one)
    te = _metrics(te_prob, te_one)

    # generalisation gaps
    gap_acc = tr["Acc"] - va["Acc"]
    gap_ce  = va["CE"] - tr["CE"]

    prefix = lambda d, p: {f"{p}_{k}": v for k, v in d.items()}
    return {
        **prefix(tr, "Train"),
        **prefix(va, "Val"),
        **prefix(te, "Test"),
        "Gap_Acc": gap_acc,
        "Gap_CE" : gap_ce,
    }

def run_plain_sgd(modelin, lr: float = 0.01, momentum: float =
    0.9):
    tf.keras.backend.clear_session()
    model      = modelin
    optimizer  = tf.keras.optimizers.SGD(learning_rate=lr,
    momentum=momentum)
```

```
    loss_fn    = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def train_step(xb, yb):
        with tf.GradientTape() as tape:
            logits = model(xb, training=True)
            loss   = loss_fn(yb, logits)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.
    trainable_variables))

    # Warm up
    xb, yb = next(iter(tr_ds))
    train_step(xb, yb)

    t0 = time.time()
    for _ in range(EPOCHS):
        for xb, yb in tr_ds:
            train_step(xb, yb)
    runtime = time.time() - t0

    metrics = evaluate(model)
    return {
        "Optimizer"      : "SGD",
        "Time (s)"       : round(runtime, 2),
        "Weight Updates" : TOTAL_STEPS,
        **{k.replace("_", " "): round(v, 6) for k, v in metrics
    .items()}
    }

```

```
75
76
77 def run_plain_adam(modelin, lr: float = 1e-3):
78     """Baseline Adam (no noise) compiled with XLA."""
79     tf.keras.backend.clear_session()
80     model      = modelin
81     optimizer  = tf.keras.optimizers.Adam(learning_rate=lr)
82     loss_fn    = tf.keras.losses.CategoricalCrossentropy()
83
84     @tf.function(jit_compile=True)
85     def train_step(xb, yb):
86         with tf.GradientTape() as tape:
87             logits = model(xb, training=True)
88             loss   = loss_fn(yb, logits)
89         grads = tape.gradient(loss, model.trainable_variables)
90         optimizer.apply_gradients(zip(grads, model.
    trainable_variables))
91
92     xb, yb = next(iter(tr_ds))
93     train_step(xb, yb)
94
95     t0 = time.time()
96     for _ in range(EPOCHS):
97         for xb, yb in tr_ds:
98             train_step(xb, yb)
99     runtime = time.time() - t0
100
101    metrics = evaluate(model)
102    return {
103        "Optimizer"      : "Adam",
```

```
        "Time (s)"       : round(runtime, 2),
        "Weight Updates" : TOTAL_STEPS,
        **{k.replace("_", " "): round(v, 6) for k, v in metrics.items()}
    }


def run_plain_nadam(modelin, lr: float = 1e-3):
    """Baseline Nadam (no noise) compiled with XLA."""
    tf.keras.backend.clear_session()
    model      = modelin
    optimizer  = tf.keras.optimizers.Nadam(learning_rate=lr)
    loss_fn    = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def train_step(xb, yb):
        with tf.GradientTape() as tape:
            logits = model(xb, training=True)
            loss   = loss_fn(yb, logits)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

    xb, yb = next(iter(tr_ds))
    train_step(xb, yb)

    t0 = time.time()
    for _ in range(EPOCHS):
        for xb, yb in tr_ds:
            train_step(xb, yb)
```

```
132     runtime = time.time() - t0
133 
134     metrics = evaluate(model)
135     return {
136         "Optimizer"      : "Nadam",
137         "Time (s)"       : round(runtime, 2),
138         "Weight Updates" : TOTAL_STEPS,
139         **{k.replace("_", " "): round(v, 6) for k, v in metrics
    .items()}
140     }
141 
142 
143 def run_plain_rmsprop(modelin, lr: float = 1e-3, rho: float =
    0.9,
144                       momentum: float = 0.0):
145     """Baseline RMSprop (no noise) compiled with XLA."""
146     tf.keras.backend.clear_session()
147     model      = modelin
148     optimizer  = tf.keras.optimizers.RMSprop(learning_rate=lr,
149                                              rho=rho, momentum=
    momentum)
150     loss_fn    = tf.keras.losses.CategoricalCrossentropy()
151 
152     @tf.function(jit_compile=True)
153     def train_step(xb, yb):
154         with tf.GradientTape() as tape:
155             logits = model(xb, training=True)
156             loss   = loss_fn(yb, logits)
157         grads = tape.gradient(loss, model.trainable_variables)
158         optimizer.apply_gradients(zip(grads, model.
```

```
    trainable_variables))

    xb, yb = next(iter(tr_ds))
    train_step(xb, yb)

    t0 = time.time()
    for _ in range(EPOCHS):
        for xb, yb in tr_ds:
            train_step(xb, yb)
    runtime = time.time() - t0

    metrics = evaluate(model)
    return {
        "Optimizer"      : "RMSprop",
        "Time (s)"       : round(runtime, 2),
        "Weight Updates" : TOTAL_STEPS,
        **{k.replace("_", " "): round(v, 6) for k, v in metrics
    .items()}
    }

def fbm_path(hurst, sigma, dev="/GPU:0"):
    fbm = FBM(n=TOTAL_STEPS, hurst=hurst, length=1.0,
              method="daviesharte").fbm()
    with tf.device(dev):
        return tf.constant(sigma * np.diff(fbm, prepend=0)
                           .astype("float32"))
def run_sgd_fbm(modelin, eta=0.01, mu=0.9, hurst=0.7, sigma
    =0.01):
    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)
```

```
    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, noise):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)
        for g, m, w in zip(grads, m_buf, model.
    trainable_variables):
            m.assign(mu * m + g + noise)
            w.assign_sub(eta * m)

    #warm-up
    xb, yb = next(iter(tr_ds))
    step(xb, yb, xi[0])

    #timed
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    metrics = evaluate(model)
    return {
        "Optimizer": "SGD+fBm", "Time (s)": round(runtime, 2),
        "Weight Updates": TOTAL_STEPS, **{k.replace('_', ' '):
    round(v, 6)
```

```
        for k, v in metrics.items()}
    }


def run_adam_fbm_fast(modelin, eta=1e-3, beta1=0.9, beta2=0.999,
                      eps=1e-8, hurst=0.7, sigma=0.01):
    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)
    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, step_idx, noise):
        t = tf.cast(step_idx + 1, tf.float32)
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, m, v, w in zip(grads, m_buf, v_buf, model.trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g + noise)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))

            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
```

```
            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))

    # warm-up
    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi[0])

    #timed
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, tf.constant(i), xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Adam+fBm", "Time (s)": round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met.items()}}
def run_nadam_fbm_fast(modelin, eta=1e-3, beta1=0.9, beta2=0.999,
                       eps=1e-8, hurst=0.7, sigma=0.01):
    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)
    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

```

```
    @tf.function(jit_compile=True)
    def step(xb, yb, step_idx, noise):
        t = tf.cast(step_idx + 1, tf.float32)
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g + noise)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))

            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
            g_hat = g / (1. - tf.pow(beta1, t))
            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
    ))

    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
    [0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, tf.constant(i), xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+fBm", "Time (s)": round(runtime
    , 2),
```

```
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}

def run_rmsprop_fbm_fast(modelin, eta=1e-3, rho=0.9, mu=0.9,
                         eps=1e-8, hurst=0.7, sigma=0.01):
    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)
    model = modelin
    r_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, noise):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, r, v, w in zip(grads, r_buf, v_buf,
                              model.trainable_variables):
            r.assign(rho * r + (1. - rho) * tf.square(g))
            rms_step = eta * g / (tf.sqrt(r) + eps)
            v.assign(mu * v + rms_step + noise)
            w.assign_sub(v)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])

```

```
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "RMSprop+fBm", "Time (s)": round(
    runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}
def run_sgd_fbm_weight_noise(modelin,
        eta   = 0.01,
        mu    = 0.9,
        hurst = 0.7,

        sigma = 0.01 ):

    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)

    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(w), trainable=False)
             for w in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, noise):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
```

```
351         grads = tape.gradient(loss, model.trainable_variables)
352 
353         for g, m, w in zip(grads, m_buf, model.
    trainable_variables):
354             m.assign(mu * m + g)
355             w.assign_sub(eta * m)
356             w.assign_add(noise)
357 
358     xb, yb = next(iter(tr_ds))
359     step(xb, yb, xi[0])
360 
361     t0 = time.time()
362     for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
363         step(xb, yb, xi[i])
364     runtime = time.time() - t0
365 
366     met = evaluate(model)
367     return {
368         "Optimizer"      : "SGD+fBm-W",
369         "Time (s)"       : round(runtime, 2),
370         "Weight Updates" : TOTAL_STEPS,
371         **{k.replace('_', ' '): round(v, 6) for k, v in met.
    items()}
372     }
373 
374 def run_adam_fbm_weight_noise(modelin,
375         eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
376         hurst=0.7, sigma=0.01):
377     tf.keras.backend.clear_session()
378 
```

```
    xi = fbm_path(hurst, sigma)
    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, step_idx, noise):
        t = tf.cast(step_idx + 1, tf.float32)
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))

            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
    [0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
```

```
407         step(xb, yb, tf.constant(i), xi[i])
408     runtime = time.time() - t0
409 
410     met = evaluate(model)
411     return {"Optimizer": "Adam+fBm-W", "Time (s)": round(
    runtime, 2),
412             "Weight Updates": TOTAL_STEPS,
413             **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}
414 def run_nadam_fbm_weight_noise(modelin,
415         eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
416         hurst=0.7, sigma=0.01):
417     tf.keras.backend.clear_session()
418     xi = fbm_path(hurst, sigma)
419     model = modelin
420     m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
421              for v in model.trainable_variables]
422     v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
423              for v in model.trainable_variables]
424     loss_fn = tf.keras.losses.CategoricalCrossentropy()
425 
426     @tf.function(jit_compile=True)
427     def step(xb, yb, step_idx, noise):
428         t = tf.cast(step_idx + 1, tf.float32)
429         with tf.GradientTape() as tape:
430             loss = loss_fn(yb, model(xb, training=True))
431         grads = tape.gradient(loss, model.trainable_variables)
432 
433         for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
```

```
            m.assign(beta1 * m + (1. - beta1) * g)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))

            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
            g_hat = g / (1. - tf.pow(beta1, t))
            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
    ))
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
    [0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, tf.constant(i), xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+fBm-W", "Time (s)": round(
    runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}
def run_rmsprop_fbm_weight_noise(modelin,
        eta=1e-3, rho=0.9, mu=0.9, eps=1e-8,
        hurst=0.7, sigma=0.01):
    tf.keras.backend.clear_session()
    xi = fbm_path(hurst, sigma)
```

```
    model = modelin
    r_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
             for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, noise):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, r, v, w in zip(grads, r_buf, v_buf, model.
trainable_variables):
            r.assign(rho * r + (1. - rho) * tf.square(g))
            rms_step = eta * g / (tf.sqrt(r) + eps)
            v.assign(mu * v + rms_step)
            w.assign_sub(v)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "RMSprop+fBm-W", "Time (s)": round(
```

```
    runtime, 2),
489             "Weight Updates": TOTAL_STEPS,
490             **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}
491 def run_sgd_adaptive_fbm(modelin,
492         eta      = 0.01,
493         mu       = 0.9,
494         sigma0   = 0.01,
495         g_ref    = 0.1,
496         sigma_min= 1e-4,
497         sigma_max= 0.1,
498         hurst    = 0.7):
499
500     tf.keras.backend.clear_session()
501     xi_base = fbm_path(hurst, 1.0, dev=DEV)
502     eps_norm = 1e-8
503
504     model = modelin
505     m_buf = [tf.Variable(tf.zeros_like(v), trainable=False)
506              for v in model.trainable_variables]
507     loss_fn = tf.keras.losses.CategoricalCrossentropy()
508
509     @tf.function(jit_compile=True)
510     def step(xb, yb, step_idx):
511         with tf.GradientTape() as tape:
512             loss = loss_fn(yb, model(xb, training=True))
513         grads = tape.gradient(loss, model.trainable_variables)
514
515         g_norm = tf.linalg.global_norm(grads)
516
```

```
517         sigma_t = tf.clip_by_value(
518             sigma0 * g_ref / (g_norm + eps_norm), sigma_min,
    sigma_max)
519 
520         noise = sigma_t * step_idx
521 
522         for g, m, w in zip(grads, m_buf, model.
    trainable_variables):
523             m.assign(mu * m + g + noise)
524             w.assign_sub(eta * m)
525 
526     xb, yb = next(iter(tr_ds))
527     step(xb, yb, xi_base[0])
528 
529     t0 = time.time()
530     for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
531         step(xb, yb, xi_base[i])
532     runtime = time.time() - t0
533 
534     met = evaluate(model)
535     return {
536         "Optimizer"      : "SGD+Adaptive-fBm",
537         "Time (s)"       : round(runtime, 2),
538         "Weight Updates" : TOTAL_STEPS,
539         **{k.replace('_', ' '): round(v, 6) for k, v in met.
    items()}
540     }
541 def run_adam_adaptive_fbm(modelin, eta=1e-3, beta1=0.9, beta2
    =0.999, eps=1e-8,
542                           sigma0=0.01, g_ref=0.1,
```

```
                        sigma_min=1e-4, sigma_max=0.1,
                        hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
               for v in model.trainable_variables]
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
               for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()
    eps_norm = 1e-8

    @tf.function(jit_compile=True)
    def step(xb, yb, step_idx):
        t = tf.cast(step_idx + 1, tf.float32)
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)
        g_norm = tf.linalg.global_norm(grads)

        sigma_t = tf.clip_by_value(sigma0 * g_ref / (g_norm +
eps_norm),
                                   sigma_min, sigma_max)
        noise   = sigma_t * step_idx

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g + noise)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))

```

```
            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))

    xb, yb = next(iter(tr_ds))
    step(xb, yb, xi_base[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Adam+Adaptive-fBm", "Time (s)": round
    (runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}

def run_nadam_adaptive_fbm(modelin, eta=1e-3, beta1=0.9, beta2
    =0.999, eps=1e-8,
                           sigma0=0.01, g_ref=0.1,
                           sigma_min=1e-4, sigma_max=0.1,
                           hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
               for v in model.trainable_variables]
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
```

```
            for v in model.trainable_variables]
loss_fn = tf.keras.losses.CategoricalCrossentropy()
eps_norm = 1e-8

@tf.function(jit_compile=True)
def step(xb, yb, step_idx):
    t = tf.cast(step_idx + 1, tf.float32)
    with tf.GradientTape() as tape:
        loss = loss_fn(yb, model(xb, training=True))
    grads = tape.gradient(loss, model.trainable_variables)
    g_norm = tf.linalg.global_norm(grads)

    sigma_t = tf.clip_by_value(sigma0 * g_ref / (g_norm +
eps_norm),
                               sigma_min, sigma_max)
    noise   = sigma_t * step_idx

    for g, m, v, w in zip(grads, m_buf, v_buf, model.
trainable_variables):
        m.assign(beta1 * m + (1. - beta1) * g + noise)
        v.assign(beta2 * v + (1. - beta2) * tf.square(g))

        m_hat = m / (1. - tf.pow(beta1, t))
        v_hat = v / (1. - tf.pow(beta2, t))
        g_hat = g / (1. - tf.pow(beta1, t))
        nesterov = beta1 * m_hat + (1. - beta1) * g_hat
        w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
))

xb, yb = next(iter(tr_ds))
```

```
    step(xb, yb, xi_base[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+Adaptive-fBm", "Time (s)":
    round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}
def run_rmsprop_adaptive_fbm(modelin, eta=1e-3, rho=0.9, mu
    =0.9, eps=1e-8,
                             sigma0=0.01, g_ref=0.1,
                             sigma_min=1e-4, sigma_max=0.1,
                             hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    r_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
               for v in model.trainable_variables]
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
               for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()
    eps_norm = 1e-8

    @tf.function(jit_compile=True)
    def step(xb, yb, step_idx):
```

```
652         with tf.GradientTape() as tape:
653             loss = loss_fn(yb, model(xb, training=True))
654         grads  = tape.gradient(loss, model.trainable_variables)
655         g_norm = tf.linalg.global_norm(grads)
656 
657         sigma_t = tf.clip_by_value(sigma0 * g_ref / (g_norm +
    eps_norm),
658                                    sigma_min, sigma_max)
659         noise   = sigma_t * step_idx
660 
661         for g, r, v, w in zip(grads, r_buf, v_buf, model.
    trainable_variables):
662             r.assign(rho * r + (1. - rho) * tf.square(g))
663             rms_step = eta * g / (tf.sqrt(r) + eps)
664             v.assign(mu * v + rms_step + noise)
665             w.assign_sub(v)
666 
667     xb, yb = next(iter(tr_ds))
668     step(xb, yb, xi_base[0])
669 
670     t0 = time.time()
671     for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
672         step(xb, yb, xi_base[i])
673     runtime = time.time() - t0
674 
675     met = evaluate(model)
676     return {"Optimizer": "RMSprop+Adaptive-fBm", "Time (s)":
    round(runtime, 2),
677             "Weight Updates": TOTAL_STEPS,
678             **{k.replace('_', ' '): round(v, 4) for k, v in met
```

```
    .items()}}

def run_sgd_adaptive_fbm_weight_noise(modelin,
        eta       = 0.01,
        mu        = 0.9,
        sigma0    = 0.01,
        g_ref     = 0.1,
        sigma_min = 1e-4,
        sigma_max = 0.1,
        hurst     = 0.7):
    tf.keras.backend.clear_session()

    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    eps_norm = 1e-8


    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(w), trainable=False)
             for w in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, xi_t):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads  = tape.gradient(loss, model.trainable_variables)

        g_norm  = tf.linalg.global_norm(grads)
        sigma_t = tf.clip_by_value(
            sigma0 * g_ref / (g_norm + eps_norm),
```

```
            sigma_min, sigma_max)

        noise = sigma_t * xi_t
        for g, m, w in zip(grads, m_buf, model.trainable_variables):
            m.assign(mu * m + g)
            w.assign_sub(eta * m)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds))
    step(xb, yb, xi_base[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i])
    runtime = time.time() - t0


    met = evaluate(model)
    return {
        "Optimizer"      : "SGD+Adaptive-fBm-W",
        "Time (s)"       : round(runtime, 2),
        "Weight Updates" : TOTAL_STEPS,
        **{k.replace('_', ' '): round(v, 4) for k, v in met.items()}
    }
def _adaptive_sigma(g_norm, sigma0, g_ref, sigma_min, sigma_max):
    sigma = sigma0 * g_ref / (g_norm + 1e-8)
    return tf.clip_by_value(sigma, sigma_min, sigma_max)
```

```
735
736 def run_adam_adaptive_fbm_weight_noise(modelin,
737         eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
738         sigma0=0.01, g_ref=0.1,
739         sigma_min=1e-4, sigma_max=0.1,
740         hurst=0.7):
741     tf.keras.backend.clear_session()
742     xi_base = fbm_path(hurst, 1.0, dev=DEV)
743     model   = modelin
744     m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
745     v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
746     loss_fn = tf.keras.losses.CategoricalCrossentropy()
747
748     @tf.function(jit_compile=True)
749     def step(xb, yb, xi_t, t_float):
750         with tf.GradientTape() as tape:
751             loss = loss_fn(yb, model(xb, training=True))
752         grads  = tape.gradient(loss, model.trainable_variables)
753         g_norm = tf.linalg.global_norm(grads)
754
755         sigma_t = _adaptive_sigma(g_norm, sigma0, g_ref,
    sigma_min, sigma_max)
756         noise   = sigma_t * xi_t
757
758         for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
759             m.assign(beta1 * m + (1. - beta1) * g)
760             v.assign(beta2 * v + (1. - beta2) * tf.square(g))
```

```
            m_hat = m / (1. - tf.pow(beta1, t_float))
            v_hat = v / (1. - tf.pow(beta2, t_float))
            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds))
    step(xb, yb, xi_base[0], tf.constant(1.0))

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i], tf.cast(i + 1, tf.float32))
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Adam+Adaptive-fBm-W", "Time (s)":
    round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}

def run_nadam_adaptive_fbm_weight_noise(modelin,
        eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
        sigma0=0.01, g_ref=0.1,
        sigma_min=1e-4, sigma_max=0.1,
        hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
```

```
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, xi_t, t_float):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads  = tape.gradient(loss, model.trainable_variables)
        g_norm = tf.linalg.global_norm(grads)

        sigma_t = _adaptive_sigma(g_norm, sigma0, g_ref,
sigma_min, sigma_max)
        noise   = sigma_t * xi_t

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
            m_hat = m / (1. - tf.pow(beta1, t_float))
            v_hat = v / (1. - tf.pow(beta2, t_float))
            g_hat = g / (1. - tf.pow(beta1, t_float))
            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
))
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi_base[0], tf.
constant(1.0))
    t0 = time.time()
```

```
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i], tf.cast(i + 1, tf.float32))
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+Adaptive-fBm-W", "Time (s)":
    round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}
def run_sgd_adaptive_fbm_weight_noise(modelin,
        eta       = 0.01,
        mu        = 0.9,
        sigma0    = 0.01,
        g_ref     = 0.1,
        sigma_min = 1e-4,
        sigma_max = 0.1,
        hurst     = 0.7):
    tf.keras.backend.clear_session()

    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    eps_norm = 1e-8


    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(w), trainable=False)
             for w in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
```

```
    def step(xb, yb, xi_t):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads  = tape.gradient(loss, model.trainable_variables)

        g_norm  = tf.linalg.global_norm(grads)
        sigma_t = tf.clip_by_value(
            sigma0 * g_ref / (g_norm + eps_norm),
            sigma_min, sigma_max)

        noise = sigma_t * xi_t
        for g, m, w in zip(grads, m_buf, model.
    trainable_variables):
            m.assign(mu * m + g)
            w.assign_sub(eta * m)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds))
    step(xb, yb, xi_base[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i])
    runtime = time.time() - t0


    met = evaluate(model)
    return {
        "Optimizer"      : "SGD+Adaptive-fBm-W",
        "Time (s)"       : round(runtime, 2),
```

```
870         "Weight Updates" : TOTAL_STEPS,
871         **{k.replace('_', ' '): round(v, 4) for k, v in met.
    items()}
872     }
873 def _adaptive_sigma(g_norm, sigma0, g_ref, sigma_min, sigma_max
    ):
874     sigma = sigma0 * g_ref / (g_norm + 1e-8)
875     return tf.clip_by_value(sigma, sigma_min, sigma_max)
876 
877 def run_adam_adaptive_fbm_weight_noise(modelin,
878         eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
879         sigma0=0.01, g_ref=0.1,
880         sigma_min=1e-4, sigma_max=0.1,
881         hurst=0.7):
882     tf.keras.backend.clear_session()
883     xi_base = fbm_path(hurst, 1.0, dev=DEV)
884     model   = modelin
885     m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
886     v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
887     loss_fn = tf.keras.losses.CategoricalCrossentropy()
888 
889     @tf.function(jit_compile=True)
890     def step(xb, yb, xi_t, t_float):
891         with tf.GradientTape() as tape:
892             loss = loss_fn(yb, model(xb, training=True))
893         grads  = tape.gradient(loss, model.trainable_variables)
894         g_norm = tf.linalg.global_norm(grads)
895 
```

```
896         sigma_t = _adaptive_sigma(g_norm, sigma0, g_ref,
    sigma_min, sigma_max)
897         noise   = sigma_t * xi_t
898 
899         for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
900             m.assign(beta1 * m + (1. - beta1) * g)
901             v.assign(beta2 * v + (1. - beta2) * tf.square(g))
902             m_hat = m / (1. - tf.pow(beta1, t_float))
903             v_hat = v / (1. - tf.pow(beta2, t_float))
904             w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))
905             w.assign_add(noise)
906 
907     xb, yb = next(iter(tr_ds))
908     step(xb, yb, xi_base[0], tf.constant(1.0))
909 
910     t0 = time.time()
911     for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
912         step(xb, yb, xi_base[i], tf.cast(i + 1, tf.float32))
913     runtime = time.time() - t0
914 
915     met = evaluate(model)
916     return {"Optimizer": "Adam+Adaptive-fBm-W", "Time (s)":
    round(runtime, 2),
917             "Weight Updates": TOTAL_STEPS,
918             **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}
919 
920 def run_nadam_adaptive_fbm_weight_noise(modelin,
921         eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
```

```
        sigma0=0.01, g_ref=0.1,
        sigma_min=1e-4, sigma_max=0.1,
        hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    m_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
for v in model.trainable_variables]
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, xi_t, t_float):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads  = tape.gradient(loss, model.trainable_variables)
        g_norm = tf.linalg.global_norm(grads)

        sigma_t = _adaptive_sigma(g_norm, sigma0, g_ref,
sigma_min, sigma_max)
        noise   = sigma_t * xi_t

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g)
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
            m_hat = m / (1. - tf.pow(beta1, t_float))
            v_hat = v / (1. - tf.pow(beta2, t_float))
            g_hat = g / (1. - tf.pow(beta1, t_float))
```

```
            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
    ))
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi_base[0], tf.
    constant(1.0))
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i], tf.cast(i + 1, tf.float32))
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+Adaptive-fBm-W", "Time (s)":
    round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}

def run_rmsprop_adaptive_fbm_weight_noise(modelin,
        eta=1e-3, rho=0.9, mu=0.9, eps=1e-8,
        sigma0=0.01, g_ref=0.1,
        sigma_min=1e-4, sigma_max=0.1,
        hurst=0.7):
    tf.keras.backend.clear_session()
    xi_base = fbm_path(hurst, 1.0, dev=DEV)
    model   = modelin
    r_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
    for v in model.trainable_variables]
    v_buf   = [tf.Variable(tf.zeros_like(v), trainable=False)
```

```
    for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, xi_t):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads  = tape.gradient(loss, model.trainable_variables)
        g_norm = tf.linalg.global_norm(grads)

        sigma_t = _adaptive_sigma(g_norm, sigma0, g_ref,
    sigma_min, sigma_max)
        noise   = sigma_t * xi_t

        for g, r, v, w in zip(grads, r_buf, v_buf, model.
    trainable_variables):
            r.assign(rho * r + (1. - rho) * tf.square(g))
            rms_step = eta * g / (tf.sqrt(r) + eps)
            v.assign(mu * v + rms_step)
            w.assign_sub(v)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi_base[0])
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi_base[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "RMSprop+Adaptive-fBm-W", "Time (s)":
```

```
    round(runtime, 2),
1000            "Weight Updates": TOTAL_STEPS,
1001            **{k.replace('_', ' '): round(v, 4) for k, v in met
    .items()}}
1002
1003 def _burst_mask(total_steps, burst_prob=0.05, num_bursts=None,
    burst_len=32, seed=SEED):
1004     rng = np.random.default_rng(seed)
1005     mask = np.zeros(int(total_steps), dtype=np.float32)
1006
1007     if num_bursts is not None and burst_len and burst_len > 0:
1008         burst_len = int(burst_len)
1009         num_bursts = int(num_bursts)
1010         for _ in range(num_bursts):
1011             start = int(rng.integers(0, max(1, total_steps -
    burst_len)))
1012             mask[start:start + burst_len] = 1.0
1013     else:
1014         mask[rng.random(int(total_steps)) < float(burst_prob)]
    = 1.0
1015
1016     return tf.constant(mask, dtype=tf.float32)
1017
1018 def _make_burst_xi(hurst, sigma, burst_prob=0.05, num_bursts=
    None, burst_len=32,
1019                    burst_scale=1.0, seed=SEED):
1020
1021     xi_full = fbm_path(hurst, sigma)
1022     xi      = tf.convert_to_tensor(xi_full)[:TOTAL_STEPS]
1023     mask    = _burst_mask(TOTAL_STEPS, burst_prob, num_bursts,
```

```
    burst_len, seed)
1024    return tf.cast(xi, tf.float32) * (tf.cast(burst_scale, tf.
    float32) * mask)
1025
1026def run_sgd_brust_fbm(modelin, eta=0.01, mu=0.9,
1027                      hurst=0.7, sigma=0.01,
1028                      burst_prob=0.05, num_bursts=None,
    burst_len=32,
1029                      burst_scale=1.0, seed=SEED):
1030
1031    tf.keras.backend.clear_session()
1032    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
    burst_len, burst_scale, seed)
1033
1034    model = modelin
1035    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
     v in model.trainable_variables]
1036    loss_fn = tf.keras.losses.CategoricalCrossentropy()
1037
1038    @tf.function(jit_compile=True)
1039    def step(xb, yb, noise):
1040        with tf.GradientTape() as tape:
1041            loss = loss_fn(yb, model(xb, training=True))
1042        grads = tape.gradient(loss, model.trainable_variables)
1043        for g, m, w in zip(grads, m_buf, model.
    trainable_variables):
1044            m.assign(mu * m + g + noise)
1045            w.assign_sub(eta * m)
1046
1047    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])
```

```
1048
1049    t0 = time.time()
1050    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
1051        step(xb, yb, xi[i])
1052    runtime = time.time() - t0
1053
1054    met = evaluate(model)
1055    return {"Optimizer": "SGD+fBm(Burst)", "Time (s)": round(
        runtime, 2),
1056            "Weight Updates": TOTAL_STEPS,
1057            **{k.replace('_', ' '): round(v, 6) for k, v in met
        .items()}}
1058
1059def run_sgd_fbm_brust_weight_noise(modelin, eta=0.01, mu=0.9,
1060                                   hurst=0.7, sigma=0.01,
1061                                   burst_prob=0.05, num_bursts=
        None, burst_len=32,
1062                                   burst_scale=1.0, seed=SEED):
1063
1064    tf.keras.backend.clear_session()
1065    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
        burst_len, burst_scale, seed)
1066
1067    model = modelin
1068    m_buf = [tf.Variable(tf.zeros_like(w), trainable=False) for
         w in model.trainable_variables]
1069    loss_fn = tf.keras.losses.CategoricalCrossentropy()
1070
1071    @tf.function(jit_compile=True)
1072    def step(xb, yb, noise):
```

```
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, m, w in zip(grads, m_buf, model.trainable_variables):
            m.assign(mu * m + g)
            w.assign_sub(eta * m)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "SGD+fBm-W(Burst)", "Time (s)": round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met.items()}}

def run_adam_brust_fbm_fast(modelin, eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
                            hurst=0.7, sigma=0.01,
                            burst_prob=0.05, num_bursts=None, burst_len=32,
                            burst_scale=1.0, seed=SEED):
```

```
1098
1099    tf.keras.backend.clear_session()
1100    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
        burst_len, burst_scale, seed)
1101
1102    model = modelin
1103    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
         v in model.trainable_variables]
1104    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
         v in model.trainable_variables]
1105    loss_fn = tf.keras.losses.CategoricalCrossentropy()
1106
1107    @tf.function(jit_compile=True)
1108    def step(xb, yb, t_int, noise):
1109        t = tf.cast(t_int + 1, tf.float32)
1110        with tf.GradientTape() as tape:
1111            loss = loss_fn(yb, model(xb, training=True))
1112        grads = tape.gradient(loss, model.trainable_variables)
1113
1114        for g, m, v, w in zip(grads, m_buf, v_buf, model.
        trainable_variables):
1115            m.assign(beta1 * m + (1. - beta1) * g + noise)
1116            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
1117            m_hat = m / (1. - tf.pow(beta1, t))
1118            v_hat = v / (1. - tf.pow(beta2, t))
1119            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))
1120
1121    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
        [0])
1122
```

```
    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, tf.constant(i), xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Adam+fBm(Burst)", "Time (s)": round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met.items()}}

def run_adam_fbm_brust_weight_noise(modelin, eta=1e-3, beta1=0.9, beta2=0.999, eps=1e-8,
                                    hurst=0.7, sigma=0.01,
                                    burst_prob=0.05, num_bursts=None, burst_len=32,
                                    burst_scale=1.0, seed=SEED):

    tf.keras.backend.clear_session()
    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts, burst_len, burst_scale, seed)

    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()
```

```
1145
1146    @tf.function(jit_compile=True)
1147    def step(xb, yb, t_int, noise):
1148        t = tf.cast(t_int + 1, tf.float32)
1149        with tf.GradientTape() as tape:
1150            loss = loss_fn(yb, model(xb, training=True))
1151        grads = tape.gradient(loss, model.trainable_variables)
1152
1153        for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
1154            m.assign(beta1 * m + (1. - beta1) * g)
1155            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
1156            m_hat = m / (1. - tf.pow(beta1, t))
1157            v_hat = v / (1. - tf.pow(beta2, t))
1158            w.assign_sub(eta * m_hat / (tf.sqrt(v_hat) + eps))
1159            w.assign_add(noise)
1160
1161    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
    [0])
1162
1163    t0 = time.time()
1164    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
1165        step(xb, yb, tf.constant(i), xi[i])
1166    runtime = time.time() - t0
1167
1168    met = evaluate(model)
1169    return {"Optimizer": "Adam+fBm-W(Burst)", "Time (s)": round
    (runtime, 2),
1170            "Weight Updates": TOTAL_STEPS,
1171            **{k.replace('_', ' '): round(v, 6) for k, v in met
```

```
    .items()}}

def run_nadam_brust_fbm_fast(modelin, eta=1e-3, beta1=0.9,
    beta2=0.999, eps=1e-8,
                             hurst=0.7, sigma=0.01,
                             burst_prob=0.05, num_bursts=None,
    burst_len=32,
                             burst_scale=1.0, seed=SEED):

    tf.keras.backend.clear_session()
    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
    burst_len, burst_scale, seed)

    model = modelin
    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
     v in model.trainable_variables]
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
     v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, t_int, noise):
        t = tf.cast(t_int + 1, tf.float32)
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, m, v, w in zip(grads, m_buf, v_buf, model.
    trainable_variables):
            m.assign(beta1 * m + (1. - beta1) * g + noise)
```

```
            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
            m_hat = m / (1. - tf.pow(beta1, t))
            v_hat = v / (1. - tf.pow(beta2, t))
            g_hat = g / (1. - tf.pow(beta1, t))
            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
    ))

    xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
    [0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, tf.constant(i), xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "Nadam+fBm(Burst)", "Time (s)": round(
    runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}

def run_nadam_fbm_brust_weight_noise(modelin, eta=1e-3, beta1
    =0.9, beta2=0.999, eps=1e-8,
                                         hurst=0.7, sigma=0.01,
                                         burst_prob=0.05,
    num_bursts=None, burst_len=32,
                                         burst_scale=1.0, seed=SEED
    ):
```

```
1218
1219    tf.keras.backend.clear_session()
1220    xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
        burst_len, burst_scale, seed)
1221
1222    model = modelin
1223    m_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
         v in model.trainable_variables]
1224    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
         v in model.trainable_variables]
1225    loss_fn = tf.keras.losses.CategoricalCrossentropy()
1226
1227    @tf.function(jit_compile=True)
1228    def step(xb, yb, t_int, noise):
1229        t = tf.cast(t_int + 1, tf.float32)
1230        with tf.GradientTape() as tape:
1231            loss = loss_fn(yb, model(xb, training=True))
1232        grads = tape.gradient(loss, model.trainable_variables)
1233
1234        for g, m, v, w in zip(grads, m_buf, v_buf, model.
        trainable_variables):
1235            m.assign(beta1 * m + (1. - beta1) * g)
1236            v.assign(beta2 * v + (1. - beta2) * tf.square(g))
1237            m_hat = m / (1. - tf.pow(beta1, t))
1238            v_hat = v / (1. - tf.pow(beta2, t))
1239            g_hat = g / (1. - tf.pow(beta1, t))
1240            nesterov = beta1 * m_hat + (1. - beta1) * g_hat
1241            w.assign_sub(eta * nesterov / (tf.sqrt(v_hat) + eps
        ))
1242            w.assign_add(noise)
```

```
1243
1244     xb, yb = next(iter(tr_ds)); step(xb, yb, tf.constant(0), xi
         [0])
1245
1246     t0 = time.time()
1247     for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
1248         step(xb, yb, tf.constant(i), xi[i])
1249     runtime = time.time() - t0
1250
1251     met = evaluate(model)
1252     return {"Optimizer": "Nadam+fBm-W(Burst)", "Time (s)":
         round(runtime, 2),
1253             "Weight Updates": TOTAL_STEPS,
1254             **{k.replace('_', ' '): round(v, 6) for k, v in met
         .items()}}
1255
1256 def run_rmsprop_brust_fbm_fast(modelin, eta=1e-3, rho=0.9, mu
         =0.9, eps=1e-8,
1257                                hurst=0.7, sigma=0.01,
1258                                burst_prob=0.05, num_bursts=None
         , burst_len=32,
1259                                burst_scale=1.0, seed=SEED):
1260
1261     tf.keras.backend.clear_session()
1262     xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
         burst_len, burst_scale, seed)
1263
1264     model = modelin
1265     r_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
          v in model.trainable_variables]
```

```
    v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for v in model.trainable_variables]
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    @tf.function(jit_compile=True)
    def step(xb, yb, noise):
        with tf.GradientTape() as tape:
            loss = loss_fn(yb, model(xb, training=True))
        grads = tape.gradient(loss, model.trainable_variables)

        for g, r, v, w in zip(grads, r_buf, v_buf, model.trainable_variables):
            r.assign(rho * r + (1. - rho) * tf.square(g))
            rms_step = eta * g / (tf.sqrt(r) + eps)
            v.assign(mu * v + rms_step + noise)
            w.assign_sub(v)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "RMSprop+fBm(Burst)", "Time (s)": round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met.items()}}
```

```
1292
1293 def run_rmsprop_fbm_brust_weight_noise(modelin, eta=1e-3, rho
     =0.9, mu=0.9, eps=1e-8,
1294                                        hurst=0.7, sigma=0.01,
1295                                        burst_prob=0.05,
     num_bursts=None, burst_len=32,
1296                                        burst_scale=1.0, seed=
     SEED):
1297
1298     tf.keras.backend.clear_session()
1299     xi = _make_burst_xi(hurst, sigma, burst_prob, num_bursts,
     burst_len, burst_scale, seed)
1300
1301     model = modelin
1302     r_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
      v in model.trainable_variables]
1303     v_buf = [tf.Variable(tf.zeros_like(v), trainable=False) for
      v in model.trainable_variables]
1304     loss_fn = tf.keras.losses.CategoricalCrossentropy()
1305
1306     @tf.function(jit_compile=True)
1307     def step(xb, yb, noise):
1308         with tf.GradientTape() as tape:
1309             loss = loss_fn(yb, model(xb, training=True))
1310         grads = tape.gradient(loss, model.trainable_variables)
1311
1312         for g, r, v, w in zip(grads, r_buf, v_buf, model.
     trainable_variables):
1313             r.assign(rho * r + (1. - rho) * tf.square(g))
1314             rms_step = eta * g / (tf.sqrt(r) + eps)
```

```
            v.assign(mu * v + rms_step)
            w.assign_sub(v)
            w.assign_add(noise)

    xb, yb = next(iter(tr_ds)); step(xb, yb, xi[0])

    t0 = time.time()
    for i, (xb, yb) in enumerate(tr_ds.repeat(EPOCHS)):
        step(xb, yb, xi[i])
    runtime = time.time() - t0

    met = evaluate(model)
    return {"Optimizer": "RMSprop+fBm-W(Burst)", "Time (s)":
    round(runtime, 2),
            "Weight Updates": TOTAL_STEPS,
            **{k.replace('_', ' '): round(v, 6) for k, v in met
    .items()}}
```

Listing B.3: Main functions

```
def make_model(): #MNIST model
    with tf.device(DEV):
        inp  = tf.keras.Input(shape=(28, 28))
        x    = tf.keras.layers.Flatten()(inp)
        x    = tf.keras.layers.Dense(128, activation="relu")(x)
        out  = tf.keras.layers.Dense(10,  activation="softmax")
    (x)
        return tf.keras.Model(inp, out)
```

Listing B.4: MNIST model

```
def make_cifar_model():
```

```
    inputs = tf.keras.Input(shape=(32, 32, 3))

    x = tf.keras.layers.Conv2D(32, 3, padding="same", use_bias=
    False)(inputs)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)
    x = tf.keras.layers.MaxPooling2D(2)(x)

    x = tf.keras.layers.Conv2D(64, 3, padding="same", use_bias=
    False)(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)
    x = tf.keras.layers.MaxPooling2D(2)(x)

    x = tf.keras.layers.Conv2D(128, 3, padding="same", use_bias
    =False)(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)

    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dense(128, activation="relu")(x)
    outputs = tf.keras.layers.Dense(10, activation="softmax")(x
    )

    model = tf.keras.Model(inputs, outputs)
    return model
```

Listing B.5: CIFAR model

```
def make_nsl_ann(input_dim:int,
                 num_classes:int = 2,
```

```
                hidden:tuple[int,...] = (256, 128, 64),
                dropout:float = 0.3):
    inputs = tf.keras.Input(shape=(input_dim,))
    x = inputs
    for units in hidden:
        x = tf.keras.layers.Dense(units, activation="relu")(x)
        x = tf.keras.layers.Dropout(dropout)(x)
    outputs = tf.keras.layers.Dense(num_classes, activation="
    softmax")(x)
    return tf.keras.Model(inputs, outputs)
```

Listing B.6: NSL model

```
def make_cic_ann(input_dim, num_classes, layers=(256, 128, 64),
     drop=0.3):
    x = inp = tf.keras.Input(shape=(input_dim,))
    for units in layers:
        x = tf.keras.layers.Dense(units, activation="relu")(x)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Dropout(drop)(x)
        drop *= 0.5
    out = tf.keras.layers.Dense(num_classes, activation="
    softmax")(x)
    return tf.keras.Model(inp, out)
```

Listing B.7: CIC model

```
def make_simple_ann(input_dim, num_classes, dropout=0.30):
    return tf.keras.Sequential([
        tf.keras.layers.Input(shape=(input_dim,)),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.BatchNormalization(),
```

```
        tf.keras.layers.Dropout(dropout),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dropout(dropout * 0.5),
        tf.keras.layers.Dense(num_classes, activation="softmax"
    )
    ])
```

Listing B.8: BOT model

```
BENCH_FUNCS = [
    lambda: run_plain_sgd(build_mnist_model()),
    lambda: run_plain_adam(build_mnist_model()),
    lambda: run_plain_nadam(build_mnist_model()),
    lambda: run_plain_rmsprop(build_mnist_model()),

    lambda: run_sgd_fbm(build_mnist_model(), hurst=HLIST),
    lambda: run_adam_fbm_fast(build_mnist_model(), hurst=HLIST)
    ,
    lambda: run_nadam_fbm_fast(build_mnist_model(), hurst=HLIST
    ),
    lambda: run_rmsprop_fbm_fast(build_mnist_model(), hurst=
    HLIST),

    lambda: run_sgd_fbm_weight_noise(build_mnist_model(), hurst
    =HLIST),
    lambda: run_adam_fbm_weight_noise(build_mnist_model(),
    hurst=HLIST),
    lambda: run_nadam_fbm_weight_noise(build_mnist_model(),
    hurst=HLIST),
    lambda: run_rmsprop_fbm_weight_noise(build_mnist_model(),
```

```
    hurst=HLIST),
16
17      lambda: run_sgd_adaptive_fbm(build_mnist_model(), hurst=
    HLIST),
18      lambda: run_adam_adaptive_fbm(build_mnist_model(), hurst=
    HLIST),
19      lambda: run_nadam_adaptive_fbm(build_mnist_model(), hurst=
    HLIST),
20      lambda: run_rmsprop_adaptive_fbm(build_mnist_model(), hurst
    =HLIST),
21
22      lambda: run_sgd_adaptive_fbm_weight_noise(build_mnist_model
    (), hurst=HLIST),
23      lambda: run_adam_adaptive_fbm_weight_noise(
    build_mnist_model(), hurst=HLIST),
24      lambda: run_nadam_adaptive_fbm_weight_noise(
    build_mnist_model(), hurst=HLIST),
25      lambda: run_rmsprop_adaptive_fbm_weight_noise(
    build_mnist_model(), hurst=HLIST),
26
27
28      lambda: run_sgd_fbm_brust_weight_noise(build_mnist_model(),
     hurst=HLIST, num_bursts=40, burst_len=32),
29      lambda: run_adam_fbm_brust_weight_noise(build_mnist_model()
    , hurst=HLIST, burst_prob=0.03),
30      lambda: run_nadam_fbm_brust_weight_noise(build_mnist_model
    (), hurst=HLIST, burst_prob=0.03),
31      lambda: run_rmsprop_fbm_brust_weight_noise(
    build_mnist_model(), hurst=HLIST, num_bursts=20, burst_len
    =128),
```

```
32
33 ]
34
35
36 OUT_DIR = Path("benchmark_out")
37 OUT_DIR.mkdir(exist_ok=True)
38
39 results = []
40 for r in range(RUNS):
41     print(f"\n=== Repetition {r+1}/{RUNS} ===")
42     for fn in BENCH_FUNCS:
43         rec = fn()
44         rec["Run"] = r + 1
45         results.append(rec)
46
47 df_all = pd.DataFrame(results)
48 df_all.to_csv(OUT_DIR / "mnist_runs.csv", index=False)
49 print("\nSaved raw runs to", OUT_DIR / "mnist_runs.csv")
```

Listing B.9: Run and save mnist example