Ridge Regression and Random Neural Networks under High-Dimensional Conditions

Derivative-Based Calibration for Stable Learning

Yehor Ruban

TU Delft

Ridge Regression and Random Neural Networks under High-Dimensional Conditions

Derivative-Based Calibration for Stable Learning

by

Yehor Ruban

Instructor: N. Parolya
Faculty: Electrical Engineering, Mathematics and Computer Science

TU Delft

Summary

This thesis investigates the problem of selecting an optimal regularization parameter in high-dimensional ridge regression models with random features. The work is situated at the intersection of machine learning, statistical signal processing, and random matrix theory, and aims to improve the understanding and stability of regression-based learning in high-dimensional settings.

When the number of model parameters becomes comparable to the number of training samples, conventional statistical assumptions no longer hold, and model performance can exhibit sharp transitions or instability. In such regimes, ridge regularization plays a critical role in balancing model bias and variance. However, determining the optimal regularization parameter γ is non-trivial: existing techniques such as cross-validation are computationally costly, while analytical approaches based on deterministic equivalents—such as the method proposed by Couillet and Liao [6]—rely on asymptotic knowledge of population statistics that may be inaccessible in practice.

To address this challenge, this thesis introduces a new *derivative-based calibration rule* for ridge regression in random-feature neural networks. The proposed method identifies the optimal value of γ directly from data by analyzing the empirical derivative of the training loss with respect to γ. This criterion provides a fully data-driven, computationally efficient alternative to deterministic-equivalent or cross-validation-based calibration.

Theoretical development and analysis are supported by controlled numerical experiments. Synthetic data simulations confirm that the derivative-based calibration achieves predictive performance comparable to Couillet's deterministic-equivalent rule across a wide range of dimensional ratios. Furthermore, perturbation analysis demonstrates that the proposed method yields a more stable feature-weight vector β under small input variations, indicating improved robustness and reproducibility of the learned model.

Validation on real data is performed using the Fashion–MNIST dataset in a binary classification setup. Results show that both calibration rules produce similar test accuracies (above 99%), while the derivative-based approach consistently exhibits lower coefficient variance and better recall. This stability of β suggests that the model learns a more reliable internal representation, which can be advantageous in practical applications where interpretability, consistency, and downstream reuse of learned features are important.

Finally, the discussion situates these findings in the context of broader research on high-dimensional learning stability, linking them to studies in bioinformatics, neuroimaging, and industrial fault detection where reproducibility of learned representations is critical. The thesis concludes that derivative-based calibration provides a simple, effective, and theoretically grounded alternative to existing methods for ridge parameter selection, combining strong empirical performance with enhanced stability and interpretability.

Contents

1 Introduction

1.1. Background and Motivation

In modern machine learning, models with a large number of parameters often outperform smaller ones, even in cases where the number of features approaches or exceeds the number of training samples. This high-dimensional regime challenges traditional statistical assumptions and motivates the need for new analytical tools to understand model behavior, stability, and generalization.

Neural networks, especially those with random or fixed hidden weights, provide a simple yet powerful framework for studying these questions. When the hidden-layer weights are drawn at random, the network acts as a nonlinear feature extractor, and the final linear layer reduces to a regression problem in the resulting random feature space. This connection allows techniques from classical statistics and random matrix theory to be used for analysis.

Among the most important components in such models is the choice of the regularization parameter in ridge regression. It controls the balance between fitting the data and maintaining stability of the learned coefficients. Traditional tuning methods, such as cross-validation or asymptotic formulas, can be computationally costly or rely on strong model assumptions. This motivates the search for alternative, data-driven calibration approaches that can perform reliably in high-dimensional settings.

1.2. Research Objectives

The main objective of this thesis is to develop and analyze a new method for selecting the ridge regularization parameter in random-feature regression models. The proposed *derivative-based calibration rule* determines the optimal regularization strength by analyzing the empirical derivative of the training loss with respect to the regularization parameter. The approach is entirely data-driven and avoids the need for repeated training or asymptotic parameter estimation.

This work builds upon the theoretical framework introduced by Romain Couillet and co-authors in their book *Random Matrix Methods for Machine Learning* [6] and related article [29], which provide deterministic-equivalent analyses of high-dimensional ridge regression and random-feature neural networks. Their framework offers a rigorous, asymptotic approach to understanding how regularization behaves when both the number of features and the number of samples grow large. Within this context, Couillet and colleagues proposed a deterministic-equivalent calibration method for choosing the optimal regularization parameter based on random matrix theory.

The present thesis extends this line of work by introducing an alternative, fully data-driven calibration scheme that does not rely on asymptotic knowledge of model parameters. Specifically, the goals are

to:

- Formulate the derivative-based regularization rule within the framework of random-feature ridge regression;
- Compare its performance to Couillet's deterministic-equivalent calibration method, both theoretically and empirically;
- Evaluate the method's stability under data perturbations and assess how it affects the learned coefficients β;
- Validate the findings through controlled simulations and real-data experiments using the Fashion–MNIST dataset.

Through these objectives, the work seeks to contribute both a practical calibration tool and a deeper understanding of how regularization influences model stability in high-dimensional random networks.

1.3. Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 reviews the theoretical background, including neural networks, random features, ridge regression, and random matrix theory. Chapter 3 develops the mathematical framework and introduces the proposed derivative-based calibration rule. Chapter 4 presents numerical experiments on both synthetic and real data, comparing the proposed method with existing approaches. Chapter 5 discusses the implications of the results, highlighting the theoretical and practical significance of coefficient stability. Finally, Chapter 6 concludes the thesis and outlines directions for future research.

Overall, the thesis combines theoretical analysis, numerical validation, and conceptual interpretation to provide a comprehensive view of regularization and stability in high-dimensional random-feature models.

2

Literature Review

Chapter 2 provides an overview of the main concepts and historical advances that underpin this thesis. We begin in Section 1 with a general and broad introduction to the concept of neural networks, highlighting both their historical origins and their progression into deep, multi-layer architectures. This sets the stage for why neural networks have become a dominant tool for complex learning problems and tasks, but also why it can be potentially challenging to analyze them in a theoretical way.

From there, Section 2 narrows the focus to random neural networks, emphasizing how random weights can be viewed as an efficient means of feature extraction, particularly in contexts with limited data or strict computational budgets. We also discuss the use of randomization in modern practice—ranging from dropout to "learning without backpropagation"—as these techniques share conceptual parallels with the simpler random networks studied in this thesis.

In Section 3, we demonstrate that regression can be seen as a natural counterpart to single-layer neural networks, especially through the lens of ridge regression on random features. Here, we connect the dots between "pure" linear regression on one hand and single-hidden-layer "extreme learning machines" on the other, underscoring how penalization (in the form of ridge regression) addresses overfitting and stabilizes parameter estimates in neural nets.

Following that, Section 4 addresses the high-dimensional regime and the corresponding role of random matrix theory (RMT). Modern applications often operate in scenarios where the number of parameters and/or features is comparable to (or larger than) the number of data points. We show that RMT, with its body of results on large random matrices and kernels, offers valuable insights into the behavior of large-scale regressions and random neural networks alike.

Finally, Section 5 synthesizes these threads—bridging random neural networks, ridge regression, and high-dimensional analysis—to foreshadow the technical work and contributions of this thesis. By understanding the interplay among these ingredients, we can pinpoint how random fixed-weight models can yield theoretical insights and practical benefits in high-dimensional settings.

2.1. Introduction to Neural Networks

Neural networks are one of the most important paradigms in machine learning, roughly inspired by the structure of actual biological neurons. The core idea involves organizing computational units (called 'neurons') into interconnected layers that in turn learn to approximate functions straight from the data. Over the past decades, this arrangement design has evolved from basic, single-layer perceptrons into deep, multi-layer architectures that can perform well in tasks such as image recognition, natural language processing, even content generation. All of the aforementioned has cemented neural networks

as the primary tool in modern understanding of artificial intelligence.

In this section, we first examine the historical perspective on neural networks, shedding light on how the early perceptrons established foundational ideas that still resonate today. Subsequently, we explore the journey from simple to deep networks, highlighting key developments and the shift toward highly expressive, large-scale models. This progression not only frames why advanced neural networks can be so powerful, but it also underscores the growing need to better understand simpler random-weight approaches both for computational practicality and for theoretical clarity.

2.1.1. Historical Perspective

The origins of artificial neural networks (ANNs) trace back to the mid-twentieth century, when researchers first began exploring simplified mathematical models of biological neurons. One of the earliest milestones was the perceptron model introduced by Rosenblatt in 1958 [41], which formalized a neuron as a linear threshold unit capable of learning simple decision boundaries. The perceptron embodied the promise of machine learning before the term was even widely used: a system that could adapt its parameters based on examples rather than explicit programming.

The optimism surrounding early neural networks was tempered by their theoretical and computational limitations. In 1969, Minsky and Papert [34] rigorously demonstrated that a single-layer perceptron could not represent non-linearly separable functions such as the XOR problem. This result, combined with limited computational resources, led to a decline in neural network research, often referred to as the first ”AI winter.” Nevertheless, the perceptron era established the mathematical and conceptual foundations for later advances in supervised learning.

Interest was reignited in the 1980s with the rediscovery and popularization of the *backpropagation* algorithm [42], which enabled efficient gradient-based training of multi-layer networks. Backpropagation, initially studied in control theory and later formalized in the context of neural computation, allowed the training of deep feedforward networks through layerwise error propagation. Around the same period, Hopfield [17] introduced recurrent neural networks as energy-based systems, and Kohonen [23] proposed self-organizing maps for unsupervised learning - broadening the scope of connectionist models beyond simple classification.

By the early 1990s, networks such as the multilayer perceptron (MLP) and convolutional neural network (CNN) [26] were already demonstrating strong performance on specific pattern-recognition tasks. However, the combination of limited computational power and small datasets again constrained progress. It was not until the 2000s, when large annotated datasets, increased computational resources (notably GPUs), and improved regularization methods such as dropout [45] became available - that neural networks began their modern resurgence. This period marked the advent of ”deep learning” [25], characterized by very deep architectures and millions of parameters capable of learning highly complex, hierarchical feature representations.

In summary, the historical trajectory of neural networks can be viewed as an evolution from simple, interpretable linear classifiers to powerful but opaque high-dimensional systems. This progression contextualizes the present thesis: while modern deep networks achieve remarkable empirical success, understanding their behavior remains challenging. Studying simplified models, such as random-feature neural networks analyzed through the lens of ridge regression and random matrix theory - offers a principled way to recover theoretical tractability without abandoning the essential nonlinear mechanisms that make neural networks effective.

2.1.2. From Simple to Deep

While early neural networks such as the perceptron demonstrated the feasibility of data-driven learning, their representational power was limited to linearly separable problems. The introduction of hidden layers provided a crucial breakthrough: multi-layer networks could approximate arbitrary continuous

functions under mild assumptions, as formalized in the *universal approximation theorem* [8, 18]. This theoretical result established that even relatively shallow networks, when equipped with non-linear activation functions, possess the capacity to represent complex mappings between inputs and outputs.

In practical terms, however, training such networks remained difficult throughout the 1980s and 1990s. Deep architectures suffered from vanishing gradients, poor initialization, and overfitting in the absence of large datasets. Research therefore concentrated on specific network designs that exploited structure in the data. Notable among these were the convolutional neural networks (CNNs) of LeCun *et al.* [27], which leveraged spatial weight sharing for image data, and recurrent neural networks (RNNs) designed for sequential or temporal patterns [11]. These architectures anticipated modern deep learning by embedding strong inductive biases into network topology.

The modern ”deep learning revolution” emerged in the mid-2000s, driven by three converging factors: increased computational power (notably GPU acceleration), large-scale datasets such as ImageNet [9], and methodological innovations in optimization and regularization. Layer-wise unsupervised pretraining using deep belief networks [15] and autoencoders [48] mitigated initialization problems, while activation functions such as the rectified linear unit (ReLU) [35] and regularization techniques like dropout [45] enabled stable gradient propagation and improved generalization. Collectively, these developments unlocked the practical potential of deep architectures containing tens or even hundreds of layers.

Subsequent milestones rapidly followed. Architectures such as AlexNet [24], VGG [44], ResNet [14], and Transformers [47] demonstrated that depth, residual connections, and attention mechanisms could be scaled to unprecedented levels, achieving state-of-the-art performance across vision, language, and multimodal tasks. Yet, as networks grew deeper and more parameter-rich, their theoretical understanding became more opaque. The same complexity that made deep networks powerful also made them difficult to analyze or interpret rigorously.

This realization has inspired a complementary line of research that seeks tractable, theoretically grounded models capturing the essential mechanisms of neural computation without the full complexity of deep learning. Among these are random-feature networks, kernel approximations, and mean-field analyses derived from random matrix theory. The present thesis situates itself within this latter tradition: by examining simplified random-weight neural networks through the lens of ridge regression, we aim to bridge the gap between deep learning practice and high-dimensional statistical theory.

2.2. Random Neural Networks

The study of random neural networks (RNNs)—also known as *random-feature models* or *extreme learning machines* (ELMs)—represents a significant simplification of traditional neural networks. In these architectures, the weights of certain layers (often the hidden or feature-extraction layers) are not trained, but instead initialized randomly and kept fixed throughout learning. Only the final layer, typically a linear readout, is optimized. This randomization transforms the learning problem from a non-convex optimization task into a tractable linear regression, enabling both analytical treatment and rapid computation.

2.2.1. Random Weights and Efficient Feature Extractors

The conceptual foundation of random-weight networks can be traced to early work on fixed-feature representations and kernel methods. Neal [36] first demonstrated that infinitely wide neural networks with random weights converge to Gaussian processes, establishing a deep connection between random networks and Bayesian kernel machines. Later, Rahimi and Recht [39] proposed the *random kitchen sinks* approach, showing that a wide class of shift-invariant kernels can be efficiently approximated by

random feature maps of the form

$$\phi(\mathbf{x}) = \sqrt{\frac{2}{N}} \begin{bmatrix} \cos(\mathbf{w}_1^\top \mathbf{x} + b_1) \\ \cos(\mathbf{w}_2^\top \mathbf{x} + b_2) \\ \vdots \\ \cos(\mathbf{w}_N^\top \mathbf{x} + b_N) \end{bmatrix}, \tag{2.1}$$

where each $\mathbf{w}_i$ is sampled independently from a distribution proportional to the Fourier transform of the kernel, and b_i is a random phase. This approach effectively replaces expensive kernel computations with low-dimensional random projections followed by a linear model, enabling scalable kernel learning in high dimensions.

A closely related stream of research explored randomization directly within neural architectures. In the *Extreme Learning Machine* framework [19], the hidden-layer weights $\mathbf{W}$ are drawn at random and fixed, while the output weights $\boldsymbol{\beta}$ are obtained in closed form through ridge regression:

$$\boldsymbol{\beta} = (\Sigma^\top \Sigma + \gamma I_N)^{-1} \Sigma^\top \mathbf{Y}, \tag{2.2}$$

where $\Sigma = \sigma(\mathbf{W}\mathbf{X})$ denotes the matrix of hidden-layer activations and $\sigma(\cdot)$ is a nonlinearity such as ReLU or sigmoid. This simple construction can approximate complex mappings while avoiding the heavy computational cost of backpropagation. Huang and colleagues showed that even with completely random hidden weights, the ELM achieves strong generalization and extremely fast training times, laying the foundation for the modern random-feature viewpoint.

From a theoretical standpoint, random-weight networks can be understood as finite-dimensional approximations of kernel machines. The random mapping $\phi(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x})$ induces an implicit kernel

$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}_{\mathbf{w}}[\sigma(\mathbf{w}^\top \mathbf{x})\, \sigma(\mathbf{w}^\top \mathbf{x}')], \tag{2.3}$$

which for specific choices of activation functions yields well-known kernels such as the *arc-cosine kernel* [4]. The random network thus serves as a Monte Carlo estimator of this kernel, bridging the gap between neural and kernel-based learning.

Randomization also provides a form of implicit regularization. Fixed random features prevent the model from overfitting by constraining its expressiveness and decoupling feature learning from weight optimization. Furthermore, randomization simplifies theoretical analysis, allowing tools from random matrix theory and statistical physics to be applied directly to study generalization, eigenvalue spectra, and limiting distributions [38, 7]. These analyses have shown that even simple random networks exhibit complex phase transitions between under- and over-parameterized regimes—insights that are difficult to obtain in fully trained deep networks.

2.2.2. Random Neural Networks in Practice

Despite their simplicity, random-feature models have found wide use in practice. Rahimi and Recht's work [40] demonstrated that random Fourier features can scale kernel learning to millions of data points without loss in accuracy. The same principle underlies several modern architectures: reservoir computing [21, 30], echo-state networks, and certain forms of dropout regularization can all be interpreted as imposing structured randomness within neural dynamics.

In computer vision and signal processing, random convolutional features have been used as lightweight alternatives to deep learned filters when computational efficiency or interpretability is required [43]. In reinforcement learning and continual learning, random projections help stabilize representation drift by maintaining a fixed feature subspace [37]. Moreover, recent theoretical works [20, 32] have demonstrated that the behavior of gradient-trained wide networks can often be approximated by their random-feature counterparts in the so-called *neural tangent kernel* (NTK) regime.

The appeal of random neural networks lies in this balance between expressiveness and tractability. They are expressive enough to model nonlinear relationships, yet simple enough to admit precise

theoretical characterizations. In the context of this thesis, this property is essential: by freezing the random features and focusing on the analytical form of the ridge regression solution (2.2), we can study high-dimensional learning dynamics directly, derive deterministic equivalents, and explore new data-driven regularization strategies such as the derivative-based calibration proposed in later chapters.

2.3. Regression in a Neural Networks Context

The connection between regression and neural networks is more than superficial: at its core, a feedforward neural network implements a composition of affine transformations and nonlinearities, followed by a linear readout. When the nonlinear mapping is fixed or random, the learning task in the final layer reduces to a regression problem on transformed features. This observation underpins both theoretical analyses of neural networks and practical algorithms such as the extreme learning machine and random feature regression used throughout this thesis.

2.3.1. Regression–Neural Network Problem Equivalence

Consider a single-hidden-layer neural network with input $\mathbf{x} \in \mathbb{R}^p$, hidden weights $\mathbf{W} \in \mathbb{R}^{N\times p}$, activation function $\sigma(\cdot)$, and output weights $\boldsymbol{\beta} \in \mathbb{R}^N$. The network output for a given input is

$$f(\mathbf{x}) = \boldsymbol{\beta}^\top \sigma(\mathbf{W}\mathbf{x}). \tag{2.4}$$

Given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, stacking all activations into a matrix

$$\Sigma = \begin{bmatrix} \sigma(\mathbf{W}\mathbf{x}_1)^\top \\ \sigma(\mathbf{W}\mathbf{x}_2)^\top \\ \vdots \\ \sigma(\mathbf{W}\mathbf{x}_n)^\top \end{bmatrix} \in \mathbb{R}^{n\times N}, \tag{2.5}$$

the empirical training objective under a squared-loss criterion becomes

$$\min_{\boldsymbol{\beta}} \ \frac{1}{n}\|\Sigma\boldsymbol{\beta} - \mathbf{Y}\|_2^2, \tag{2.6}$$

where $\mathbf{Y} = [y_1, y_2, \ldots, y_n]^\top$. If $\mathbf{W}$ and σ are fixed, the optimization in (2.6) is identical to a linear regression on the nonlinear feature vectors $\phi(\mathbf{x}_i) = \sigma(\mathbf{W}\mathbf{x}_i)$. In this sense, the neural network acts as a nonlinear feature generator, while the training process in the final layer corresponds to ordinary least squares (OLS) estimation:

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\Sigma^\top\Sigma)^{-1}\Sigma^\top\mathbf{Y}. \tag{2.7}$$

This equivalence highlights the dual view of single-layer networks as regressors operating in a random feature space. From a functional perspective, $f(\mathbf{x})$ approximates an underlying mapping $f^*(\mathbf{x})$ via linear combination of basis functions $\{\sigma(\mathbf{w}_i^\top\mathbf{x})\}_{i=1}^N$, analogously to classical regression models that approximate functions through polynomial or spline bases. When N is large, the random basis becomes highly expressive, and the regression solution (2.7) provides an efficient projection of the target function onto the subspace spanned by the random features.

2.3.2. Ridge Regression (Penalization)

In high-dimensional settings, where N (the number of features) is comparable to or larger than n (the number of samples), the matrix $\Sigma^\top\Sigma$ in (2.7) is often ill-conditioned or singular. This makes the OLS solution unstable and highly sensitive to small perturbations in the data. To mitigate this, a regularization term is introduced, leading to the *ridge regression* or *Tikhonov regularization* estimator:

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\Sigma^\top\Sigma + \gamma I_N)^{-1}\Sigma^\top\mathbf{Y}, \tag{2.8}$$

where $\gamma > 0$ is the regularization parameter controlling the trade-off between data fidelity and weight magnitude.

Ridge regression can be interpreted from several complementary perspectives:

- **Statistical:** it shrinks coefficient estimates toward zero, reducing variance at the cost of a small bias [16]. This stabilizes predictions in the presence of multicollinearity or limited data.
- **Bayesian:** it corresponds to a maximum a posteriori (MAP) estimator under a Gaussian prior $\boldsymbol{\beta} \sim \mathcal{N}(0, \frac{1}{\gamma} I_N)$, linking regularization to prior belief on model complexity [46].
- **Numerical:** it improves the conditioning of $\Sigma^\top \Sigma$, ensuring invertibility and controlling the amplification of noise.

In the neural-network setting, ridge regularization plays a role analogous to weight decay or ℓ_2-regularization in deep learning. The effect of γ on the spectrum of $(\Sigma^\top \Sigma)$ determines both the smoothness of the fitted function and the stability of the learned weights [12]. As $\gamma \to 0$, the estimator approaches the minimum-norm interpolator; as $\gamma \to \infty$, it collapses toward zero.

Formally, ridge regression defines a kernel machine with implicit kernel

$$k(\mathbf{x}, \mathbf{x}') = \frac{1}{N} \sigma(\mathbf{W}\mathbf{x})^\top (\sigma(\mathbf{W}\mathbf{x}')), \tag{2.9}$$

so that the predictor may be expressed equivalently as

$$f(\mathbf{x}) = \mathbf{k}(\mathbf{x})^\top (\mathbf{K} + \gamma I_n)^{-1} \mathbf{Y}, \tag{2.10}$$

where $\mathbf{K} = \Sigma\Sigma^\top / N$ is the empirical kernel matrix. This dual formulation makes the connection between ridge regression, kernel methods, and random-feature neural networks explicit, establishing a unified mathematical framework that later sections of this thesis leverage for high-dimensional analysis and regularization calibration.

2.4. High-Dimensional Statistics and Random Matrix Theory

The practical and theoretical challenges of modern machine learning often arise in the so-called *high-dimensional regime*, where the number of model parameters or features is comparable to, or even exceeds, the number of available samples. Classical statistical theory, which assumes a fixed feature dimension p and an increasing sample size $n \to \infty$, no longer provides reliable approximations in this regime. Instead, understanding the behavior of estimators and algorithms requires asymptotic frameworks that allow both p and n to grow together at a comparable rate.

2.4.1. High-Dimensional Regime

In high-dimensional statistics, quantities that were once negligible—such as correlations among features or small eigenvalues of sample covariance matrices—become dominant. A central parameter characterizing this setting is the *aspect ratio*

$$c = \frac{p}{n}, \tag{2.11}$$

which measures the dimensionality of the problem relative to the number of observations. When $c \ll 1$, the system is overdetermined, and classical intuition holds: estimators are consistent, and sample covariance matrices approximate their population counterparts. However, when c approaches or exceeds unity ($c \geq 1$), the sample covariance matrix becomes singular, and naive estimators such as ordinary least squares (2.7) become ill-posed. This is precisely the regime where regularization methods like ridge regression become essential.

The study of this high-dimensional behavior has revealed a series of nonintuitive phenomena. For example, ridge regression and related estimators exhibit a *double descent* risk curve [3], where increasing model complexity initially worsens generalization (the classical bias–variance trade-off) but later improves it again once $p > n$. These effects stem from spectral properties of random design matrices and have motivated the development of asymptotic tools to characterize estimator performance when both p and n are large.

Modern theoretical analyses therefore treat the design matrix $X \in \mathbb{R}^{p\times n}$ as random, often assuming entries that are independent and identically distributed (i.i.d.) with zero mean and variance scaled as $1/n$. This scaling ensures nontrivial limiting spectra as $n, p \to \infty$. The statistics of the eigenvalues of sample covariance matrices $S = \frac{1}{n}XX^\top$ form the cornerstone of this analysis and are governed by results from *random matrix theory* (RMT).

2.4.2. Random Matrix Theory Essentials

Random matrix theory provides a framework for studying the spectral behavior of large random matrices—such as covariance matrices, kernel matrices, or weight matrices in neural networks—when their dimensions grow without bound. Originating in nuclear physics with Wigner's work on energy spectra [50], RMT has since become a central tool in high-dimensional statistics [2] and modern machine learning [7].

One of the most fundamental results in this field is the *Marchenko–Pastur law* [31], which describes the limiting empirical distribution of eigenvalues of the sample covariance matrix $S = \frac{1}{n}XX^\top$. If the entries of X are i.i.d. with mean zero and variance $1/n$, then as $p, n \to \infty$ with $p/n \to c$, the spectral density of S converges almost surely to

$$\rho(\lambda) = (1 - \tfrac{1}{c})_+ \, \delta(\lambda) + \frac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{2\pi c\lambda}, \qquad \lambda \in [\lambda_-, \lambda_+], \tag{2.12}$$

where $\lambda_\pm = (1 \pm \sqrt{c})^2$ define the spectral support and $(x)_+ = \max(0, x)$. This result implies that, even in the absence of structure, the eigenvalues of high-dimensional covariance matrices exhibit systematic, nontrivial distributions that deviate from their population counterparts.

RMT also provides tools such as the *Stieltjes transform*, defined as

$$m(z) = \int \frac{1}{\lambda - z}\, d\rho(\lambda), \tag{2.13}$$

which serves as a key analytical function for characterizing the spectral properties of random matrices and for deriving deterministic equivalents in high-dimensional limits. These equivalents approximate random quantities—such as the generalization error or the trace of resolvents—with deterministic limits as $n, p \to \infty$ [5].

In the context of ridge regression and random-feature networks, random matrix theory enables closed-form asymptotic expressions for training and test errors [10, 28, 13]. These results make it possible to predict model behavior and regularization effects without Monte Carlo simulations, by solving fixed-point equations involving the spectral density (2.12) or its transform (2.13). Crucially, they reveal how phenomena such as double descent, bias–variance trade-offs, and stability depend on spectral properties of the random design matrix.

Overall, RMT provides the mathematical backbone for analyzing high-dimensional learning systems, including random-feature neural networks. It supplies the deterministic equivalents and asymptotic limits upon which the theoretical developments and calibration strategies of this thesis are built.

In this thesis, random matrix theory underpins not only the analysis of the ridge estimator but also the construction of the random-feature model itself. Specifically, the hidden-layer weight matrix $\mathbf{W}$ is drawn with i.i.d. Gaussian entries, a choice justified by RMT results showing that Gaussian ensembles yield well-defined and analytically tractable spectral distributions in the large-dimensional limit. This property

ensures that the empirical covariance of the random features $\Sigma = \sigma(\mathbf{WX})$ concentrates around its deterministic equivalent, enabling precise asymptotic analysis. Consequently, the theoretical framework of RMT directly supports both the modeling assumptions and the derivation of the derivative-based calibration rule proposed in this thesis, as it allows one to characterize how the choice of γ and the feature dimensionality jointly affect stability and generalization in high-dimensional regimes. That is to be discussed in details later in Chapter 3.

2.5. Synthesis and Relevance to Our Work

The preceding sections have outlined the key developments that converge to form the conceptual foundation of this thesis. Neural networks have evolved from simple perceptrons to deep, highly expressive architectures capable of learning complex representations. Alongside this empirical success, simplified variants such as random-feature networks and extreme learning machines have emerged as tractable models that retain much of the representational richness of deep networks while admitting analytical treatment.

The random-weight paradigm provides a crucial simplification: by fixing the hidden-layer parameters and focusing learning on a single linear output layer, the training problem reduces to a convex regression task. This structure allows the use of closed-form estimators such as ridge regression, linking neural computation directly to classical statistical theory. The equivalence between neural networks and regression models, made explicit in (2.8), reveals that many stability and generalization properties of networks are ultimately governed by the spectral characteristics of the random feature matrix $\Sigma = \sigma(\mathbf{WX})$.

High-dimensional statistics and random matrix theory extend this connection further by providing the analytical machinery to characterize such spectral properties. Results such as the Marchenko–Pastur law and its deterministic equivalents describe how the eigenvalue distributions of large random matrices govern estimator variance, bias, and stability. Through this lens, neural networks—and random-feature models in particular—can be studied as high-dimensional statistical systems whose performance depends not only on their architecture or activation function, but also on the asymptotic interactions between sample size, feature dimensionality, and regularization strength.

This theoretical synthesis motivates the central focus of the present thesis. Traditional calibration methods for ridge-type estimators, such as cross-validation or Couillet's deterministic-equivalent approach [6], provide systematic ways to tune the regularization parameter γ. However, these methods rely either on computationally intensive resampling or on asymptotic approximations that require full knowledge of model parameters. By contrast, the derivative-based calibration rule developed in this work offers a fully data-driven alternative: it determines the optimal γ by analyzing the empirical derivative of the training loss with respect to the regularization parameter itself.

Viewed in this broader context, the proposed approach embodies the same philosophy that underlies much of modern high-dimensional learning: to design estimators that remain theoretically interpretable, computationally efficient, and empirically robust even when operating in regimes far beyond the reach of classical statistical assumptions. The subsequent chapters build upon this synthesis, developing both the analytical framework and the empirical validation that demonstrate how derivative-based calibration can serve as a practical and theoretically grounded regularization principle for random-feature neural networks.

3

Methodology

In the methodology section, the setting of the study will first be described, detailing the high-dimensional setting and the corresponding parameters arrangement involved, as well as the mathematical formulation of the single-layer perceptron model with random Gaussian weights. Following this, the problem inherent in the conventional approach to ridge regression will be articulated, with a particular focus on the discrepancies observed in the optimal regularization parameter γ found through minimizing traditional for this setting error metrics as opposed to different, direct error measures. Finally, potential solutions to address these discrepancies will be proposed and explored, with the aim of refining the ridge regression method to achieve more reliable and consistent results in high-dimensional contexts.

In recent decades, random neural networks have gained prominence as an effective solution to various challenges in machine learning. These networks are particularly advantageous in scenarios with limited training data, as they help mitigate the constraints of computational and memory resources. Furthermore, random neural networks serve as efficient random feature extractors, enhancing their utility across a range of applications. In this context, the application of ridge regression within these networks, is explored, aiming to leverage their strengths while addressing the unique challenges they present.

3.1. Model Setting

As was stated before, focus of this study lies within researching behaviour of ridge regression methods in application with relatively simple neural network setting in high dimensions, and the issues of the conventional approach caused by such an arrangement. In this section, model setting as well as the toolbox of the original, traditional approach to high-dimensional ridge regression is being discussed.

3.1.1. Ridge Regression With Random Neural Network

A structure of the single-hidden-layer neural network that is being used for the studies is depicted on the figure 3.1. 2

Hence, the setting is the following:

Given the input data matrix $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathbb{R}^{p \times n}$, where each column $\mathbf{x}_i \in \mathbb{R}^p$ represents an input vector, the neural network processes this input through a randomly initialized weight matrix $\mathbf{W} \in \mathbb{R}^{N \times p}$. The entries of $\mathbf{W}$ are independent and identically distributed (i.i.d.), following a standard Gaussian distribution.

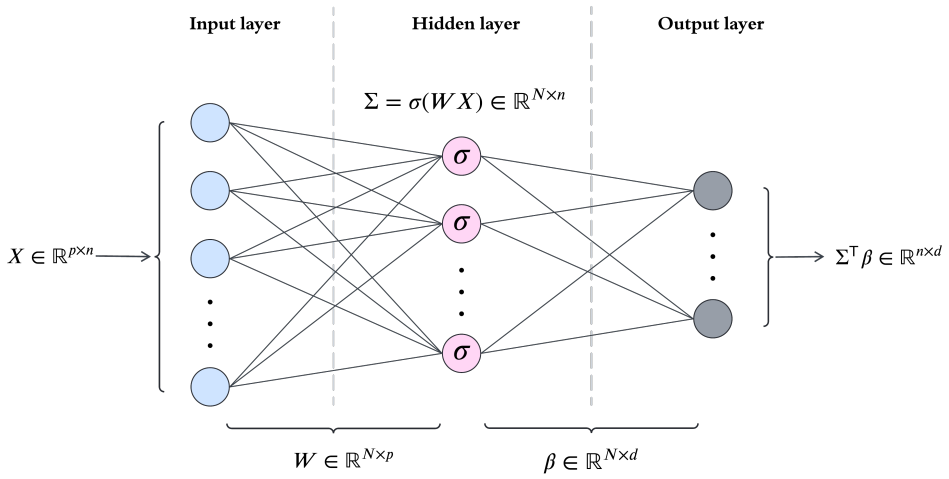


Figure 3.1: A diagram illustrating the structure of the single-layer random neural network.

The output of the first layer is denoted by $\mathbf{\Sigma} = \sigma(\mathbf{WX}) \in \mathbb{R}^{N\times n}$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is some activation function applied entry-wise to the product $\mathbf{WX}$. This results in a feature matrix $\mathbf{\Sigma}$, where each column $\sigma(\mathbf{W}\mathbf{x}_i)$ represents the transformed feature vector for the input $\mathbf{x}_i$. These columns can be interpreted as random features (nonlinear random features in case function σ is nonlinear itself) of the input data.

The second layer of the neural network involves a weight matrix $\beta \in \mathbb{R}^{N\times d}$, which is learned to map the feature matrix $\mathbf{\Sigma}$ to the target output $\mathbf{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_n] \in \mathbb{R}^{d\times n}$, where each column $\mathbf{y}_i \in \mathbb{R}^d$ represents the target vector associated with the input $\mathbf{x}_i$. The learning of β is performed by minimizing a regularized loss function.

Therefore, the overall architecture of the neural network involves transforming the input data through a random weight matrix, applying some activation function to obtain random nonlinear features, and then learning a weight matrix to map these features to the desired output. This setup leverages the randomness in the weights to extract useful features from the input data, while the learning process focuses on minimizing the prediction error through a regularized optimization problem.

The output weight matrix β is learned to minimize the regularized mean squared error:

$$L(\beta) = \frac{1}{n}\sum_{i=1}^{n} \left\|\mathbf{y}_i - \beta^\top \sigma(\mathbf{W}\mathbf{x}_i)\right\|^2 + \gamma \left\|\beta\right\|_F^2$$

The aim is to find the $\hat{\beta}$ that minimizes this loss. The loss function can be rewritten using matrix notation. The loss function can be written as:

$$L(\beta) = \frac{1}{n}\mathsf{Tr}((\mathbf{Y} - \beta^\top \mathbf{\Sigma})(\mathbf{Y} - \beta^\top \mathbf{\Sigma})^\top) + \gamma \mathsf{Tr}(\beta^\top \beta).$$

We then differentiate $L(\beta)$ with respect to β:

$$\frac{\partial L(\beta)}{\partial \beta} = \frac{\partial}{\partial \beta}\left(\frac{1}{n}\mathsf{Tr}((\mathbf{Y} - \beta^\top \mathbf{\Sigma})(\mathbf{Y} - \beta^\top \mathbf{\Sigma})^\top) + \gamma \mathsf{Tr}(\beta^\top \beta)\right).$$

Using matrix calculus, we have:

$$\frac{\partial}{\partial \beta}\left(\frac{1}{n}\mathsf{Tr}((\mathbf{Y} - \beta^\top \mathbf{\Sigma})(\mathbf{Y} - \beta^\top \mathbf{\Sigma})^\top)\right) = \frac{2}{n}\mathbf{\Sigma}(\mathbf{Y} - \beta^\top \mathbf{\Sigma})^\top,$$

and

$$\frac{\partial}{\partial \beta}\left(\gamma \mathsf{Tr}(\beta^\top \beta)\right) = 2\gamma\beta.$$

Combining these results, we get:

$$\frac{\partial L(\beta)}{\partial \beta} = \frac{2}{n}\mathbf{\Sigma}(\mathbf{Y} - \beta^\top \mathbf{\Sigma})^\top + 2\gamma\beta = \frac{2}{n}\mathbf{\Sigma}(\mathbf{\Sigma}^\top \beta - \mathbf{Y}^\top) + 2\gamma\beta.$$

Setting the gradient to zero for minimization, we have:

$$0 = \frac{2}{n}\mathbf{\Sigma}(\mathbf{\Sigma}^\top \hat{\beta} - \mathbf{Y}^\top) + 2\gamma\hat{\beta}.$$

From there, rearranging the terms:

$$0 = \frac{2}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top \hat{\beta} + \frac{2}{n}\mathbf{\Sigma}\mathbf{Y}^\top + 2\gamma\hat{\beta}$$

$$0 = \left(\frac{1}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top + \gamma \mathbf{I}_N\right)\hat{\beta} + \frac{1}{n}\mathbf{\Sigma}Y^\top$$

$$\left(\frac{1}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top + \gamma \mathbf{I}_N\right)\hat{\beta} = \frac{1}{n}\mathbf{\Sigma}\mathbf{Y}^\top$$

Thus we obtain the closed-form expression for the *ridge-regressor*:

$$\hat{\beta} = \frac{1}{n}\mathbf{\Sigma}\left(\frac{1}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top + \gamma \mathbf{I}_n\right)^{-1}\mathbf{Y}^\top. \tag{3.1}$$

3.1.2. Error Approximation in High-Dimensional Setting: a Conventional Approach

In the realm of high-dimensional settings, traditional approaches to error estimation have been well-documented and provide foundational insights into the behavior of models. This conventional approach is detailed extensively in the book [6].

Indeed, it is not difficult to observe that the error terms can be computed directly from the data. The training mean squared error (MSE) on the given training set $(\mathbf{X}, \mathbf{Y})$ is given by:

$$E_{\text{train}} = \frac{1}{n}\left\|\mathbf{Y}^\top - \mathbf{\Sigma}^\top \beta\right\|_F^2 = \frac{\gamma^2}{n}\text{tr}\mathbf{Y}\mathbf{Q}^2(\gamma)\mathbf{Y}^\top, \tag{3.2}$$

Please note, that here β is as in the formula (3.1), whilst $\mathbf{Q}(\gamma)$ is the resolvent of $\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma}$, defined as:

$$\mathbf{Q}(\gamma) \equiv \left(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma \mathbf{I}_n\right)^{-1}. \tag{3.3}$$

Similarly, the test MSE on a test set $(\hat{\mathbf{X}}, \hat{\mathbf{Y}}) \in \mathbb{R}^{p\times\hat{n}} \times \mathbb{R}^{d\times\hat{n}}$ of test sample size $\hat{n}$ is given by:

$$E_{\text{test}} = \frac{1}{\hat{n}}\left\|\hat{\mathbf{Y}}^\top - \hat{\mathbf{\Sigma}}^\top \beta\right\|_F^2 = \frac{1}{\hat{n}}\text{tr}\hat{\mathbf{Y}}\hat{\mathbf{Y}}^\top - \frac{2}{n\hat{n}}\text{tr}\mathbf{Y}\mathbf{Q}\mathbf{\Sigma}^\top\hat{\mathbf{\Sigma}}\hat{\mathbf{Y}}^\top + \frac{1}{n^2\hat{n}}\text{tr}\mathbf{Y}\mathbf{Q}\mathbf{\Sigma}^\top\hat{\mathbf{\Sigma}}\hat{\mathbf{\Sigma}}^\top\mathbf{\Sigma}\mathbf{Q}\mathbf{Y}^\top \tag{3.4}$$

where $\hat{\mathbf{\Sigma}} = \sigma(\mathbf{W}\hat{\mathbf{X}})$.

The traditional approach, as mentioned before, opts for effectively estimating these error 3.2 & 3.4 rather than using the original expressions. The question arises: why estimate errors when direct computation of E_{train} and E_{test} is possible? The challenge lies in the high-dimensional regime, where parameters p, n, and N (dimensionality of the data, size of the dataset and number of neurons in the hidden layer, respectively) grow large simultaneously. Under these conditions, computing E_{train}, for example, becomes computationally intensive due to the inversion and the inherent randomness of $\mathbf{\Sigma}$ through W.

The randomness, while not a direct computational hurdle, complicates the assessment of the asymptotic behavior of E_{train}. Small changes in input can lead to significant variations in output, complicating the understanding of error behavior. To address this, the resolvent $\hat{\mathbf{Q}}$ is employed to approximate the asymptotic behavior of E_{train}. This method leverages the resolvent to simplify the complexity associated with the randomness of $\mathbf{W}$. This logic is applicable to E_{test} as well, providing a comprehensive framework for error approximation in high-dimensional settings.

The approximations $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$ are based on the assumption that the matrix $\mathbf{W}$ is sub-Gaussian and that the function σ is Lipschitz continuous [29]. These assumptions are necessary to apply concentration inequalities and derive deterministic equivalents for the resolvent and the error terms.

Assumptions and Lemmas To rigorously justify the use of $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$, we rely on several key results from random matrix theory.

Assumption 1. (Sub-Gaussian $\mathbf{W}$): The matrix $\mathbf{W}$ is defined as $\mathbf{W} = \phi(\tilde{\mathbf{W}})$, where $\tilde{\mathbf{W}}$ has i.i.d. $\mathcal{N}(0,1)$ entries and $\phi(\cdot)$ is λ_ϕ-Lipschitz. For $a = \phi(b) \in \mathbb{R}$, ϕ maps $b \sim \mathcal{N}(0, I_p)$ to $a \sim \mathcal{N}_\phi(0, I_p)$.

Assumption 2. (Function σ): The function σ is Lipschitz continuous with parameter λ_σ. This assumption holds for many activation functions used in neural networks, such as the rectified linear unit (ReLU).

Assumption 3. (Growth Rate): As $n \to \infty$,

$$0 < \liminf_{n\to\infty} \min\left\{\frac{p}{n}, \frac{N}{n}\right\} \leq \limsup_{n\to\infty} \max\left\{\frac{p}{n}, \frac{N}{n}\right\} < \infty$$

while $\gamma, \lambda_\sigma, \lambda_\phi > 0$ and d are kept constant. Additionally,

$$\limsup_{n\to\infty} \|\mathbf{X}\| < \infty, \quad \limsup_{n\to\infty} \max_{ij} |\mathbf{Y}_{ij}| < \infty.$$

Under these assumptions, we can state the following key results:

Lemma 1. (Concentration of Quadratic Forms): Let Assumptions 1 and 2 hold. For $\mathbf{X} \in \mathbb{R}^{p\times n}$ and $w \sim \mathcal{N}_\phi(0, I_p)$, define the random vector $\sigma \equiv \sigma(w^\top \mathbf{X})$. For $\mathbf{A} \in \mathbb{R}^{n\times n}$ with $\|\mathbf{A}\| \leq 1$,

$$P\left(\left|\frac{1}{N}\sigma^\top \mathbf{A}\sigma - \frac{1}{N}\text{tr}(\mathbf{\Phi}\mathbf{A})\right| > t\right) \leq Ce^{-cN\min\left(\frac{t^2}{t_0^2}, t\right)}$$

for some constants $C, c > 0$, where $\mathbf{\Phi} = \mathbb{E}[\sigma(w^\top \mathbf{X})\sigma(w^\top \mathbf{X})^\top]$[29] .

This lemma ensures that the quadratic forms involving $\sigma(w^\top \mathbf{X})$ concentrate around their expectations, allowing us to replace random terms with deterministic equivalents.

Theorem 3.1.2.1. (Asymptotic Equivalent for $\mathbb{E}[\mathbf{Q}]$): Let Assumptions 1-3 hold and define $\bar{\mathbf{Q}}$ as

$$\bar{\mathbf{Q}} = \left(\frac{N}{n}\frac{\mathbf{\Phi}}{1+\delta} + \gamma I_n\right)^{-1}, \tag{3.5}$$

where δ is the unique positive solution to $\delta = \frac{1}{N}\text{tr}(\mathbf{\Phi}\bar{\mathbf{Q}})$. Then, for all $\epsilon > 0$, there exists $c > 0$ such that

$$\|\mathbb{E}[\mathbf{Q}] - \bar{\mathbf{Q}}\| \leq cn^{-1/2+\epsilon}$$

[29].

This theorem provides an asymptotic equivalent for the expectation of the resolvent $\mathbf{Q}$, allowing us to approximate $\mathbb{E}[\mathbf{Q}]$ with $\bar{\mathbf{Q}}$ in high-dimensional settings.

Using these results, we can derive the asymptotic approximations for the training and test mean squared errors:

Theorem 3.1.2.2. (Asymptotic Training Mean-Square Error): Let Assumptions 1-3 hold. Then, for all $\epsilon > 0$,

$$n^{1/2-\epsilon}(E_{\text{train}} - \hat{E}_{\text{train}}) \to 0 \quad \text{almost surely},$$

where

$$E_{\text{train}} = \frac{1}{N}\left\|\mathbf{Y}^\top - \mathbf{\Sigma}^\top\hat{\beta}\right\|_F^2 = \frac{\gamma^2}{N}\text{tr}\mathbf{Y}\mathbf{Q}^2(\gamma)\mathbf{Y}^\top,$$
$$\hat{E}_{\text{train}} = \frac{\gamma^2}{N}\text{tr}\left(\mathbf{Y}^\top\mathbf{Y}\bar{\mathbf{Q}}\left(\frac{1}{n}\text{tr}(\mathbf{\Phi}\bar{\mathbf{Q}}^2)(I_N - \frac{1}{n}\text{tr}(\mathbf{\Phi}^2\bar{\mathbf{Q}}^2))^{-1}\right)\bar{\mathbf{Q}}\right) \tag{3.6}$$

[29].

Similarly, for the test mean squared error:

Conjecture 3.1.2.1. ***Deterministic Equivalent for*** E_{test}*: Let Assumptions 1-2 hold. Then, for all* $\epsilon > 0$*,*

$$n^{1/2-\epsilon}(E_{test} - \hat{E}_{test}) \to 0 \quad \textit{almost surely},$$

where

$$E_{test} = \frac{1}{\hat{n}}\left\|\hat{\mathbf{Y}}^\top - \hat{\mathbf{\Sigma}}^\top\hat{\beta}\right\|_F^2,$$
$$\hat{E}_{test} = \frac{1}{\hat{n}}\left\|\hat{\mathbf{Y}}^\top - \hat{\mathbf{\Phi}}^\top\bar{\mathbf{Q}}\mathbf{Y}^\top\right\|_F^2 + \frac{1}{n}tr(\mathbf{Y}^\top\mathbf{Y}\bar{\mathbf{Q}}\mathbf{\Phi}\bar{\mathbf{Q}})\left(1 - \frac{1}{n}tr(\mathbf{\Phi}\bar{\mathbf{Q}})\right)^{-1} \tag{3.7}$$

[29].

Kernel-Based Reformulation of Asymptotic Errors Based on our previous considerations, we can express the results in terms of the limiting kernel $\mathbf{K}$, which provides a deterministic framework for understanding the behavior of the model. The kernel $\mathbf{K}$ effectively captures the feature mapping induced by the activation function σ and is pivotal in characterizing the asymptotic properties of the model's performance. This approach leverages **Theorem 3.1.2.1** and **Conjecture 3.1.2.1**, which demonstrate how the effective kernel $\bar{\mathbf{K}}$ influences the training and test errors.

To compute the effective kernel $\bar{\mathbf{K}}$, we employ the fixed-point equation derived from **Theorem 3.1.2.1**, which is iteratively solved to obtain:

$$\bar{\mathbf{K}} = \frac{N}{n}\frac{\mathbf{K}}{1+\delta}, \tag{3.8}$$

where δ is defined as:

$$\delta = \frac{1}{N}\text{tr}(\mathbf{K}\bar{\mathbf{Q}}).$$

Derivation of Asymptotic Errors with $\mathbf{K}$ The asymptotic training mean squared error E_{train} is given by:

$$E_{\text{train}} = \frac{1}{N}\left\|\mathbf{Y}^\top - \mathbf{\Sigma}^\top\beta\right\|_F^2 = \frac{\gamma^2}{N}\text{tr}(\mathbf{Y}\mathbf{Q}^2\mathbf{Y}^\top), \tag{3.9}$$

which can be rewritten using the kernel as:

$$\bar{E}_{\text{train}} = \frac{\gamma^2}{n}\text{tr}\left(\mathbf{Y}\bar{\mathbf{Q}}\left(\frac{\frac{1}{N}\text{tr}(\bar{\mathbf{Q}}\bar{\mathbf{K}}\bar{\mathbf{Q}})}{1 - \frac{1}{N}\text{tr}(\bar{\mathbf{K}}\bar{\mathbf{Q}}\bar{\mathbf{K}}\bar{\mathbf{Q}})}\bar{\mathbf{K}} + \mathbf{I}_n\right)\bar{\mathbf{Q}}\mathbf{Y}^\top\right) \tag{3.10}$$

where:

$$\bar{\mathbf{Q}} = \left(\frac{N}{n}\frac{\bar{\mathbf{K}}}{1+\delta} + \gamma\mathbf{I}_n\right)^{-1}.$$

Similarly, the test mean squared error E_{test} is expressed as:

$$E_{\text{test}} = \frac{1}{\hat{n}} \left\| \hat{\mathbf{Y}}^\top - \hat{\mathbf{\Sigma}}^\top \beta \right\|_F^2, \tag{3.11}$$

and can be reformulated in terms of the kernel:

$$\bar{E}_{\text{test}} = \frac{1}{\hat{n}} \left\| \hat{\mathbf{Y}}^\top - \bar{\mathbf{K}}_{\mathbf{X}\hat{\mathbf{X}}}^\top \bar{\mathbf{Q}} \mathbf{Y}^\top \right\|_F^2 + \frac{\frac{1}{N}\text{tr}\left(\mathbf{Y}\bar{\mathbf{Q}}\bar{\mathbf{K}}\bar{\mathbf{Q}}\mathbf{Y}^\top\right)}{1 - \frac{1}{N}\text{tr}\bar{\mathbf{K}}\bar{\mathbf{Q}}\bar{\mathbf{K}}\bar{\mathbf{Q}}} \left(\frac{1}{\hat{n}} \text{tr}\bar{\mathbf{K}}_{\hat{\mathbf{X}}\hat{\mathbf{X}}} - \frac{1}{\hat{n}} \text{tr}(\mathbf{I}_n + \gamma\bar{\mathbf{Q}})(\bar{\mathbf{K}}_{\mathbf{X}\hat{\mathbf{X}}}^\top \bar{\mathbf{K}}_{\mathbf{X}\hat{\mathbf{X}}} \bar{\mathbf{Q}}) \right) \tag{3.12}$$

where the effective kernels are defined as:

$$\mathbf{K}_{\mathbf{XX}} = \mathbb{E}[\sigma(\mathbf{X}^\top \mathbf{w})\sigma(\mathbf{w}^\top \mathbf{X})],$$

$$\mathbf{K}_{\mathbf{X}\hat{\mathbf{X}}} = \mathbb{E}[\sigma(\mathbf{X}^\top \mathbf{w})\sigma(\mathbf{w}^\top \hat{\mathbf{X}})],$$

$$\mathbf{K}_{\hat{\mathbf{X}}\hat{\mathbf{X}}} = \mathbb{E}[\sigma(\hat{\mathbf{X}}^\top \mathbf{w})\sigma(\mathbf{w}^\top \hat{\mathbf{X}})],$$

and their normalized counterparts:

$$\bar{\mathbf{K}} = \frac{N}{n} \frac{\mathbf{K}}{1+\delta},$$

$$\bar{\mathbf{K}}_{\mathbf{X}\hat{\mathbf{X}}} = \frac{N}{n} \frac{\mathbf{K}_{\mathbf{X}\hat{\mathbf{X}}}}{1+\delta},$$

$$\bar{\mathbf{K}}_{\hat{\mathbf{X}}\hat{\mathbf{X}}} = \frac{N}{n} \frac{\mathbf{K}_{\hat{\mathbf{X}}\hat{\mathbf{X}}}}{1+\delta}.$$

The kernel $\mathbf{K}$ replaces the need for direct computation of the matrix $\mathbf{\Sigma}$ by leveraging deterministic equivalents, which simplify the complexity of high-dimensional analysis. This methodology aligns with the results derived in [6], where the effective kernel $\bar{\mathbf{K}}$ is shown to have a significant impact on regression performance, providing a reliable approximation of the model's behavior in the asymptotic regime. By applying these insights, we can gain a deeper understanding of the model's performance and address potential issues such as overfitting by examining the interaction between p/n, N/n, and the choice of activation functions.

In summary, these results provide a rigorous foundation for approximating the training and test mean squared errors in high-dimensional settings. By using the deterministic equivalents $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$, we can effectively handle the complexity introduced by the randomness of $\mathbf{\Sigma}$ and gain insights into the asymptotic behavior of the error terms [29].

3.1.3. Variability and Extensions of the Model

In this subsection different parameters and model arrangement are going to be considered in order to explore the possible variability of the model. Exploring various aspects of the neural network model

is crucial for understanding its performance and behavior. Modifying different parameters and functions, such as activation functions, kernel choices, and ratios like $\frac{p}{n}$ and $\frac{N}{n}$, can significantly impact the model's ability to generalize and avoid overfitting. Moreover, it will be later shown in the results section that our analysis leads to different results, which vary substantially with changes in the parameters.

Different Activation Functions and Kernels Activation functions play a pivotal role in defining the nonlinear transformations within the neural network. Commonly used activation functions include ReLU, sigmoid, tanh, and others, each impacting the network's behavior differently. These functions lead to different kernel functions, influencing how data is mapped and processed. For instance, the ReLU activation function corresponds to a specific kernel, while other functions like sigmoid and tanh result in distinct kernels.

Different activation functions lead to different kernels, impacting the behavior and performance of the neural network. For a given dataset $\mathbf{X}$, it is possible to compute the "limiting" kernel $\mathbf{K}$ for the listed activation functions $\sigma(\cdot)$ using theoretical results. By iterating the fixed-point equation presented in Theorem 3.1.2.1, one can derive the effective kernel $\bar{\mathbf{K}} \equiv \frac{N}{n}\frac{\mathbf{K}}{1+\delta}$ in practical settings where n, p, N are large. This effective kernel provides insights into the regression performance and is advantageous as it applies to deterministic input data $\mathbf{X}$ rather than relying on randomly modeled data. Table 3.1 from [6] lists the limiting kernels for various activation functions, highlighting the diverse impact these functions have on the model:

$\sigma(t)$	$\kappa(\mathbf{x}, \mathbf{y})$
t	$\mathbf{x}^\top \mathbf{y}$
$\lvert t\rvert$	$\frac{2}{\pi}\lVert\mathbf{x}\rVert \cdot \lVert\mathbf{y}\rVert \left(\mathcal{L} \cdot \arcsin(\mathcal{L}) + \sqrt{1-\mathcal{L}^2}\right)$
$\mathsf{ReLU}(t) \equiv \max(t, 0)$	$\frac{1}{2\pi}\lVert\mathbf{x}\rVert \cdot \lVert\mathbf{y}\rVert \left(\mathcal{L} \cdot \arccos(-\mathcal{L}) + \sqrt{1-\mathcal{L}^2}\right)$
$a_+ \max(t, 0) + a_- \max(-t, 0)$	$\frac{1}{2}(a_+^2 + a_-^2)\mathbf{x}^\top \mathbf{y} + \frac{1}{2\pi}\lVert\mathbf{x}\rVert \cdot \lVert\mathbf{y}\rVert \left((a_+ + a_-)^2 \left(-\mathcal{L} \cdot \arccos(\mathcal{L}) + \sqrt{1-\mathcal{L}^2}\right)\right)$
$a_2 t^2 + a_1 t + a_0$	$a_2^2 \left(2(\mathbf{x}^\top \mathbf{y})^2 + \lVert\mathbf{x}\rVert^2 \lVert\mathbf{y}\rVert^2\right) + a_1^2 \mathbf{x}^\top \mathbf{y} + a_2 a_0 \left(\lVert\mathbf{x}\rVert^2 + \lVert\mathbf{y}\rVert^2\right) + a_0^2$
$\mathsf{erf}(t)$	$\frac{2}{\pi} \arcsin\left(\frac{2\mathbf{x}^\top \mathbf{y}}{\sqrt{(1+2\lVert\mathbf{x}\rVert^2)(1+2\lVert\mathbf{y}\rVert^2)}}\right)$
$\mathbf{1}_{t>0}$	$\frac{1}{2} - \frac{1}{2\pi} \arccos(\mathcal{L})$
$\mathsf{sign}(t)$	$\frac{2}{\pi} \arcsin(\mathcal{L})$
$\cos(t)$	$\exp\left(-\frac{1}{2}(\lVert\mathbf{x}\rVert^2 + \lVert\mathbf{y}\rVert^2)\right) \cosh(\mathbf{x}^\top \mathbf{y})$
$\sin(t)$	$\exp\left(-\frac{1}{2}(\lVert\mathbf{x}\rVert^2 + \lVert\mathbf{y}\rVert^2)\right) \sinh(\mathbf{x}^\top \mathbf{y})$
$\exp(-t^2/2)$	$\frac{1}{\sqrt{(1+\lVert\mathbf{x}\rVert^2)(1+\lVert\mathbf{y}\rVert^2) - (\mathbf{x}^\top \mathbf{y})^2}}$

Table 3.1: Limiting kernel $\kappa(\mathbf{x}, \mathbf{y}) = \mathbb{E}[\sigma(\mathbf{w}^\top \mathbf{x})\sigma(\mathbf{w}^\top \mathbf{y})]$ for standard Gaussian $\mathbf{w}$, with $\mathcal{L} \equiv \frac{\mathbf{x}^\top \mathbf{y}}{\lVert\mathbf{x}\rVert \cdot \lVert\mathbf{y}\rVert}$.

Variability with Increasing Layers and Loss Functions Beyond activation functions and kernel selections, the variability of the model's architecture itself, such as the number of layers, plays a crucial role in determining performance. Adding more layers can introduce additional complexity and depth, allowing the network to capture intricate patterns and interactions within the data. However, this complexity must be balanced with the risk of overfitting, particularly in high-dimensional settings where the ratio p/n and N/n can exacerbate model instability.

In the upcoming results section, we will present experiments that demonstrate the impact of different architectural configurations on the model's performance. This will include analyses of varying the number of layers and other hyperparameters to provide insight into how such modifications can be optimally tuned for specific tasks.

3.2. Experimental Setup

For the initial testing, we consider a one-hidden-layer neural network with Gaussian weights and a linear activation function for simplicity. The goal is to study the behavior of this network in relation to the training and test errors, particularly when regularizing the model using the ridge regression penalty.

In our setup, the input matrix X (both for training and test sets) is generated as a Gaussian random matrix with independent standard normal entries. We use the following specific steps for data generation:

- The weight matrix $\mathbf{W}$ is Gaussian, meaning each entry of $\mathbf{W} \in \mathbb{R}^{N \times p}$ is drawn from $\mathcal{N}(0, 1)$, where N represents the number of neurons in the hidden layer and p is the input dimensionality.
- The target vector $\mathbf{b}$ is generated randomly with binary entries, where each entry takes a value of 0 or 1 according to a certain density parameter. For instance, a density of 90% means approximately 90% of the entries of $\mathbf{b}$ are set to 1 and the rest to 0.
- The output vector $\mathbf{Y}$ is generated according to the relationship $\mathbf{Y} = \mathbf{\Sigma}^\top \mathbf{b} + \epsilon$, where $\mathbf{\Sigma} = \sigma(\mathbf{WX})$ is the activation matrix, $\mathbf{X}$ represents the input matrix, and ϵ is a Gaussian noise term with a certain noise level.

For the ridge regression model, we aim to find the optimal regularization parameter γ that minimizes both the training and test errors. To estimate the training error E_{train}, we calculate it as the squared difference between the observed training outputs $\mathbf{Y}_{\text{train}}$ and the predictions of the model. Similarly, the test error E_{test} is computed on the unseen test data. Additionally, deterministic equivalents of these errors, $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$, are obtained using approximations from random matrix theory, which simplify the analysis of the model's behavior.

For this setup, we use the following values for the key parameters:

- p/n ratio (a parameter under investigation), denoted as c_1, where p is the input dimension and n is the number of training data points.
- N/n ratio (another parameter under investigation), denoted as c_2, where N is the number of neurons in the hidden layer.
- A range of values for n (number of data points) - for the initial experiment taken as values from 100 to 3000, increasing in increments of 200.
- A binary target vector $\mathbf{b}$ generated with a certain density value (parameter under investigation).
- A range of regularization parameters γ explored, from 10^{-7} to 10^2.

We explore these settings using both a Monte Carlo method (random-based computations of E_{train} and E_{test}) and deterministic equivalents based on approximations for the kernel matrix $\mathbf{K}$.

The choice of the regularization parameter γ was made by examining the range in which the most interesting behavior occurs. As shown in the next section (Figure X.X), the test and training errors exhibit significant changes in behavior when γ is between 10^{-7} and 10^2. This range captures the transition where the errors are minimized and gives us insights into the model's performance. Consequently, we focus on this interval to explore the optimal value of γ.

Regarding the choice of data point values n, we selected the range of 100 to 3000 to fit within the computational resources available on the Delft hypercomputer cluster. Larger values of n were not feasible due to resource limitations, and this range was found to be sufficient to observe the desired trends in the model's behavior.

3.3. Algorithm Optimization

This section describes how we transform a straightforward ("naive") implementation of the pipeline into an optimized one that is both faster and numerically more stable. We assume the reader is familiar with the experimental setup and notation from the preceding sections: given standardized data $(X_{\mathrm{tr}}, X_{\mathrm{te}})$, labels $y_{\mathrm{tr}}, y_{\mathrm{te}} \in \{-1, +1\}$, a random-features matrix $W \in \mathbb{R}^{N\times p}$, and an activation $\sigma(\cdot)$, we evaluate two one–dimensional calibration rules for the ridge penalty γ-Coulliet's deterministic-equivalent test proxy $\overline{E}_{\mathrm{test}}(\gamma)$ (to be discussed in later chapters, cf. Eq. (3.12)) and our direct-loss rule based on $\partial_\gamma \mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma)$ (cf. Eq. (4.3b))—and then form $\widehat{\beta}(\gamma)$ via the closed-form ridge expression (Eq. (3.1)).

3.3.1. Naive baseline: what is expensive

In a direct implementation, for *each* grid value γ one typically:

1. Recomputes kernel blocks $K_{\mathrm{tr}} = K(X_{\mathrm{tr}}, X_{\mathrm{tr}})$, $K_{xX} = K(X_{\mathrm{tr}}, X_{\mathrm{te}})$, $K_{XX} = K(X_{\mathrm{te}}, X_{\mathrm{te}})$;
2. Solves the fixed point for $\delta = \delta(\gamma)$ by repeatedly forming and inverting

$$Q(\gamma, \delta) \;=\; \frac{N}{n}\,\frac{K_{\mathrm{tr}}}{1+\delta} \;+\; \gamma I_n;$$

3. Evaluates $\overline{E}_{\mathrm{test}}(\gamma)$ and/or $\partial_\gamma \mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma)$ using dense $n \times n$ linear algebra;
4. Forms the ridge solution $\widehat{\beta}(\gamma) = \frac{\Sigma}{n}\big(\frac{1}{n}\Sigma^\top\Sigma + \gamma I\big)^{-1} y$, where $\Sigma = \sigma(WX_{\mathrm{tr}})$.

Steps (2)-(4) perform an $n \times n$ factorization/inversion *per* γ. In perturbation studies, repeating the same work per scenario quickly dominates runtime and accumulates numerical error.

3.3.2. Optimized pipeline: what we change

We refactor the computation so that all operations that do not depend on γ (and, where possible, that do not depend on the scenario) are computed once and then reused. The concrete changes are:

(O1) Precompute kernels once and reuse. Compute the Gram blocks

$$K_{\mathrm{tr}}, \quad K_{xX}, \quad K_{XX}$$

once per dataset (or once per scenario when the training inputs are perturbed). The arc–cosine kernel implementation is numerically stabilized via norm products clamped away from 0 and cosine arguments clipped to $(-1, 1)$.

(O2) Eigendecompose K_{tr} once; diagonalize the fixed–point map. Write $K_{\mathrm{tr}} = V\Lambda V^\top$ with $\Lambda = \mathrm{diag}(\lambda_i)$. The fixed–point iteration

$$\delta \;\mapsto\; \frac{1}{n}\sum_{i=1}^{n} \frac{\lambda_i}{a(\delta)\lambda_i + \gamma}, \qquad a(\delta) = \frac{N/n}{1+\delta},$$

no longer requires forming Q^{-1}; it becomes a cheap diagonal update using only $\{\lambda_i\}$. All $\overline{E}_{\mathrm{test}}(\gamma)$ terms that involve traces with $Q(\gamma,\delta)^{-1}$ reduce to sums over $d_i(\gamma,\delta) = a(\delta)\lambda_i + \gamma$.

(O3) SVD trick for random features; avoid matrix inverses. Let $A = \Sigma/\sqrt{n}$ with thin SVD $A = USV^\top$, $S = \mathrm{diag}(s_i)$ $(i = 1, \ldots, r)$. Then

$$\Big(\tfrac{1}{n}\Sigma^\top\Sigma + \gamma I\Big)^{-1} \;=\; V\,\mathrm{diag}\Big(\tfrac{1}{s_i^2+\gamma}\Big)V^\top \;+\; \frac{1}{\gamma}P_{\mathrm{null}},$$

so that both $\widehat{\beta}(\gamma)$ and the direct derivative

$$\partial_\gamma \mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma) = \gamma\left(\sum_{i=1}^{r} \frac{(v_i^\top y)^2}{(s_i^2+\gamma)^3} + \frac{\|y - P_{\mathrm{span}} y\|_2^2}{\gamma^3}\right) - \widehat{\sigma}^2_{\mathrm{RSS}}(\gamma)\left(\sum_{i=1}^{r} \frac{1}{(s_i^2+\gamma)^2} + \frac{n-r}{\gamma^2}\right)$$

are computed by *elementwise* operations on $\{s_i^2\}$. The RSS variance is computed consistently from the same SVD:

$$\widehat{\sigma}^2_{\mathrm{RSS}}(\gamma) = \frac{1}{n}\left\| \frac{\gamma}{S^2+\gamma} V^\top y \right\|_2^2 + \frac{1}{n}\|y - P_{\mathrm{span}} y\|_2^2.$$

(O4) Factor once, solve many. Cholesky/triangular solves replace explicit inverses everywhere (when we operate in the $n \times n$ domain). In the RKHS route, all per-γ work lives on the diagonal $\{d_i\}$; in the random features route, all per-γ work lives on $\{s_i^2\}$.

(O5) Grid evaluation by broadcasting. We evaluate $\overline{E}_{\mathrm{test}}(\gamma)$ and $\partial_\gamma \mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma)$ on the full log–grid using vectorized broadcasts over $d_i(\gamma, \delta)$ or $s_i^2 + \gamma$, thereby avoiding Python loops.

(O6) Caching across scenarios. We keep W fixed and reuse all W–dependent quantities (e.g. shapes, normalization). For Coulliet's route, only $(K_{\mathrm{tr}}, K_{xX}, K_{XX})$ and their eigenpairs change when training inputs are perturbed.

3.3.3. Complexity at a glance

Let n be the train size, $|\mathcal{G}|$ the grid length, and $r = \mathrm{rank}(A) \le \min\{N, n\}$.

- **Naive**: $O(|\mathcal{G}|\, n^3)$ for repeated inversions in the δ loop and in $\beta(\gamma)$ per grid point, plus kernel recomputation per γ.
- **Optimized**:
  - One eigendecomposition of K_{tr}: $O(n^3)$.
  - One SVD of A: $O(nr^2)$ or $O(Nr^2)$ (economy SVD).
  - Per γ: $O(n)$ for Coulliet (diagonal arithmetic on $\{d_i\}$), $O(r)$ for Direct (on $\{s_i^2\}$).
  - Per scenario: recompute kernels and one eigendecomposition of K_{tr}; reuse all grid–wise vectorization and SVD formulas.

In practice this reduces wall-clock time by one to two orders of magnitude for the grids and scenario counts considered, while improving numerical stability by avoiding explicit inverses.

3.3.4. Practical notes

- **Storage.** We store baseline and per–scenario betas/gammas in compressed NPZs. Reusing W ensures fair, paired comparisons across scenarios.
- **Determinism.** A single seed controls scenario design and W; results reported in the main text do not depend on the particular seed choice in any qualitative way.

Takeaway. The optimized pipeline moves all γ-dependence to diagonal arithmetic on spectral quantities, eliminates explicit inverses, and restricts per scenario searches to local grids. This yields substantial speedups and improves numerical behavior, especially critical in the perturbation analysis, where many closely related problems must be solved repeatedly.

3.4. Problem Arising

To briefly recap: in the preceeding sections we have explicitly derived the expressions for E_{train} (3.2) and E_{test} (3.4), which represent training and test mean squared errors (MSE) respectively, in the high-dimensional regime. By using results from random matrix theory we formulated deterministic equivalents $\hat{E}_{\text{train}}$ (3.10) and $\hat{E}_{\text{test}}$ (3.12) for E_{train} and E_{test}. These equivalents allow for the approximation of these error metrics in scenarios where the number of data points n, the number of features p, and the number of neurons N are large. Approximations themselves depend on the resolvent $\hat{\mathbf{Q}}$ (3.5), which captures the effects of randomness inherent in the data through the Gram matrix and the kernel matrix. From there we have found an optimal value of hyperparameter γ that allows to optimize expressions for both $\hat{E}_{\text{train}}$ (3.10) and $\hat{E}_{\text{test}}$ (3.12).

However, a critical issue arises when comparing the optimal γ values obtained through this approach with those obtained through direct minimization of the ridge loss function, which in our case we dxefined as:

$$\mathcal{L}_{ridge} = \frac{\|\hat{\beta} - b\|_F^2}{\|b\|_F^2}, \tag{3.13}$$

where b is the target vector, and $\hat{\beta}$ is the solution obtained from ridge regression: (3.1):

$$\hat{\beta} = \frac{1}{n}\mathbf{\Sigma}\left(\frac{1}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top + \gamma \mathbf{I}_n\right)^{-1}\mathbf{Y}^\top,$$

This metric evaluates how well the predicted coefficients match the true target vector, with the goal of minimizing the relative Frobenius norm error.

The core of the problem lies in the discrepancy between the optimal γ values obtained by minimizing $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$ (derived from random matrix theory approximations) and the optimal γ obtained by minimizing the ridge loss function directly. Ideally, one would expect that these values converge or align to some degree, as both approaches aim to minimize the model error. However, the experimental results suggest otherwise, showing significant differences in the γ values that yield the best performance for each case. We are going to see the details of this discrepancy in the results sections further.

3.4.1. Potential cause of the problem

One potential cause of this discrepancy could be rooted in the assumptions and approximations used in deriving $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$. The use of random matrix theory allows us to simplify the problem by averaging over the randomness in the data, which can smooth out some variability that is captured more directly in the ridge loss function. As a result, the deterministic equivalents may not fully capture all aspects of the data structure or noise present in real-world data.

Another contributing factor could be the simplified setup used for the target vector b, which is binary (0s and 1s) with a fixed density. While this setup is useful for initial theoretical testing, it may not reflect the complexity of more general real-world data, where target vectors can exhibit more variability and noise. The binary nature of b could limit the flexibility of the model, potentially influencing the optimization of γ in different ways when using the ridge loss function versus the random matrix theory approximations.

Finally, the inherent differences between the MSE-based training and test errors and the ridge loss function itself may also play a role. The ridge loss function focuses on minimizing the difference between the predicted coefficients $\hat{\beta}$ and the true target vector b, while the MSE errors focus on minimizing the prediction errors for the outputs. This distinction may lead to divergent results in terms of what constitutes the ”optimal” value for γ.

4

Results

In the following chapter we will present the results of our experiments and findings regarding the relationship between the regularization parameter γ, Mean Squared Error , ridge loss function and various model parameters.

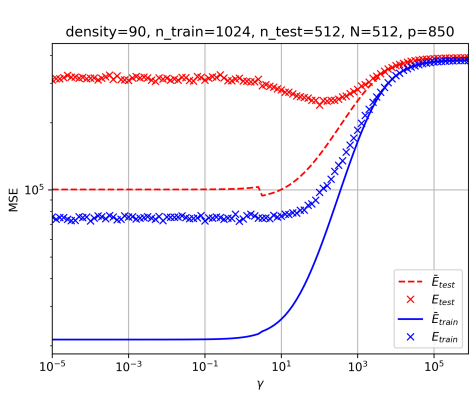
4.1. Research on the original Experiment

In this section we interrogate the simplest controllable setting in which our theory can be tested: data generated exactly according to the "synthetic law" $\mathbf{Y} = \mathbf{\Sigma}^\top \mathbf{b} + \varepsilon$ with a *known* ground-truth coefficient vector b. The benchmark serves three purposes. First, it allows us to *reproduce* the Gaussian-W baseline of Couillet & Liao [6] and recover the random-matrix-theory optimal regularisation γ^*_{RMT}. Second, by comparing this predictive optimum with the direct alignment loss $\mathcal{L}_{\text{ridge}}$, we expose the *causal gap*—the systematic misalignment between γ^*_{RMT} and the value that best recovers b (Section 4.1.4). Finally, sweeping three aspect ratios $p/n \in \{0.7, 1.0, 2.0\}$ and three sparsity levels of b (10 %, 50 %, 90 %), we chart the regimes where this gap widens or narrows, providing a roadmap for the more complex real-data experiments that follow.

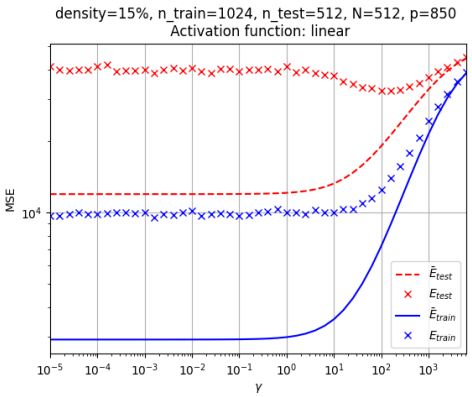
4.1.1. Mean Squared Error against the regularization parameter γ

We will begin with replicating the initial experiment from the book [6], investigating the Mean Squared Error values for both E_{train} (3.9) , E_{test} (3.11) as well as $\hat{E}_{\text{train}}$ (3.10) , $\hat{E}_{\text{test}}$ (3.12) against a ridge regularization parameter γ. The goal is to observe the trends in the MSE for both the training and test sets as γ changes, and to identify the optimal γ that minimizes the test error and whether it is possible in general.

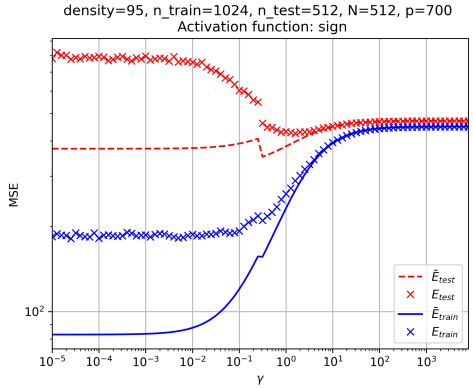
In order to illustrate the impact of different activation functions (linear, ReLU, and signum) and varying levels of the target vector density (15% versus 90%), we present in Figure 4.1 six separate plots. Each sub-figure shows the training and test MSE as a function of the ridge parameter γ. As expected, the linear, ReLU, and signum activations exhibit similar qualitative "U-shapes" in the test-error curves, though at different scales depending on density and nonlinearities. The graph presented here shows a characteristic behavior: the training error increases as γ increases, while the test error initially decreases, reaches a minimum, and then begins to increase again as γ becomes too large. This result aligns with the conclusion from the book that the hyperparameter γ can be fine-tuned to optimize test performance, given that n, p, and N are not too small and comparable in magnitude.



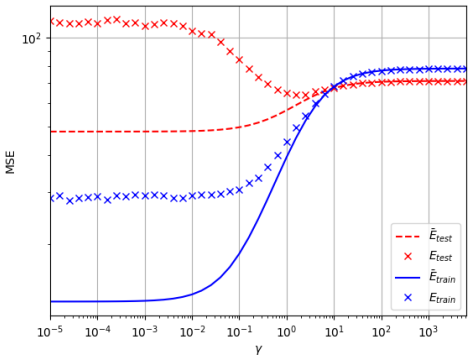
(a) Linear, 90% density



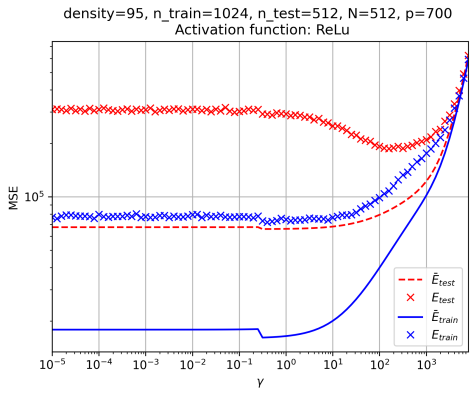
(b) Linear, 15% density



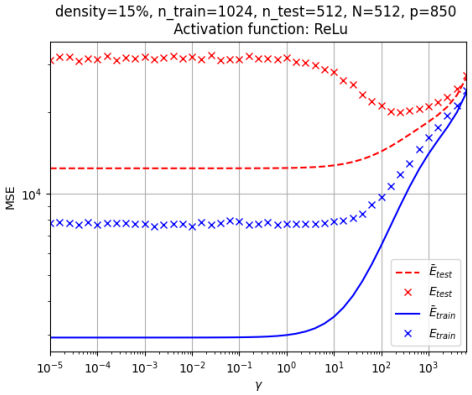
(c) Signum, 90% density



(d) Signum, 15% density



(e) ReLU, 90% density



(f) ReLU, 15% density

Figure 4.1: Training and test MSE versus the regularization parameter γ for three activation functions (linear, signum, ReLU) under two different densities of the target vector (90% and 15%). Each subfigure exhibits the typical U-shape in test error, alongside a rising training error as γ increases.

4.1.2. Optimal γ value and error behaviour with increasing amount of data

In this subsection, we investigate the behavior of the mean squared errors (E_{test} and $\hat{E}_{\text{test}}$) and their corresponding optimal γ values as the number of data points n increases, while keeping the ratios p/n and N/n fixed. The parameters have been chosen such that p, N and n are sufficiently large and comparable in magnitude. This choice is motivated by the fact that in the high-dimensional regime, where the number of features p is approximately of the same order as the number of samples n, the theoretical results from random matrix theory hold.

The experiment is designed by keeping p/n and N/n fixed, while increasing n from 100 to 3500 in increments. For each value of n, 80% of the data is used for training and 20% for testing. At each step, we search for the optimal regularization parameter γ from a grid ranging from 10^{-7} to 10^{1} (in increments of 0.1), which minimizes E_{test} and $\hat{E}_{\text{test}}$. These optimal γ values are then plotted against n, along with the corresponding values for the errors.

The following figures present the results of this experiment:

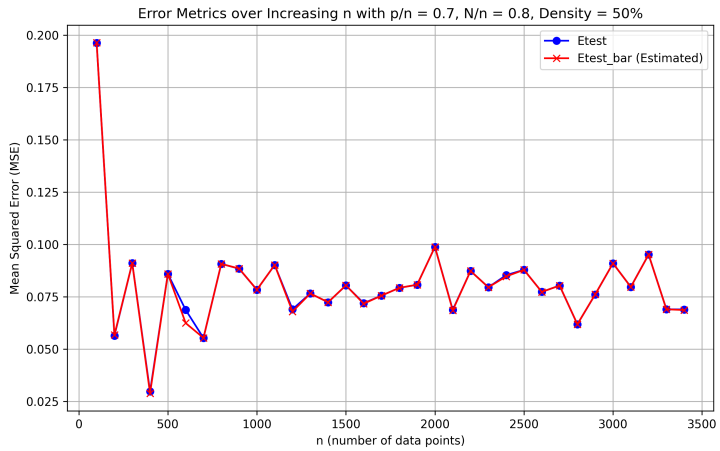


Figure 4.2: Error metrics (E_{test} and $\hat{E}_{\text{test}}$) over increasing n with $p/n = 0.7$, $N/n = 0.8$, and density = 50%.

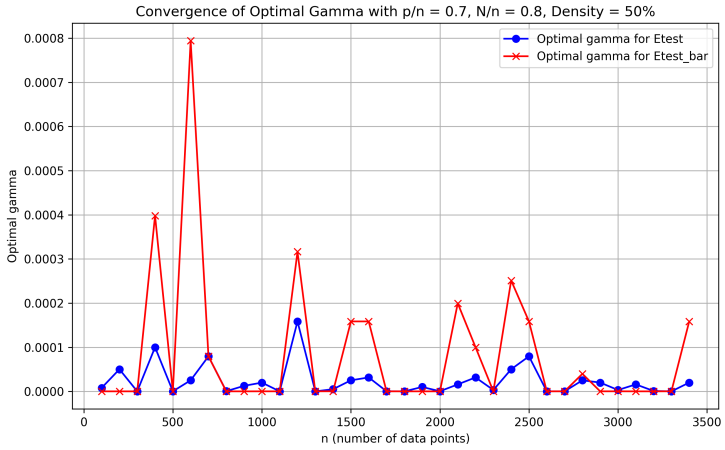


Figure 4.3: Convergence of optimal γ for E_{test} and $\hat{E}_{\text{test}}$ over increasing n with $p/n = 0.7$, $N/n = 0.8$, and density = 50%.

From the results in Figures 4.2 and 4.3, we observe that the errors themselves remain mostly stable as the number of data points increases, with slight fluctuations that can be attributed to the random nature of the data generation. However, the optimal values for γ show more variability. While there is

some fluctuation, there is also a noticeable trend suggesting that the optimal γ values for E_{test} and $\hat{E}_{\text{test}}$ show converging behaviour as n increases.

This empirical result aligns with the theoretical conjecture 3.1.2.1 shown earlier, which asserts that the difference between E_{test} and $\hat{E}_{\text{test}}$ should diminish as n increases.

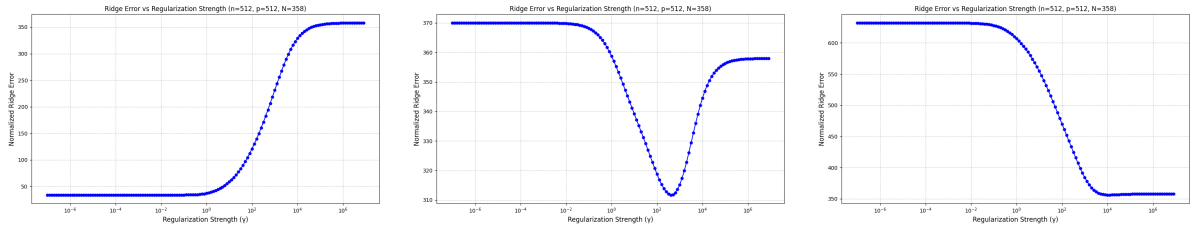
4.1.3. Optimal γ Values study for Different Loss Functions

In this subsection, we address a critical issue that arises in our investigation: the discrepancy between the optimal γ values needed to minimize different error metrics. The setting for this analysis remains consistent with the previous sections, where we explore the behavior of error metrics and optimal regularization parameters (γ) while increasing the number of data points (n), and keeping the ratios p/n and N/n fixed. As mentioned earlier, the experiment generates data according to the law $Y = \Sigma^\top b + \epsilon$, where b is a sparse vector, and Σ represents the network's internal operations. For each increment in n, we determine the optimal γ that minimizes both E_{test} and $\hat{E}_{\text{test}}$.

However, when we compare the optimal γ values required for minimizing the ridge loss function we defined earlier with those minimizing the test errors (E_{test} and $\hat{E}_{\text{test}}$), we observe a noticeable difference. As shown in the graphs below, the optimal γ for the ridge loss function does not always align with the optimal values for E_{test} and $\hat{E}_{\text{test}}$.

In this experiment, we also consider the "direct" loss function, given by Equation (3.13), which measures the Frobenius norm difference between the predicted coefficients $\hat{\beta}$ and the target vector b. As seen in the results, the optimal γ values obtained for minimizing this direct loss function differ from those obtained when optimizing the test errors, E_{test} and $\hat{E}_{\text{test}}$. The graphs clearly illustrate this difference, with the values for γ needed to minimize the direct loss function consistently diverging from the values required for minimizing the test errors. These results highlight the distinct behavior of the ridge loss function compared to the test errors derived from random matrix theory.

Effect of Regularization Strength on Ridge Regression Error (Different Densities). Figure 4.4 compares three plots of the normalized ridge error $\|\hat{\beta} - b\|_F^2 / \|b\|_F^2$ (3.13) versus γ under different densities for b. In the **middle panel** (approximately 50% density), we see a pronounced "U-shaped" curve with a clear global minimum. By contrast, the **left** and **right** panels (very high or very low density) show flatter or less sharply defined minima, suggesting that in those regimes, the ridge-regularized solution does not exhibit as strong a trade-off between underfitting and overfitting.



(a) Density ≈ 95% **(b)** Density ≈ 50% **(c)** Density ≈ 15%

Figure 4.4: Normalized ridge error $\|\hat{\beta} - b\|^2 / \|b\|^2$ as a function of the regularization strength γ (log scale) for three different densities. The center plot (50% density) exhibits a clear "U-shape," whereas the other two densities do not show a single, pronounced minimum.

4.1.4. Diagnosing the Causal Gap

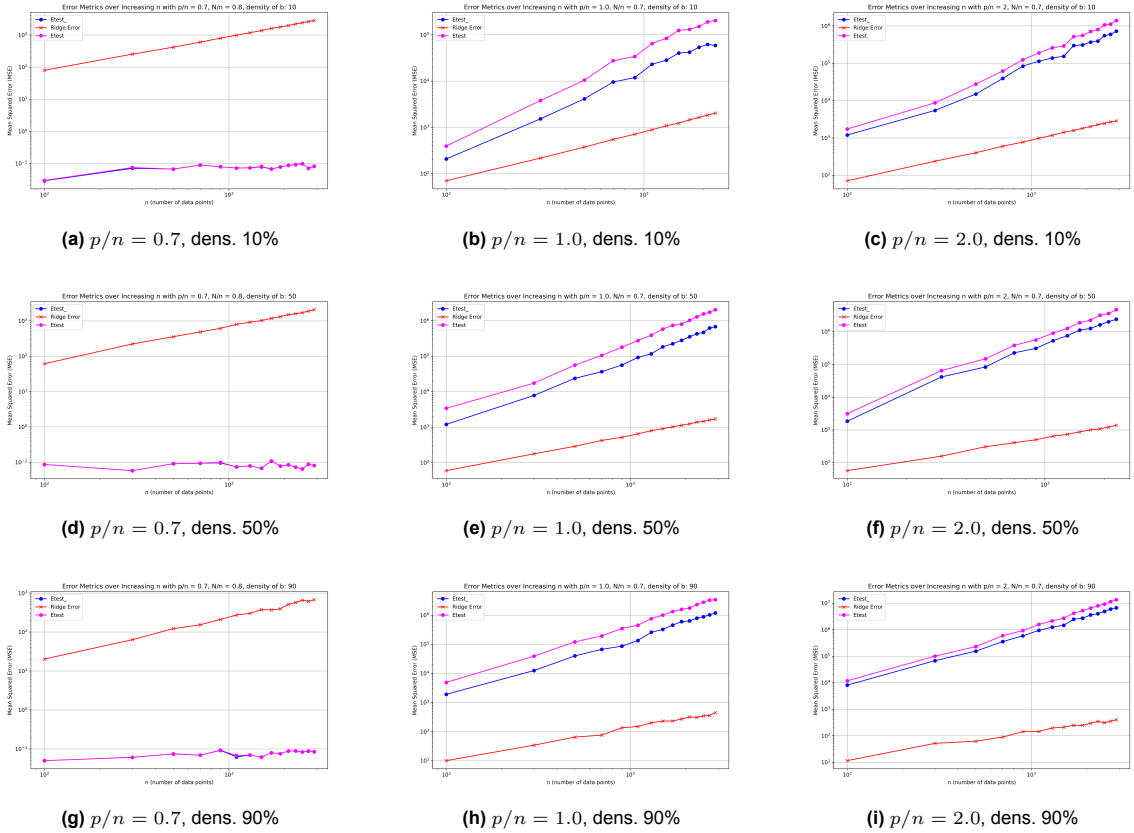
Having reproduced the MSE-based optimum γ^*_{RMT}, we now turn to the question of **causal alignment**: *does the predictive optimum also recover the ground-truth mechanism* b*?*[1] Concretely, we contrast three objective curves as a function of γ:

[1] Causal faithfulness here is operationalised by the direct alignment loss $\mathcal{L}_{\text{ridge}}$ introduced in (3.13).

- the **direct causal loss** $\mathcal{L}_{\text{ridge}}(\gamma)$ (3.13) (red dotted),
- the **RMT proxy** $\hat{E}_{\text{test}}(\gamma)$ (blue dotted), and
- the **empirical test MSE** $E_{\text{test}}(\gamma)$ (purple dotted).

The graphs below show the difference between the value of each objective curve once minimized by γ against the increasing number of points n. Different plots demonstrate different effects according to various values of p/n ratio as well as the density of an underlying true vector β.

Note that the grid of γ values from 10^{-7} to 10^{5} through 0.1 increment was used for the optimization in this experiment. For each objective curve, the value of the corresponding loss function has been calculated for all values of γ, then the minimum has been chosen.



(a) $p/n = 0.7$, dens. 10% **(b)** $p/n = 1.0$, dens. 10% **(c)** $p/n = 2.0$, dens. 10%

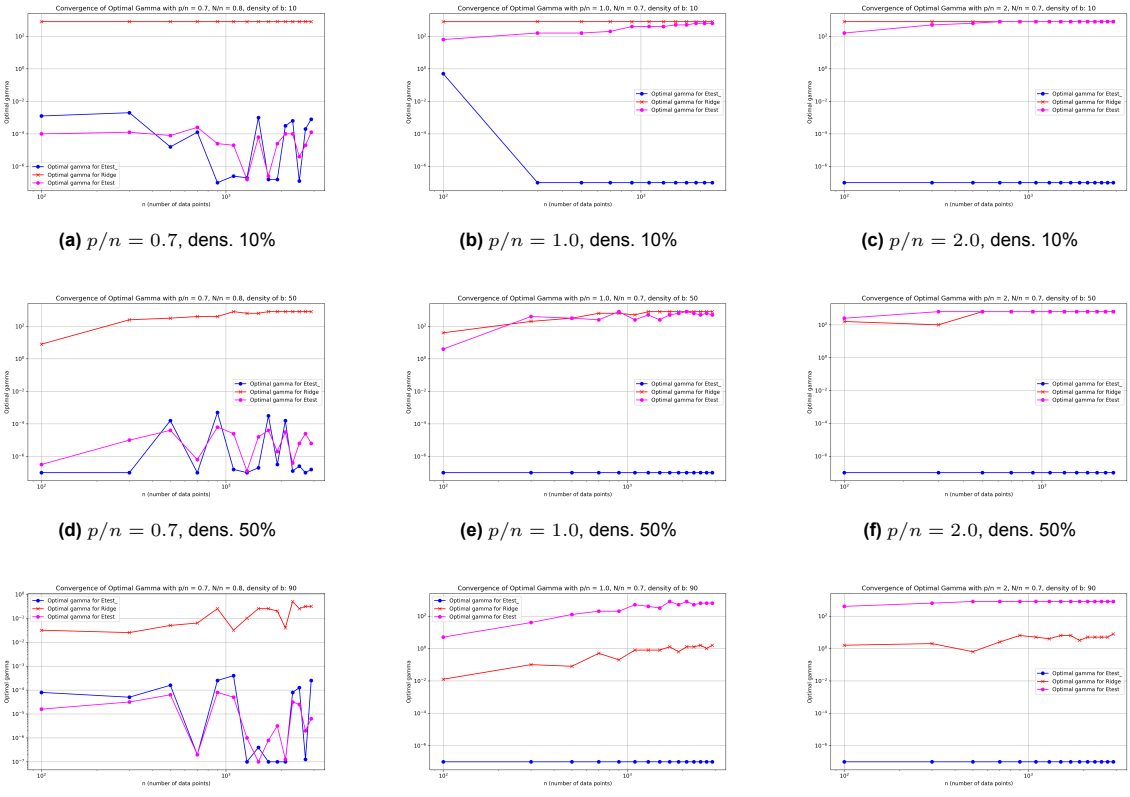
(d) $p/n = 0.7$, dens. 50% **(e)** $p/n = 1.0$, dens. 50% **(f)** $p/n = 2.0$, dens. 50%

(g) $p/n = 0.7$, dens. 90% **(h)** $p/n = 1.0$, dens. 90% **(i)** $p/n = 2.0$, dens. 90%

Figure 4.5: Direct causal loss $\mathcal{L}_{\text{ridge}}$ (3.13) (black), RMT-predicted $\hat{E}_{\text{test}}$ (blue dashed) and empirical E_{test} (red dotted) versus γ across nine synthetic regimes. Stars mark the minimisers of each curve.

We sweep nine synthetic regimes (Figure 4.5): three aspect ratios $p/n \in \{0.7,\ 1.0,\ 2.0\}$ (left–to–right columns) and three sparsity levels of b (10 %, 50 %, 90 %; top–to–bottom rows). Each panel marks with a star the γ that minimises the corresponding curve.

It also does seem prudent to look at the actual optimal γ values found through loss minimization. Below is the graph of all such γ values.



(a) $p/n = 0.7$, dens. 10% **(b)** $p/n = 1.0$, dens. 10% **(c)** $p/n = 2.0$, dens. 10%

(d) $p/n = 0.7$, dens. 50% **(e)** $p/n = 1.0$, dens. 50% **(f)** $p/n = 2.0$, dens. 50%

(g) $p/n = 0.7$, dens. 90% **(h)** $p/n = 1.0$, dens. 90% **(i)** $p/n = 2.0$, dens. 90%

Figure 4.6: Optimal regularisation strength γ^* as a function of sample size n for nine synthetic regimes. Each panel reports: γ_{Etest} (blue), γ_{Etest^*} (magenta), and the causal γ_{Ridge} (red) that minimises $\mathcal{L}_{\text{ridge}}$. Log–log axes highlight stability or drift across two orders of magnitude in n.

Sample–size sweep: predictive error vs. causal alignment. Figure 4.5 (error curves) and Figure 4.6 (corresponding γ^*'s) convey a consistent message across all nine regimes. First, the *predictive* quantities E_{test} and $\hat{E}_{\text{test}}$ track each other almost perfectly once $n \gtrsim 200$, confirming the accuracy of the RMT surrogate. Second – and crucial for our causality narrative – the *causal* loss $\mathcal{L}_{\text{ridge}}$ (red curve) behaves very differently: it remains one to three orders of magnitude lower than the predictive errors for $p/n < 1$ and sparse b, and, even in the over-parameterised cases (p/n=1, 2), it grows far more slowly.

The optimal regularisation strengths in Figure 4.6 expose why. The γ that minimises $\mathcal{L}_{\text{ridge}}$ (red $\times$) sits consistently around 10^{-1} – 10^{0} and varies little with n, whereas the predictive optima (blue and magenta dots) collapse toward $\sim 10^{-5}$ as n grows. Put plainly, *the model needs a substantially stronger penalty to recover the ground-truth coefficients than it does to minimise prediction error*. This divergence widens with higher sparsity and larger aspect ratio, exactly the regimes where Figure 4.5 showed the largest causal gap. Hence, tuning γ on E_{test} (or its RMT proxy) is not merely sub-optimal for causal recovery—it systematically pushes the solution toward a coefficient vector that mis-aligns with the data-generating mechanism. Bridging this gap is therefore essential, motivating the gradient-based correction developed in the next section.

4.2. Loss–Landscape Analysis

The causal gap identified in Section 4.1.4 raises a natural question: *how should the regularisation strength γ be adjusted to minimise the alignment loss $\mathcal{L}_{ridge}$ itself?* In this section, we take a differential view, deriving closed-form (or nearly closed-form) expressions for the gradient $\partial\mathcal{L}_{\text{ridge}}/\partial\gamma$ under the synthetic law. We first inspect the *oracle* gradient – available when the true variance components are known. And then introduce an empirical variance estimator that yields a practical, data-driven surrogate.

Let us start first with deriving a closed-form expression for $\mathcal{L}(\gamma)$ as in 3.13

Note first that synthetic data generation law has two representations: $\mathbf{Y} = \mathbf{\Sigma}^\top b + \epsilon^\top$ version, or $\mathbf{Y} = b^\top\mathbf{\Sigma} + \epsilon$). β regressor from 3.1 has then two equivalent representations according to a data-generation law:

$$\hat{\beta} = \frac{1}{n}(\frac{1}{n}\mathbf{\Sigma}\mathbf{\Sigma}^\top + \gamma\mathbf{I}_N)^{-1}\mathbf{\Sigma}\mathbf{Y}^\top, \text{ and}$$

$$\hat{\beta} = \frac{1}{n}\mathbf{\Sigma}\left(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\right)^{-1}\mathbf{Y}^\top.$$

We are going to proceed with the latter variant in our calculations. For the sake of simplicity, we are also going to omit the normalizing factor $||b||$ and assume $\mathcal{L}(\gamma) = ||\hat{\beta} - b||_F^2$.

Then the derivation goes as follows:

$$\mathcal{L}(\gamma) = ||\hat{\beta} - b||_F^2 = (\hat{\beta} - b)^\top(\hat{\beta} - b) = \hat{\beta}^\top\hat{\beta} - 2\hat{\beta}^\top b + b^\top b.$$

Uncovering the first term of that, we get:

$$\hat{\beta}^\top\hat{\beta} = \frac{1}{n^2}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma I_n)^{-1}\mathbf{\Sigma}^\top\mathbf{\Sigma}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma I_n)^{-1}\mathbf{Y}^\top$$

Then, let us rearrange the terms a little and then add and subtract γI_n from/to the central side. We thus obtain:

$$\begin{aligned}
\hat{\beta}^\top\hat{\beta} = \frac{1}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}((\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n) - \gamma\mathbf{I}_n)(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\mathbf{Y}^\top \\
= \quad \frac{1}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}(\mathbf{Y}^\top - \gamma(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\mathbf{Y}^\top) \\
= \quad \frac{1}{n}(\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\mathbf{Y}^\top - \gamma\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-2}\mathbf{Y}^\top)) \\
= \quad \frac{1}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\mathbf{Y}^\top - \frac{\gamma}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-2}\mathbf{Y}^\top.
\end{aligned}$$

Therefore we can write down a full expression for $\mathcal{L}(\gamma)$ as such:

$$\mathcal{L}(\gamma) = \frac{1}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\mathbf{Y}^\top - \frac{\gamma}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-2}\mathbf{Y}^\top - 2\frac{1}{n}\mathbf{Y}(\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)^{-1}\Sigma^\top b + b^\top b.$$

Next up, let us take the derivative of the expression above with respect to γ:

$$\frac{\partial \mathcal{L}(\gamma)}{\partial \gamma} = -\frac{1}{n}\,\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-2}\mathbf{Y}^\top - \Big(\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-2}\mathbf{Y}^\top - 2\gamma\,\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-3}\mathbf{Y}^\top\Big)$$
$$+ \quad 2\,\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-2}\mathbf{\Sigma}^\top b$$
$$= \quad -2\,\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-2}\mathbf{Y}^\top + 2\gamma\,\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-3}\mathbf{Y}^\top + 2\,\tfrac{1}{n}\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n\big)^{-2}\mathbf{\Sigma}^\top b.$$

Setting the derivative to zero and multiplying by $n/2$ yields:

$$\hat{\gamma}\,\mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \hat{\gamma}\mathbf{I}_n\big)^{-3}\mathbf{Y}^\top \;-\; \mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \hat{\gamma}\mathbf{I}_n\big)^{-2}\mathbf{Y}^\top \;+\; \mathbf{Y}\big(\tfrac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \hat{\gamma}\mathbf{I}_n\big)^{-2}\mathbf{\Sigma}^\top b \;=\; 0.$$

Among these terms the only unknown quantity is $\mathbf{\Sigma}^\top b$, which under the synthetic law $Y = \mathbf{\Sigma}^\top b + \varepsilon$ can be rewritten as $\mathbf{Y} - \varepsilon$.

In order to simplify further derivations, we will introduce some notation changes at this point. Note that in the experimental setting used in the central part of this thesis, target output $\mathbf{Y}$ is of the size $1 \times n$, and therefore is a vector. We will refer to a vector-shaped target output variable as $\mathbf{y}$ onward. Furthermore, let us introduce the following resolvent notation:

$$\mathbf{Q} \equiv (\frac{1}{n}\mathbf{\Sigma}^\top\mathbf{\Sigma} + \gamma\mathbf{I}_n)$$

Considering the above notation, the expression for the loss function derivative takes the following form:

$$\frac{\partial \mathcal{L}(\gamma)}{\partial \hat{\gamma}} = \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbf{y}\mathbf{Q}^{-2}\mathbf{y}^\top + \mathbf{y}\mathbf{Q}^{-2}\Sigma^\top b$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbf{y}\mathbf{Q}^{-2}\mathbf{y}^\top + \mathbf{y}\mathbf{Q}^{-2}(\mathbf{y}^\top - \epsilon^\top)$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbf{y}\mathbf{Q}^{-2}\mathbf{y}^\top + \mathbf{y}\mathbf{Q}^{-2}\mathbf{y}^\top - \mathbf{y}\mathbf{Q}^{-2}\epsilon^\top$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbf{y}\mathbf{Q}^{-2}\epsilon^\top \quad = 0.$$

Note that the only out-of-sample part of the expression above is $\mathbf{y}\mathbf{Q}^{-2}\epsilon^\top$, due to the unknown noise parameter ϵ. To find the optimal parameter $\hat{\gamma}$ we are to estimate that part. In order to do that, let us take the expectation of that expression with respect to ϵ, taking into account that by definition, ϵ is a mean-zero noise factor with some variance σ^2:

$$\mathbb{E}[\frac{\partial \mathcal{L}(\gamma)}{\partial \hat{\gamma}}|\epsilon]$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbb{E}[\mathbf{y}\mathbf{Q}^{-2}\epsilon^\top|\epsilon]$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbb{E}[(b^\top\mathbf{\Sigma} + \epsilon)\mathbf{Q}^{-2}\epsilon^\top|\epsilon]$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbb{E}[b^\top\Sigma\mathbf{Q}^{-2}\epsilon^\top + \epsilon\mathbf{Q}^{-2}\epsilon^\top|\epsilon]$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \mathbb{E}[\epsilon\mathbf{Q}^{-2}\epsilon^\top|\epsilon]$$
$$= \quad \hat{\gamma}\mathbf{y}\mathbf{Q}^{-3}\mathbf{y}^\top - \sigma^2\mathsf{tr}(\mathbf{Q}^{-2}) \quad = 0.$$

We thus obtain the ”oracle” expression for direct loss derivative. But in order to extract the optimal parameter from the formula - we would need to know the real variance value. Therefore, the question now is – how do we approximate the real out-of-sample noise variance σ^2. The most straightforward approach would be to consider the Residual Sum of Squares estimator:

$$\hat{\sigma^2}_{RSS} = \frac{(\mathbf{y} - \hat{\mathbf{y}})^\top (\mathbf{y} - \hat{\mathbf{y}})}{n} \quad ,$$

Where $\hat{\mathbf{y}} = \hat{\beta}^\top \mathbf{\Sigma}$.

Hence, Loss-Landscape Analysis yields three expression forms for the loss function derivative:

1. $\dfrac{\partial \mathcal{L}(\gamma)}{\partial \hat{\gamma}} = \hat{\gamma} \mathbf{y} \mathbf{Q}^{-3} \mathbf{y}^\top - \mathbf{y} \mathbf{Q}^{-2} \epsilon^\top \quad \longleftarrow$ real
2. $\left.\dfrac{\partial \mathcal{L}(\gamma)}{\partial \hat{\gamma}}\right|_{\mathbb{E}[\cdot|\epsilon]} = \hat{\gamma} \mathbf{y} \mathbf{Q}^{-3} \mathbf{y}^\top - \sigma^2 \text{tr}(\mathbf{Q}^{-2}) \quad \longleftarrow$ under expectation (the ”oracle”)
3. $\left.\dfrac{\partial \mathcal{L}(\gamma)}{\partial \hat{\gamma}}\right|_{\hat{\sigma}} = \hat{\gamma} \mathbf{y} \mathbf{Q}^{-3} \mathbf{y}^\top - \hat{\sigma}^2_{RSS} \text{tr}(\mathbf{Q}^{-2}) \quad \longleftarrow$ with estimated variance

In the following sections we are going to investigate the behavior of parameter γ found through the oracle and the expression with the estimated variance, and compare that γ with the one found through optimizing $\hat{E}_{train}$ and $\hat{E}_{test}$.

4.3. Robustness study

In this subsection we are going to conduct a series of experiments aimed at studying values of regularization parameter γ related to optimizing each of the above-derived expressions and the behavior of the direct loss computed using those values.

We begin the robustness analysis by asking a simple question: *do the derivative–based losses admit a (practical) minimum when measurement noise is present?* Concretely, we plot the derivative of the causal loss $\partial \mathcal{L} / \partial \gamma$ as a function of γ for several noise levels and examine whether we can drive it arbitrarily close to zero on a sufficiently fine logarithmic grid.

Derivative formulas. We consider two expressions (derived in the previous section) that differ only in the variance term. We write them for $\gamma > 0$ and $\mathbf{Q}(\gamma) \equiv \left(\mathbf{A}^\top \mathbf{A} + \gamma \mathbf{I}\right)$ in the following way:

$$\left.\frac{\partial \mathcal{L}(\gamma)}{\partial \gamma}\right|_{\text{real variance}} = \gamma\, \mathbf{y}^\top \mathbf{Q}(\gamma)^{-3} \mathbf{y} \; - \; \sigma^2 \; \text{tr}\big(\mathbf{Q}(\gamma)^{-2}\big). \tag{4.1}$$

$$\left.\frac{\partial \mathcal{L}(\gamma)}{\partial \gamma}\right|_{\text{estimated variance}} = \gamma\, \mathbf{y}^\top \mathbf{Q}(\gamma)^{-3} \mathbf{y} \; - \; \widehat{\sigma}^2_{RSS} \; \text{tr}\big(\mathbf{Q}(\gamma)^{-2}\big), \tag{4.2}$$

where $\widehat{\sigma}^2_{RSS}$ denotes a data-dependent variance plug-in (estimated variance).

Notice that with slight abuse of notation we put a σ subscript in 4.1 instead of $\mathbb{E}[\cdot|\epsilon]$ for the sake of simplicity. That meant to denote that the oracle derivative uses the true value of variance σ of the noise term $\varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$, The estimated variant replaces σ^2 with the residual estimate $\hat{\sigma}^2_{\text{RSS}}$.

Experimental setup. We fix $n = 200,\ p/n = 1.0,\ N/n = 0.8$ and a 50 %-dense b. For each standard deviation level $\sigma \in \{1, 2, 3, 4, 5\}$ we:

1. generate synthetic data $Y = b^\top \Sigma + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$;
2. sweep γ on logarithmic grids $10^{-5} \le \gamma \le 10^{3}$ (estimated) and $10^{-3} \le \gamma \le 10^{2}$ (oracle);
3. evaluate (4.1) and (4.2) at each γ.

Operationally, we call a loss *minimisable on the grid* if there exists a grid point γ for which the (non-negative) derivative is closest to zero, i.e. it minimizes $\left|\partial \mathcal{L}/\partial \gamma\right|$ (or equivalently attains the smallest positive value).

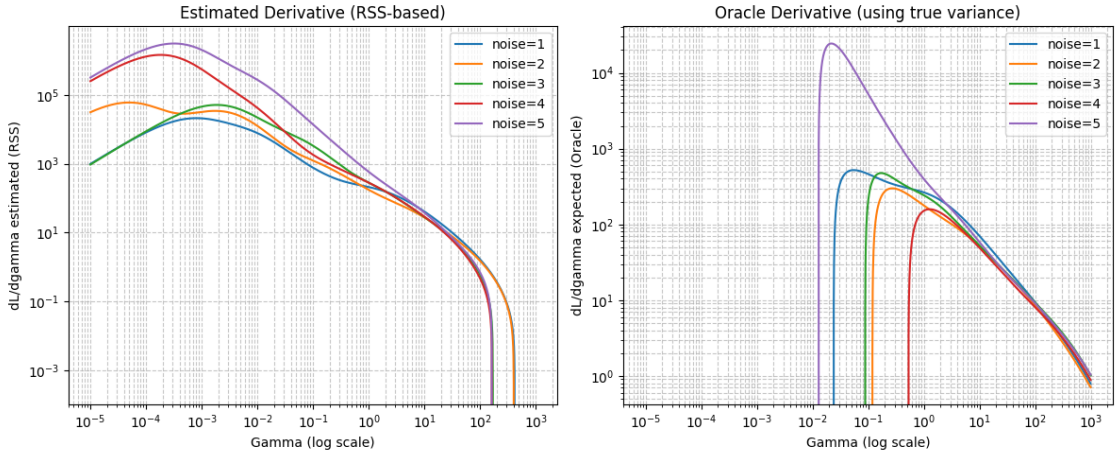


Figure 4.7: Derivative of the causal loss under five noise levels. **Left:** RSS□based estimate on $\gamma \in [10^{-5}, 10^{3}]$. **Right:** oracle derivative on $\gamma \in [10^{-3}, 10^{2}]$ (both axes log–log).

Observation. Across all tested noise levels, the curves of $\partial \mathcal{L}/\partial \gamma$ (both oracle and estimated) decrease with γ and admit a unique zero crossing within the displayed range. Consequently, with a sufficiently fine logarithmic grid, we can always find a γ whose derivative value lies arbitrarily close to zero, i.e. both losses are *minimisable* in the above grid sense. This justifies using the derivative root (or its closest grid proxy) as a robust, noise-aware choice of the regulariser, which we will compare directly against $\hat{E}_{\text{train}}$ and $\hat{E}_{\text{test}}$-based selections in the next subsection.

Fixed vs. nonfixed RSS in the derivative

We now specify two practical RSS–based plug-ins for (4.2). Let $\hat{\sigma}^2_{\mathrm{RSS}}(\cdot)$ be the residual–variance estimator as a function of the argument indicated in parentheses. Then:

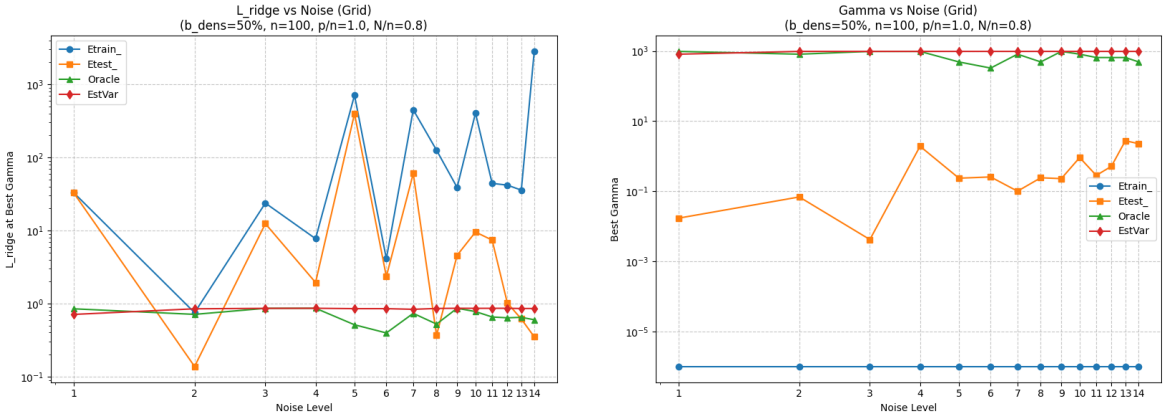
$$\textbf{fixed–RSS:}\quad \left.\frac{\partial \mathcal{L}}{\partial \gamma}\right|_{\text{RSS, fixed}} (\gamma; \gamma_{\mathrm{rss}}) = \gamma\, \mathbf{y}^\top \mathbf{Q}(\gamma)^{-3} \mathbf{y} \;-\; \hat{\sigma}^2_{\mathrm{RSS}}(\gamma_{\mathrm{rss}})\ \mathrm{tr}\big(\mathbf{Q}(\gamma)^{-2}\big), \tag{4.3a}$$

$$\textbf{nonfixed–RSS:}\quad \left.\frac{\partial \mathcal{L}}{\partial \gamma}\right|_{\text{RSS, nonfixed}} (\gamma) = \gamma\, \mathbf{y}^\top \mathbf{Q}(\gamma)^{-3} \mathbf{y} \;-\; \hat{\sigma}^2_{\mathrm{RSS}}(\gamma)\ \mathrm{tr}\big(\mathbf{Q}(\gamma)^{-2}\big). \tag{4.3b}$$

Both (4.3a)-(4.3b) are concrete instances of the estimated-variance derivative (4.2), differing only in whether the RSS variance is evaluated at a fixed reference γ_{rss} (“fixed–RSS”) or at the current γ (“nonfixed–RSS”).

where $\mathbf{Q}(\gamma) = \frac{1}{n}\mathbf{\Sigma}^\top \mathbf{\Sigma} + \gamma \mathbf{I}_n$. In the *fixed* variant, the variance is computed once at a reference value γ_{rss} (here we take $\gamma_{\mathrm{rss}} = \arg\min_\gamma \hat{E}_{\text{test}}(\gamma)$), and then held fixed while we scan γ in the derivative. In the *nonfixed* variant, the same γ used in the derivative also enters the RSS computation, effectively defining a self-consistent, γ-dependent variance plug-in.

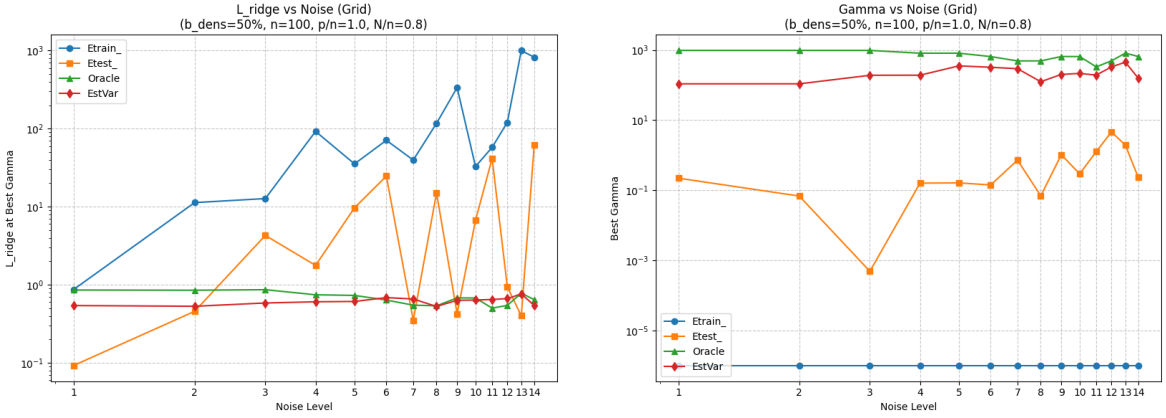
Experimental setting. We use: $n = 100$, $p/n = 1.0$, $N/n = 0.8$, a 50%–dense b, noise levels $\sigma \in \{1, 2, \ldots, 14\}$, and a logarithmic grid $\gamma \in [10^{-6}, 10^{3}]$ with step $\Delta \log_{10} \gamma = 0.01$. For robustness, all curves are *averaged over six independent runs*. At each noise level we select four candidates: $\gamma_{\hat{E}_{\text{train}}}$, $\gamma_{\hat{E}_{\text{test}}}$, γ_{oracle} (root of the oracle derivative), and γ_{RSS} (root of the corresponding RSS derivative). We then evaluate $\mathcal{L}_{\text{ridge}}$ at each selected γ.



(a) $\mathcal{L}_{\text{ridge}}$ at the selected γ vs. noise (fixed-RSS). **(b)** Selected γ vs. noise (fixed-RSS).

Figure 4.8: Fixed-RSS variant: variance $\hat{\sigma}^2_{\text{RSS}}(\gamma_{\text{rss}})$ computed at $\gamma_{\text{rss}} = \arg\min_\gamma \hat{E}_{\text{test}}(\gamma)$, then held constant while scanning γ in the derivative. Points are averages over six runs.

Fixed-RSS observations (Figure 4.8). Across noise levels, the *causal* selections ($\gamma_{\text{oracle}}, \gamma_{\text{RSS}}$) yield $\mathcal{L}_{\text{ridge}}$ that remains near $\mathcal{O}(1)$ and changes smoothly with noise, while the prediction–driven selections ($\gamma_{\hat{E}_{\text{train}}}, \gamma_{\hat{E}_{\text{test}}}$) produce larger and more volatile $\mathcal{L}_{\text{ridge}}$. The corresponding γ values show that causal tuning prefers substantially stronger regularisation.



(a) $\mathcal{L}_{\text{ridge}}$ at the selected γ vs. noise (nonfixed-RSS). **(b)** Selected γ vs. noise (nonfixed-RSS).

Figure 4.9: Nonfixed-RSS variant: variance $\hat{\sigma}^2_{\text{RSS}}(\gamma)$ recomputed at each grid point and fed back into the derivative. Points are averages over six runs.

Nonfixed-RSS observations (Figure 4.9). Allowing the variance estimate to follow the scanned γ produces a self-consistent root and, in several noise regimes, an even lower $\mathcal{L}_{\text{ridge}}$ than fixed-RSS while retaining the same qualitative behaviour (stable loss and stronger γ). Both variants confirm the central robustness claim of this section: *a causally aligned regulariser can be obtained for a wide range of noise levels, and it consistently outperforms predictive selections in direct parameter recovery.*

4.4. Convergence Study

The robustness results above showed that a causally aligned regulariser can be recovered across a wide range of noise levels. We now ask a complementary question: *how do the corresponding choices of γ and their achieved alignment error behave as the sample size n grows?* In other words, do the derivative-based prescriptions stabilise (and agree) as we move toward the high–dimensional asymptotic?

Experimental set-up. We fix the aspect ratios and ground-truth sparsity and sweep n, such that:

$$p/n = 1.0, \qquad N/n = 0.8, \qquad \text{density}(b) = 50\%, \qquad \sigma = 8.$$

As usual, for each $n \in \{100, 200, 300, 400, 500\}$ we generate synthetic data $Y = b^\top \Sigma + \varepsilon$ and compute four candidates on the same logarithmic grid $\gamma \in [10^{-6}, 10^{3}]$ with step $\Delta \log_{10} \gamma = 0.01$: (i) $\gamma_{\widehat{E}_{\text{train}}}$ minimising $\widehat{E}_{\text{train}}$, (ii) $\gamma_{\widehat{E}_{\text{test}}}$ minimising $\widehat{E}_{\text{test}}$, (iii) γ_{oracle} as the root of the oracle derivative $\partial \mathcal{L}/\partial \gamma$ in (4.1), and (iv) γ_{estVar} as the root of the empirical variance version (4.2). To smooth finite-sample fluctuations we average every curve over ten independent repeats.[2] For convergence of the *alignment* itself we report $\mathcal{L}_{\text{ridge}}(\gamma)$ evaluated at each selected γ.

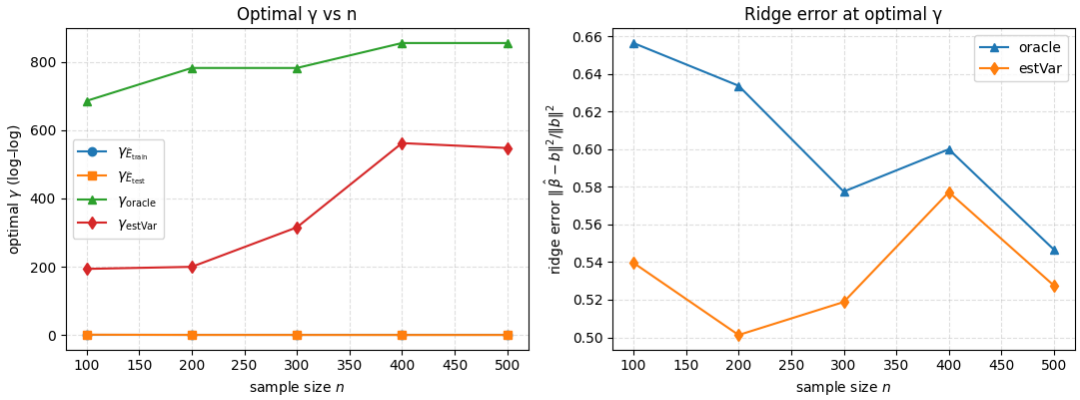


Figure 4.10: Behaviour of the regularization parameter and direct loss expression with growing sample size n (averaged over 10 runs; $p/n = 1.0$, $N/n = 0.8$, $\sigma = 8$, 50% density of b). *Left panel:* the four γ selections $\gamma_{\widehat{E}_{\text{train}}}$, $\gamma_{\widehat{E}_{\text{test}}}$, γ_{oracle}, and γ_{estVar}. *Right panel:* direct alignment loss $\mathcal{L}_{\text{ridge}}$ at γ_{oracle} and γ_{estVar}.

Findings

- **Derivative-based gap shrinks with n.** As n increases from 100 to 500, the two causal selections move closer: γ_{oracle} remains in the high hundreds (about 7×10^2-9×10^2), while γ_{estVar} rises from roughly 2×10^2 toward 5×10^2-6×10^2, narrowing the gap. In contrast, the predictive optima $\gamma_{\widehat{E}_{\text{train}}}$ and $\gamma_{\widehat{E}_{\text{test}}}$ stay near zero (orders of magnitude smaller). *Reading the log axis:* the tick labels show the *actual* γ values (spacing is logarithmic only). Thus a point around "600" means $\gamma \approx 6 \times 10^2$; "850" means $\gamma \approx 8.5 \times 10^2$ (not 10^{600}, etc.).
- **Alignment gap shrinks with n.** The right panel of Figure 4.10 shows $\mathcal{L}_{\text{ridge}}(\gamma_{\text{oracle}})$ and $\mathcal{L}_{\text{ridge}}(\gamma_{\text{estVar}})$ both decreasing as n grows, and their difference narrows–clear finite-sample convergence behaviour.
- **Empirical edge persists.** Across the inspected n, the estVar choice attains $\mathcal{L}_{\text{ridge}}$ that is consistently comparable to, and often lower than, the oracle – echoing the advantage observed in the robustness study.

[2] Runs were executed in batches on the cluster, which is why the displayed grid of n stops at 500.

Takeaway (revised). Due to runtime limits on the TU Delft cloud cluster and averaging over repeated runs, we report $n \in \{100, 200, 300, 400, 500\}$. This range is too short to *prove* asymptotic convergence, but Figure 4.10 already exhibits stabilisation of the derivative-based γ in the 10^2-10^3 decade and a steady reduction of the alignment error with n. We fixed the noise level to $\sigma = 8$ (no special theoretical status); this choice was guided empirically by earlier experiments showing that for $\sigma \gtrsim 6$-7, the estVar rule tracks or slightly improves upon the oracle. Within these constraints, the evidence supports the practical conclusion: the data-driven derivative selection γ_{estVar} is stable in n, operates on the same scale as the oracle, and delivers improving $\mathcal{L}_{\text{ridge}}$ as sample size grows, while predictive selections remain far smaller and misaligned with the causal objective.

4.5. Real-Data Validation: Fashion−MNIST

We validate the proposed estimator on real images using a two-class subset of Fashion-MNIST (labels 1 vs. 2) under the same ReLU random-features ridge classification pipeline and preprocessing used throughout the paper, therefore, mimicking the author's experiment. The study proceeds in two steps. First, we assess *predictive robustness* in a head-to-head comparison with the Coulliet E_{test} calibration: over 50 Monte–Carlo runs (new random feature matrices W each run), we train both methods on identical train/test splits and aggregate accuracies and confusion matrices to verify that our approach does not underperform Coulliet's method on average. Second, holding the baseline split fixed, we conduct a controlled *perturbation analysis*: we introduce localized edits to the *training* inputs (single pixel and square patch brightness shifts of varying sizes), refit both models on the perturbed data using their respective γ selection rules, and compare the resulting coefficient vectors β against their baseline counterparts learned on the original data. This design lets us quantify not only changes in test performance but also how each method's inferred importance pattern *moves under perturbations* (stability vs. reactivity and concentration), providing empirical evidence about the interpretability and robustness of the learned representations.

4.5.1. Robustness analysis - fashion MNIST

We begin by conducting a robustness study to verify that our proposed method performs at least as well as the Coulliet's E_{test} calibration in terms of predictive accuracy. Specifically, we run 50 Monte-Carlo trials on the two-class Fashion MNIST split under identical preprocessing and ReLU random-features ridge settings, aggregate accuracies and confusion matrices, and compare the two methods head-to-head.

Setting and data. We evaluate the proposed estimator on a two-class subset of Fashion MNIST (labels 1 vs. 2), cast as a binary classification problem with labels $y \in \{-1, +1\}$. Images are 28×28; we vectorize each image to $x \in \mathbb{R}^p$ with $p = 784$. We adopt the same random-features ridge pipeline used throughout the paper, with a ReLU nonlinearity:

$$\sigma(t) = \max\{t, 0\} \quad ,$$

and the corresponding ReLU (first-order arc-cosine) kernel $K(\cdot, \cdot)$ used to build the deterministic-equivalent criterion.

Data Preprocessing. Before training, the Fashion-MNIST data are transformed to match the theoretical assumptions of the random-features ridge model and ensure consistent global scaling across Monte-Carlo runs. The procedure proceeds as follows.

In terms of class selection and vectorization, we retain only two classes (labels 1 and 2), corresponding to a binary classification task with $y \in \{-1, +1\}$. Each 28×28 grayscale image is vectorized into $x \in \mathbb{R}^p, \quad p = 784$. Let $X = [x_1, \ldots, x_M] \in \mathbb{R}^{p \times M}$ denote the full dataset of all selected samples.

Then, we perform global normalization. All pixel intensities are first scaled to the unit interval by

dividing by the global maximum

$$X \leftarrow \frac{X}{\max(X)}.$$

This ensures all entries of X lie in $[0, 1]$.

We then perform a global centering and rescaling step so that the average squared norm of the centered vectors equals p. Let

$$\mu_g = \frac{1}{M}\sum_{i=1}^{M} x_i, \qquad X_c = X - \mu_g \mathbf{1}_M^\top,$$

and define the global scaling factor

$$s_g = \frac{\sqrt{p}}{\sqrt{\frac{1}{M}\sum_{i=1}^{M} \|x_i - \mu_g\|_2^2}}.$$

The globally standardized data are then

$$X_{\text{std}} = s_g X_c.$$

After global standardization, we perform a second normalization across the pooled subset of the two chosen classes to ensure comparability of within-class magnitudes. Let

$$X^{(1)}, X^{(2)} \in \mathbb{R}^{p \times M_j}$$

denote the subsets for classes 1 and 2, respectively. We form their pooled matrix

$$X_{\text{pool}} = [\, X^{(1)} \; X^{(2)} \,] \quad ,$$

and compute

$$\mu_p = \frac{1}{M_1 + M_2}\sum_i x_i^{(\text{pool})} \quad , \qquad s_p = \frac{\sqrt{p}}{\sqrt{\frac{1}{M_1+M_2}\sum_i \|x_i^{(\text{pool})} - \mu_p\|_2^2}} \quad .$$

Each class subset is then re-centered and rescaled via

$$X^{(j)} \leftarrow s_p\,(X^{(j)} - \mu_p) \quad , \qquad j \in \{1, 2\} \quad .$$

Next step is the train–test partitioning. With a prescribed aspect ratio $c_1 = p/n$ and training fraction ρ_{tr}, the number of training and test samples per class are

$$n = \left\lfloor \frac{p}{c_1} \right\rfloor, \qquad n_{\text{te}} = \left\lfloor n\,\frac{1-\rho_{\text{tr}}}{\rho_{\text{tr}}} \right\rfloor.$$

For each class j, a random permutation π_j (with fixed seed) selects disjoint subsets for training and test:

$$X_{\text{tr}}^{(j)} = X^{(j)}_{[:,\,\pi_j(1:n_j)]}, \qquad X_{\text{te}}^{(j)} = X^{(j)}_{[:,\,\pi_j(n_j+1:n_j+n_{\text{te},j})]} \quad .$$

The final training and test matrices are concatenations

$$X_{\text{tr}} = [\, X_{\text{tr}}^{(1)} \; X_{\text{tr}}^{(2)} \,], \qquad X_{\text{te}} = [\, X_{\text{te}}^{(1)} \; X_{\text{te}}^{(2)} \,] \quad ,$$

with corresponding binary label vectors

$$y_{\text{tr}} = \begin{bmatrix} -\mathbf{1}_{n_1} \\ +\mathbf{1}_{n_2} \end{bmatrix}, \qquad y_{\text{te}} = \begin{bmatrix} -\mathbf{1}_{n_{\text{te},1}} \\ +\mathbf{1}_{n_{\text{te},2}} \end{bmatrix}.$$

In summary, this preprocessing pipeline enforces deterministic centering and scaling of all images, guarantees $\mathbb{E}\|x_i\|_2^2 \approx p$, and produces balanced train–test splits under fixed seeds. The resulting $(X_{\text{tr}}, X_{\text{te}}, y_{\text{tr}}, y_{\text{te}})$ pairs form the standardized inputs to the ReLU random-features ridge classifier used throughout the validation experiments.

Model and hyperparameters. Similarly to the Coulliet's experiment We consider a random-features ridge classifier with hidden width (inner neuron layer) $N = \lfloor c_2 n \rceil$ and i.i.d. Gaussian features $W \in \mathbb{R}^{N\times p}$, $W_{ij} \sim \mathcal{N}(0,1)$. Given inputs $X \in \mathbb{R}^{p\times n}$ and labels $y \in \{-1,+1\}^n$, let

$$\Sigma \;=\; \sigma(WX)$$

with $\sigma(t) = \max\{t, 0\}$ (ReLU). For any ridge parameter $\gamma > 0$, the coefficient vector is obtained *in closed form* from the standard ridge expression (see (3.1)):

$$\widehat{\beta}(\gamma) \quad \text{as in} \quad (3.1).$$

Prediction uses the score $f(x) = \Sigma(x)^\top \widehat{\beta}(\gamma)$ and the decision rule $\operatorname{sign}(f(x))$. The *choice* of γ is determined by the calibration rules described next; $\widehat{\beta}$ is then formed by plugging that γ into (3.1).

Coulliet calibration and our direct calibration. We compare two one–parameter selection rules for γ evaluated on the same logarithmic grid $\gamma \in \{10^t : t \in [-6, 4],\ \Delta t = 0.005\}$:

- **Coulliet ($\overline{E}_{\text{test}}$) calibration.** We select

  $$\gamma_{\mathrm{C}} \;=\; \arg\min_{\gamma}\ \overline{E}_{\text{test}}(\gamma),$$

  where $\overline{E}_{\text{test}}(\gamma)$ is the deterministic–equivalent test proxy from the theory (cf. Eq. (3.12), which depends on the fixed–point $\delta(\gamma)$).

- **Direct calibration (nonfixed RSS; ours).** We select

  $$\gamma_{\mathrm{D}} \;=\; \arg\min_{\gamma}\ \big|\partial_\gamma \mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma)\big|,$$

  where $\mathcal{L}^{(\mathrm{RSS})}_{\mathrm{direct}}(\gamma)$ is the nonfixed RSS–based direct loss from our loss–landscape analysis (Eq. (4.3b)). In other words, we pick the grid point whose derivative magnitude is minimal.

In both cases, once γ is chosen (either γ_{C} or γ_{D}), we compute $\widehat{\beta}$ via the closed–form ridge expression (3.1) and classify by $\operatorname{sign}(f)$.

Monte–Carlo protocol. We run 50 Monte–Carlo trials to assess predictive robustness. The training/test splits $(X_{\text{train}}, X_{\text{test}})$ and labels are held fixed across trials. The Coulliet selection γ_{C} is computed once from $(X_{\text{train}}, X_{\text{test}})$ via $\overline{E}_{\text{test}}(\gamma)$ and is common to all trials. For each trial $r = 1, \ldots, 50$, we draw an independent $W^{(r)}$, recompute the direct selection $\gamma_{\mathrm{D}}^{(r)}$ on the grid (the direct criterion depends on W), form

$$\widehat{\beta}_{\mathrm{C}} \;=\; \widehat{\beta}(\gamma_{\mathrm{C}}), \qquad \widehat{\beta}_{\mathrm{D}}^{(r)} \;=\; \widehat{\beta}(\gamma_{\mathrm{D}}^{(r)}),$$

and evaluate train/test predictions by thresholding at zero. We record per–run accuracies and confusion vectors $(\mathrm{TP}, \mathrm{TN}, \mathrm{FP}, \mathrm{FN})$, and we report run–wise average accuracies together with confusion totals summed over the 50 runs (normalized by 50 when per–run rates are required).

Concrete hyperparameters (this experiment).

$$\text{activation} = \mathrm{ReLU}, \quad (L_-, L_+) = (1, 2), \quad c_1 = \frac{p}{n} = 1.0, \quad c_2 = \frac{N}{n} = 1.0, \quad \text{train fraction } \tau = 0.8,$$

$$\gamma \text{ grid: } \ \gamma \in \{10^t :\ t \in [-6, 4],\ \Delta t = 0.005\}.$$

Here c_1 sets $n \approx p$ to retain high-dimensional setting, and c_2 fixes $N \approx n$. The test proxy $\overline{E}_{\text{test}}(\gamma)$ uses the ReLU (first–order arc–cosine) kernel as in the theory section, through Eq. (3.12); our direct rule uses Eq. (4.3b). In all cases, the final coefficients are obtained from the ridge closed form (3.1).

Outputs. For transparency and downstream analysis, we store (i) the per–run accuracy arrays for both methods on train and test, (ii) the summed confusion counts across runs, and (iii) the per–run direct selections $\{\gamma_{\mathrm{D}}^{(r)}\}_{r=1}^{50}$ together with the common Coulliet γ_{C}, enabling paired head–to–head statistical comparisons and dispersion summaries across the 50 trials.

Predictive results on Fashion–MNIST. Under the ReLU random–features ridge model with $p = n = N = 784$ and 50 Monte–Carlo trials, both calibration rules achieve near–perfect prediction. The Coulliet selection (via $\overline{E}_{\text{test}}$) yields $\gamma_{\text{C}} = 70.79$, whereas our direct rule (nonfixed RSS derivative) concentrates at a larger scale (median $\gamma_{\text{D}} \approx 295.12$). Averaged across runs, test accuracy is 0.9956 for the direct rule versus 0.9930 for Coulliet; train accuracy is 0.9943 (direct) versus 1.0000 (Coulliet). The test–set confusion totals (summed over runs) are reported below.

	Coulliet (γ_{C})		**Direct (γ_{D})**	
	Positive	Negative	Positive	Negative
True	4833	4898	4861	4896
False	2	67	4	39
Accuracy	0.9930		0.9956	

Table 4.1: Test confusion matrices (totals over 50 Monte–Carlo trials) for the two calibration methods. Entries correspond to counts of true and false predictions across all runs.

Discussion of findings. (i) Both methods are highly accurate; the direct calibration attains a small but consistent improvement on test ($+0.27$ percentage points). (ii) The gain is driven primarily by *fewer false negatives* (FN: $67 \to 39$), with a negligible increase in false positives (FP: $2 \to 4$). This translates into higher recall for the positive class under our rule. (iii) The larger γ selected by the direct method implies stronger regularization; correspondingly, its train accuracy is slightly below perfect (0.9943 vs. 1.0000 for Coulliet), yet generalizes marginally better on test-consistent with reduced overfitting. (iv) Overall, the direct rule matches or improves predictive performance while selecting a regularization scale aligned with the loss–landscape analysis, reinforcing its practical viability alongside Coulliet's $\overline{E}_{\text{test}}$ calibration.

4.5.2. Perturbation analysis: Fashion--MNIST

Having established that our direct-loss calibration for the regularization parameter γ attains accuracy comparable to Coulliet's method, we now examine the behaviour and structure of the learned coefficient vector β. We probe the *stability vs. reactivity* of the two calibration rules under localized input edits. Starting from a fixed train/test split on the Fashion–MNIST 1 vs. 2 task and a fixed random-features matrix W, we apply controlled brightness shifts to selected pixels (singletons and square patches) *in the training inputs only*, refit each method with its own γ-selection rule, and compare the learned coefficient vectors to their baseline counterparts. This design isolates how each rule's inferred importance pattern moves under small, spatially localized changes. The test data remain unchanged throughout the analysis, ensuring that any observed differences stem solely from modifications in the training set. This allows us to attribute changes in the learned coefficients to the learning dynamics of each calibration rule, rather than to shifts in the evaluation distribution.

Baseline. With activation $\sigma(t) = \max\{t, 0\}$ (ReLU), aspect ratios $c_1 = p/n = 1.0$ and $c_2 = N/n = 1.0$ ($p = n = N = 784$), and train fraction $\tau = 0.8$, we construct

$$\Sigma = \sigma(W X_{\text{tr}}), \qquad \Sigma(x) = \sigma(Wx),$$

just like before, and then select γ by two one–dimensional rules evaluated on the common logarithmic grid

$$\gamma \in \left\{10^t : \; t \in [-6,\, 3.5],\; \Delta t = 0.005\right\}.$$

Coulliet's calibration picks

$$\gamma_{\text{C}} = \arg\min_{\gamma} \; \overline{E}_{\text{test}}(\gamma) \quad \text{(deterministic equivalent; cf. Eq. (3.12))},$$

while our *direct* rule picks

$$\gamma_{\text{D}} = \arg\min_{\gamma} \left|\partial_\gamma \mathcal{L}^{(\text{RSS})}_{\text{direct}}(\gamma)\right| \quad \text{(nonfixed RSS; cf. Eq. (4.3b))}.$$

Given either γ, the coefficients are formed by the closed–form ridge expression (Eq. (3.1)), yielding baseline vectors

$$\beta_{\mathrm{C}}^{\mathrm{base}} = \widehat{\beta}(\gamma_{\mathrm{C}}), \qquad \beta_{\mathrm{D}}^{\mathrm{base}} = \widehat{\beta}(\gamma_{\mathrm{D}}).$$

Perturbations (training-only, localized). Let $X_{\mathrm{tr}} \in \mathbb{R}^{p\times n}$ and $X_{\mathrm{te}} \in \mathbb{R}^{p\times n_{\mathrm{te}}}$ denote the standardized training and test matrices, respectively, constructed as described in Section 4.5.1. Recall that in our controlled high-dimensional setting, we fix $p = n = 784$ and $n_{\mathrm{te}} \approx 196$, corresponding to 784 synthetic training and 196 test images (each of size 28×28) drawn from the two selected Fashion-MNIST classes. For a chosen subset of feature indices $S \subset \{1,\dots,p\}$, a perturbation magnitude $\Delta > 0$, and a sign $s \in \{-1,+1\}$, we define a *training-only* perturbation

$$X'_{\mathrm{tr}} \;=\; X_{\mathrm{tr}} + s\,\Delta\,\mathbf{1}_S\,\mathbf{1}_n^\top, \qquad X'_{\mathrm{te}} = X_{\mathrm{te}},$$

where $\mathbf{1}_S \in \mathbb{R}^p$ is an indicator vector equal to 1 on S and 0 elsewhere. The test matrix X_{te} remains unaltered, ensuring that performance differences can be attributed solely to modifications of the training set and not to shifts in the evaluation distribution.

We take $\Delta = 0.5$ in the standardized feature space, which corresponds to a moderate brightness adjustment in the original image domain. The perturbation therefore increases or decreases the brightness of selected pixels across all training images (one entire pixel column of X_{tr}) while leaving the test images intact. The following localized perturbation scenarios are considered:

- **Single pixel:** $|S| = 1$ (six random pixel locations; random sign).
- 5×5 **patch:** $|S| = 25$ (six random top-left locations; random sign).
- 10×10 **patch:** $|S| = 100$ (six random locations; random sign).
- 20×20 **patch:** $|S| = 400$ (six random locations; random sign; only if the 28×28 image size permits).

Each perturbation thus modifies the brightness of either a single pixel or a contiguous block of pixels across all $n = 784$ training images, allowing us to evaluate how small, spatially localized changes in the training data affect the learned coefficients.

For each perturbed training matrix X'_{tr} we recompute the two γ's on the *same* grid:

$$\gamma_{\mathrm{C}}^{(\mathrm{scen})} = \arg\min_{\gamma}\ \overline{E}_{\mathrm{test}}^{(\mathrm{scen})}(\gamma), \qquad \gamma_{\mathrm{D}}^{(\mathrm{scen})} = \arg\min_{\gamma}\ \big|\partial_\gamma \mathcal{L}_{\mathrm{direct}}^{(\mathrm{RSS})}(\gamma; X'_{\mathrm{tr}})\big|,$$

where $\overline{E}_{\mathrm{test}}^{(\mathrm{scen})}$ uses kernels built from $(X'_{\mathrm{tr}}, X_{\mathrm{te}})$ and the direct derivative is evaluated with the same fixed W but the edited X'_{tr}. We then form

$$\beta_{\mathrm{C}}^{(\mathrm{scen})} = \widehat{\beta}\Big(\gamma_{\mathrm{C}}^{(\mathrm{scen})}\Big), \qquad \beta_{\mathrm{D}}^{(\mathrm{scen})} = \widehat{\beta}\Big(\gamma_{\mathrm{D}}^{(\mathrm{scen})}\Big),$$

and evaluate train/test predictions on $(X'_{\mathrm{tr}}, X_{\mathrm{te}})$ by thresholding at zero.

For each scenario we store $(\gamma_{\mathrm{C}}^{(\mathrm{scen})}, \gamma_{\mathrm{D}}^{(\mathrm{scen})})$, the perturbed coefficients $(\beta_{\mathrm{C}}^{(\mathrm{scen})}, \beta_{\mathrm{D}}^{(\mathrm{scen})})$, and the corresponding train/test metrics. In the analysis, we will compare each perturbed vector to its baseline counterpart to quantify how concentrated or diffuse the response is under localized edits, and whether the direct rule's stronger regularization yields more stable importance patterns than the Coulliet selection.

Coefficient sensitivity under localized training edits. For each perturbation scenario s, we compare the perturbed coefficients $\beta^{(s)}$ to the baseline coefficients β^{base} via the relative squared ℓ_2–deviation

$$\Delta_{\ell_2}(s) \;=\; 100 \times \frac{\big\|\beta^{(s)} - \beta^{\mathrm{base}}\big\|_2^2}{\big\|\beta^{\mathrm{base}}\big\|_2^2}\ \ [\%].$$

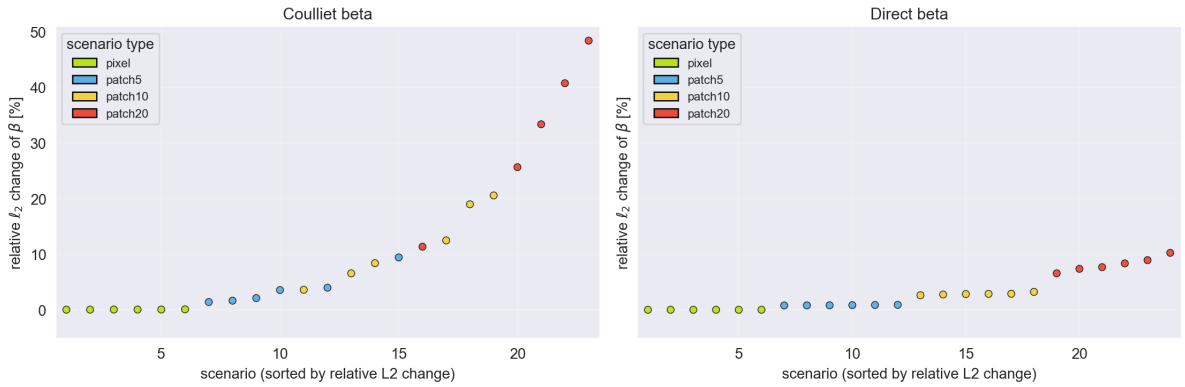


Figure 4.11: Relative ℓ_2 change of the learned coefficients under training-only, localized brightness edits. Left: Coulliet beta. Right: Direct beta. Scenarios are sorted by Δ_{ℓ_2}; colors indicate perturbation type.

We report $\Delta_{\ell_2}(s)$ for all scenarios and for both calibration rules (Coulliet and Direct), sorting scenarios along the horizontal axis by increasing Δ_{ℓ_2}. Markers are color–coded by perturbation type (single pixel, 5×5, 10×10, 20×20). For readability, extreme outliers (top 1% of Δ_{ℓ_2} across all panels) are omitted from the plot; these correspond to boundary selections of γ and are discussed in the text.

Main takeaway Figure 4.11 summarizes the relative change of the learned coefficients under localized perturbations of the training data. Each marker corresponds to a single perturbation scenario, with the vertical axis showing the relative ℓ_2–distance between the perturbed and baseline coefficients. Overall, Coulliet's calibration exhibits substantially larger coefficient shifts than the direct-loss rule, particularly for larger (10×10 and 20×20) patch edits. Quantitatively, the top Coulliet scenarios reach relative changes between 25% and 50%, with one extreme case exceeding $2.9 \times 10^6\%$ due to an instability in γ-selection near the boundary of the grid. In contrast, the corresponding top-five direct-loss scenarios remain in the range of roughly 7-10%. These results confirm that the direct-loss method produces significantly more stable coefficient vectors under identical data perturbations, suggesting that its calibration rule imposes a smoother and more robust regularization response. The extreme outlier in the Coulliet set (the 20×20 patch with a negative brightness shift) will be examined separately below to illustrate the mechanism behind such a large deviation.

Overlay of baseline and perturbed coefficients by perturbation type. To better understand how each calibration rule reacts to localized changes in the training data, we visualize the learned coefficient vectors β before and after perturbation. For each perturbation type (single pixel, 5×5, 10×10, and 20×20 patch), Figure 4.12 compares the baseline coefficients $\beta^{(\text{base})}$ with their perturbed counterparts $\beta^{(s)}$. Each subplot shows the coefficients sorted by their baseline order (ascending $\beta^{(\text{base})}$ values), allowing us to observe how localized brightness edits in the training set shift the learned importance pattern across the feature dimension. The left column corresponds to Coulliet's γ-selection rule, and the right column to our direct-loss calibration.

The blue line represents the baseline $\beta^{(\text{base})}$, while the orange line shows the coefficients obtained after the perturbation. Higher transparency of the orange curve allows the blue baseline to remain visible beneath it, highlighting where the perturbed β departs most from its reference. Together, these overlays provide an intuitive measure of model *stability* under input perturbations.

While Figure 4.12 allows direct inspection of how the learned coefficients shift along the feature index, the magnitude of those changes can sometimes be obscured when small and large coefficients coexist on the same scale. To make these deviations more apparent, we sort the coefficients by their absolute baseline magnitude in Figure 4.13, which emphasizes where the most influential features in the baseline model experience the strongest or weakest reweighting after perturbation.

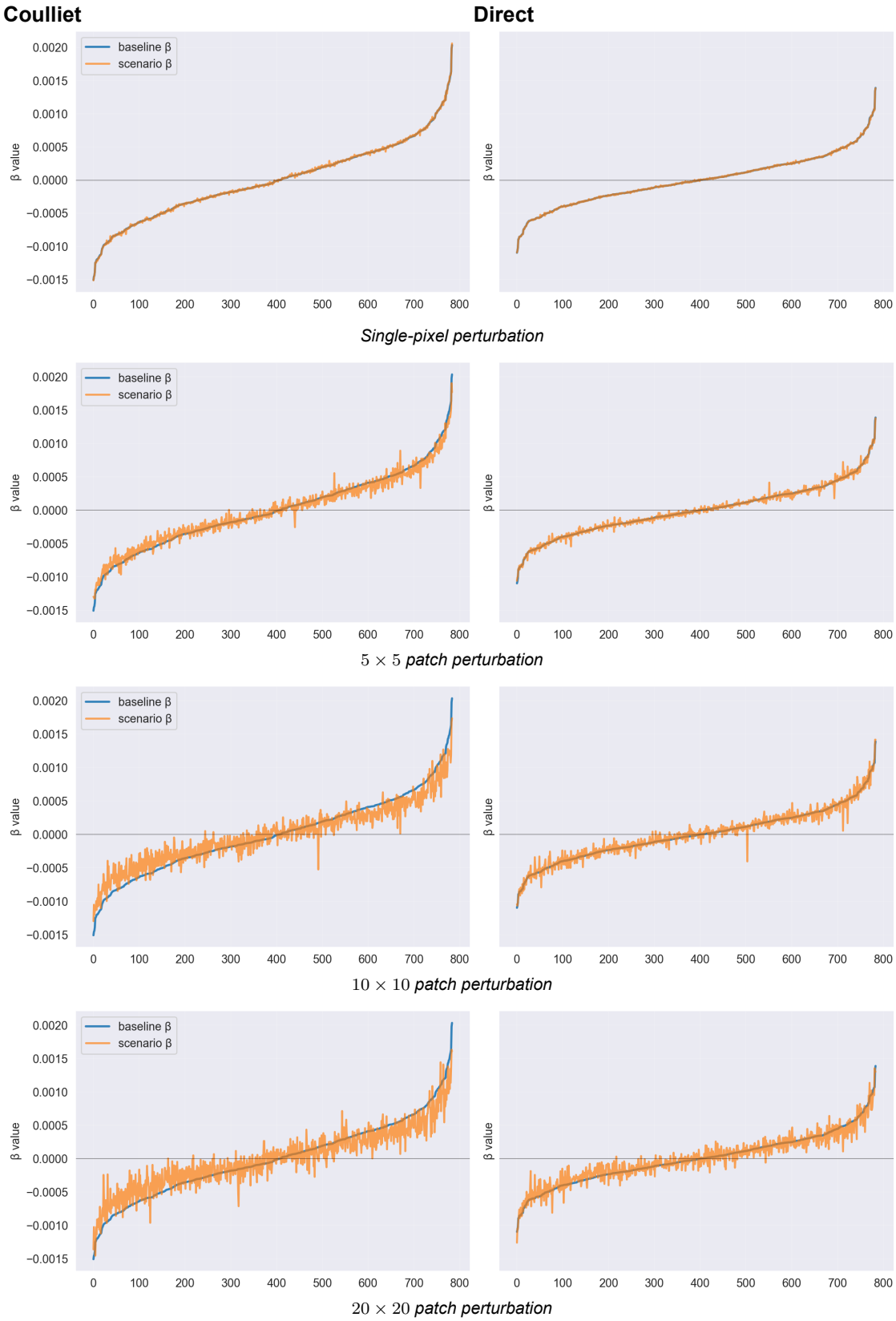


Figure 4.12: Overlay of baseline and perturbed coefficient vectors β for four representative perturbation types. Each row shows a two-panel image with Coulliet (left) and Direct (right). Blue: baseline coefficients $\beta^{(\mathrm{base})}$; orange: perturbed coefficients $\beta^{(s)}$.

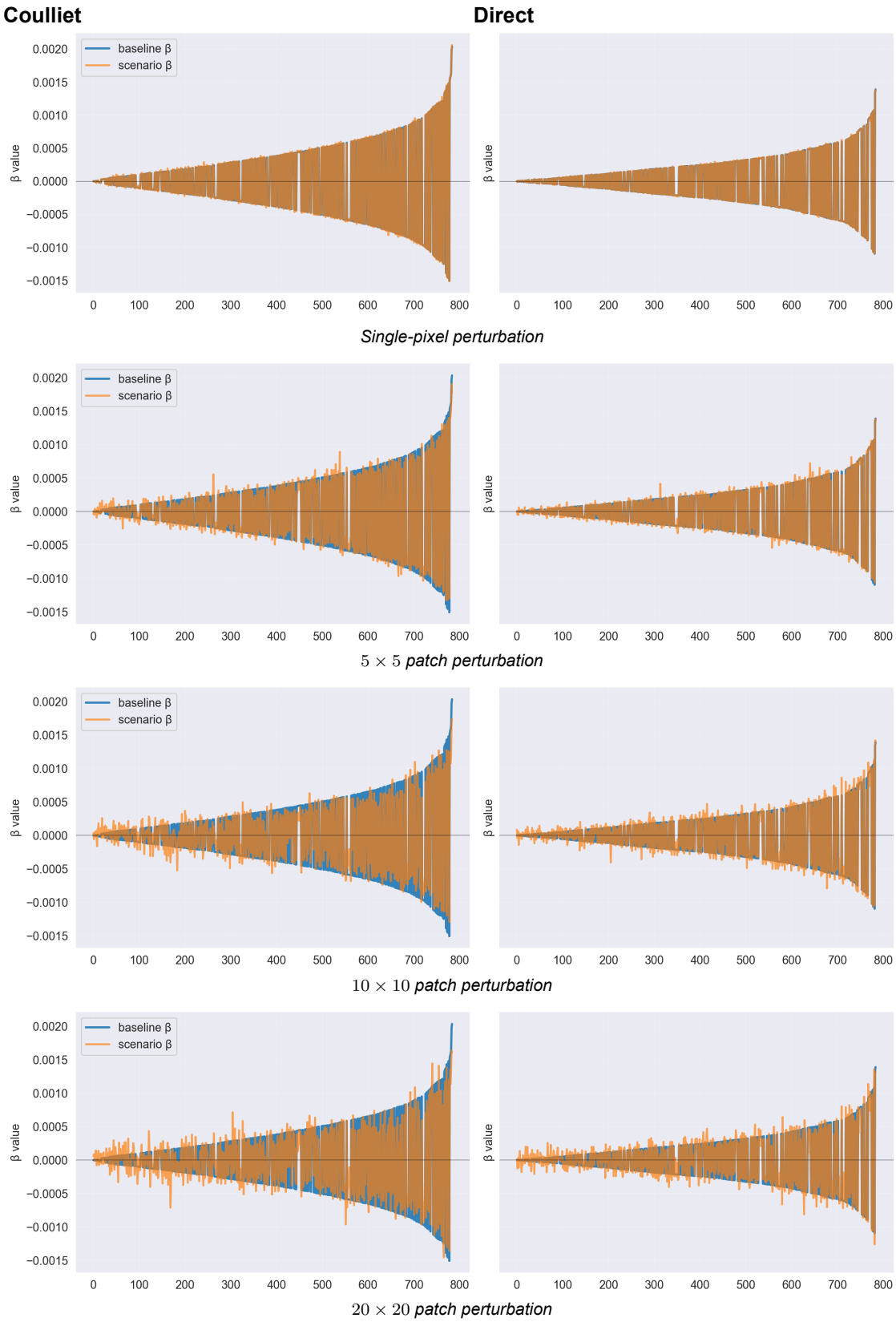
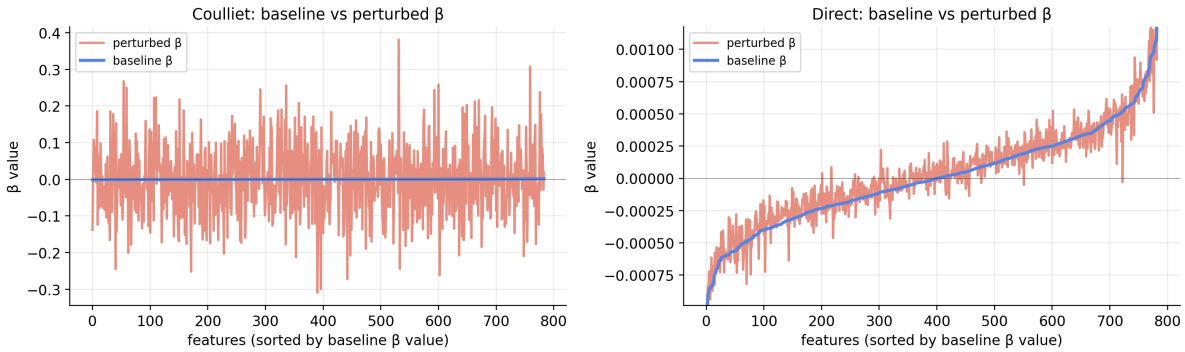


Figure 4.13: Same as Figure 4.12, but with coefficients sorted by their absolute baseline magnitude $|\beta^{(\text{base})}|$. Sorting by absolute value highlights how perturbations affect the largest-magnitude coefficients most strongly.



Extreme 20×20 *patch: baseline vs. perturbed* β *(features sorted by baseline order).*

Figure 4.14: Stress case where Coulliet's calibration collapses to the grid floor ($\gamma_{\mathrm{C}}^{(\mathrm{scen})} \approx 10^{-6}$), while the direct-loss rule stays in a moderate range. *Note:* the two panels use different vertical scales (we zoom the Direct panel to reveal its small variation). The blue baseline curves are essentially identical across panels; the visual discrepancy is solely due to the differing y-axis ranges.

Focused overlays on the most influential coefficients. Complementing the full–vector overlays, we zoom in on the extreme tails of the baseline coefficient distribution. For each perturbation type (single pixel, 5×5, 10×10, 20×20), we select the 100 coefficients from a chosen tail of the *baseline* vector and overlay the corresponding segments of $\beta^{(\mathrm{base})}$ and $\beta^{(s)}$:

$$\mathcal{I}_{\mathrm{tail}} = \begin{cases} \text{indices of the } 100 \text{ smallest entries of } \beta^{(\mathrm{base})}, & \text{if tail = bottom,} \\ \text{indices of the } 100 \text{ largest entries of } \beta^{(\mathrm{base})} \text{ (ordered by } |\beta^{(\mathrm{base})}|), & \text{if tail = top.} \end{cases}$$

Plotting $\{\beta_i^{(\mathrm{base})}, \beta_i^{(s)}\}_{i \in \mathcal{I}_{\mathrm{tail}}}$ against the rank within the selected tail emphasizes how the most negative/most positive baseline coefficients are reweighted by training perturbations. As before, the left column corresponds to Coulliet's calibration and the right to our direct-loss rule.

You can see the tail baseline-vs-scenario plots in figures 4.15 and 4.16 for top and bottom tail respectively.

Interpretation of tail-slice overlays. Examining the coefficient tails in Figures 4.15 and 4.16 reveals a clear structural difference between the two calibration rules. For Coulliet's method, the perturbations tend to *amplify* the absolute magnitude of the coefficients: in the bottom tail, the perturbed β's are on average less negative (shifted upward), while in the top tail they are systematically lower than the baseline curve. This pattern indicates that Coulliet's calibration adjusts γ in a way that consistently rebalances the relative strength of extreme coefficients under localized training edits.

By contrast, the proposed *direct-loss* rule produces markedly more stable coefficient profiles. Across all perturbation types and both tails, the perturbed $\beta^{(s)}$ curves remain close to their baseline $\beta^{(\mathrm{base})}$ counterparts, with substantially reduced variance and no systematic bias in either direction. This suggests that the direct-loss calibration maintains a more consistent scaling of the learned weights and exhibits stronger robustness to localized input modifications.

A stress case where Coulliet's calibration collapses to vanishing regularization. To illustrate the behaviour behind the largest relative change omitted in the scatter 4.11 (for plotting purposes), we examine a 20×20 patch scenario that produces an extreme deviation for Coulliet's rule (top entry in the "Top-5 Coulliet changes"). In this instance the Coulliet grid search selects a *near-zero* penalty, $\gamma_{\mathrm{C}}^{(\mathrm{scen})} \approx 10^{-6}$ (i.e., $\Delta \log_{10} \gamma \ll 0$ relative to the baseline), effectively turning off regularization. The direct-loss rule, by contrast, remains in a moderate range. We display two overlays of the coefficient vectors-unsorted (by the baseline order) and sorted by $|\beta^{(\mathrm{base})}|$ to make the pattern visible under both views.

What happens and why? Two effects jointly explain the blow-up in the relative-change metric for Coulliet. (i) *Tiny denominator:* in this scenario the baseline Coulliet vector has a very small norm,

$$100 \cdot \frac{\|\beta^{(\mathrm{scen})} - \beta^{(\mathrm{base})}\|_2^2}{\|\beta^{(\mathrm{base})}\|_2^2}$$

is therefore magnified even for moderate absolute differences. (ii) *Near-zero regularization with rank deficiency:* when $\gamma \to 0$, the ridge solution becomes highly sensitive to small singular directions of $A = \Sigma/\sqrt{n}$; with $\mathrm{rank}(A) = r < n$, the closed form adds a large null-space component (scaling like $1/\gamma$). Although this component is annihilated by Σ (so many coordinates stay near zero in the overlay), the span component can inflate along ill-conditioned directions, yielding a large *relative* change. In contrast, the direct-loss rule keeps γ away from the boundary, producing a smoother, better-conditioned update with much smaller variance and no catastrophic swing.

For reference, below are the top 5 scenario feature vector changes for both Coulliet and Direct method:

Largest relative changes.

Coulliet: top-5 $100 \cdot \|\beta^{(s)} - \beta^{\mathsf{base}}\|_2^2 / \|\beta^{\mathsf{base}}\|_2^2$			**Direct: top-5** $100 \cdot \|\beta^{(s)} - \beta^{\mathsf{base}}\|_2^2 / \|\beta^{\mathsf{base}}\|_2^2$		
tag	type	change [%]	tag	type	change [%]
patch20_04_r7c3_d+0.50sgn-1	patch20	2,977,344.967	patch20_03_r0c0_d+0.50sgn1	patch20	10.275
patch20_05_r0c5_d+0.50sgn1	patch20	48.425	patch20_06_r0c8_d+0.50sgn1	patch20	8.966
patch20_06_r0c8_d+0.50sgn1	patch20	40.770	patch20_02_r1c2_d+0.50sgn1	patch20	8.384
patch20_02_r1c2_d+0.50sgn1	patch20	33.395	patch20_04_r7c3_d+0.50sgn-1	patch20	7.703
patch20_03_r0c0_d+0.50sgn1	patch20	25.682	patch20_05_r0c5_d+0.50sgn1	patch20	7.404

The Coulliet side shows markedly larger relative deviations; the first row is the outlier we analyze in the section above.

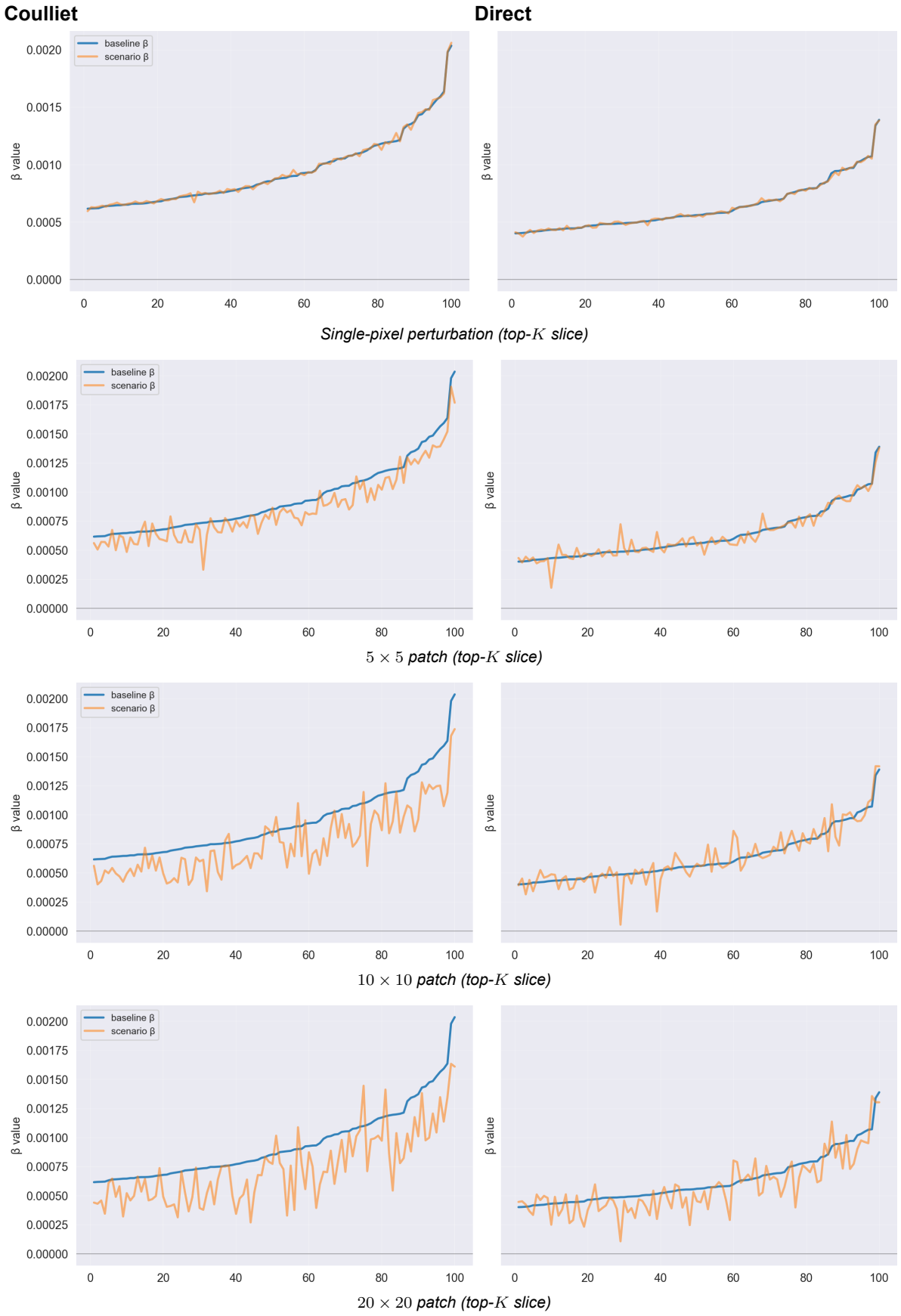


Figure 4.15: Top-tail-slice overlays of baseline vs. perturbed coefficients (top 100 beta values). Each row is a two-panel image with Coulliet (left) and Direct (right); the horizontal axis is the rank within the selected tail.

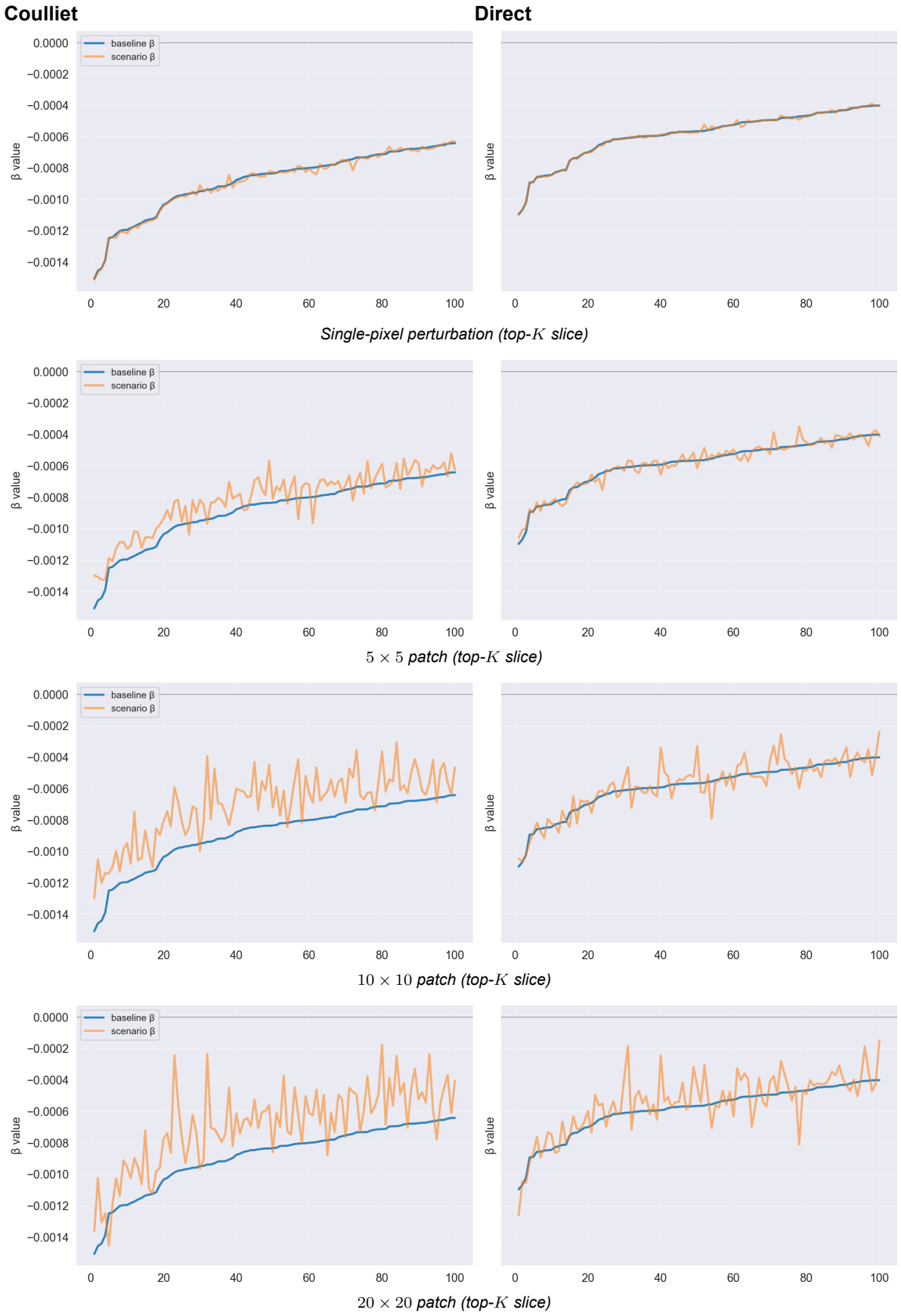


Figure 4.16: Bottom-tail-slice overlays of baseline vs. perturbed coefficients (bottom 100 beta values). Each row is a two-panel image with Coulliet (left) and Direct (right); the horizontal axis is the rank within the selected tail.

5

Discussion

The preceding chapters have established both the theoretical formulation and empirical validation of the proposed derivative-based regularization parameter selection rule for ridge regression in random-feature neural networks. Having presented the numerical and real-data results in Chapter 4, we now turn to a broader discussion of their implications.

This chapter aims to synthesize the key findings, interpret their meaning in light of the high - ridge regression framework. We further consider why the observed stability of the learned feature vectors $\boldsymbol{\beta}$ constitutes a practical advantage, in which contexts such stability may be desirable, and how it relates to interpretability and downstream reusability of the model. The discussion concludes with reflections on limitations, potential application domains, and directions for future research.

5.1. Summary of Findings

Let us start off by providing a short recap of the main experiments and their key findings once more.

5.1.1. Recap

The empirical evaluation conducted in Section 4.5 (Real Data Validation: Fashion-MNIST) provides a strong indication that the proposed derivative-based regularization parameter selection rule performs on par with, and in several aspects slightly better than, the deterministic-equivalent calibration method of Couillet [6].

On the binary Fashion-MNIST experiment, both calibration rules achieved nearly identical predictive performance, with test accuracies exceeding 99%. The derivative-based rule, however, yielded a marginally higher median test accuracy (99.56% vs. 99.30% for Couillet's rule) while exhibiting a reduced number of false negatives. This points to a slightly stronger recall and suggests that the proposed method does not sacrifice predictive power despite introducing an alternative criterion for γ selection.

More importantly, the perturbation analysis revealed a qualitative difference between the two approaches. Under small, controlled perturbations of the input data, models trained using the derivative-based calibration exhibited a notably more stable feature vector $\boldsymbol{\beta}$. In other words, the learned regression coefficients demonstrated lower sensitivity to localized variations in the training data, while remaining responsive to semantically meaningful structure within the input images. This stability implies a form of regularization that constrains model variance without suppressing relevant discriminative information.

Together, these findings highlight two key outcomes. First, the proposed method maintains competitive accuracy in a realistic, high-dimensional setting. Second, and perhaps more significantly, it yields solutions that are less sensitive to data perturbations, producing feature representations that are both robust and interpretable. This combination - predictive equivalence and enhanced stability - forms the basis for the broader discussion of the method's advantages, potential applications, and implications that follows in the subsequent sections.

5.1.2. Interpretations of the Results

The comparative analysis between the two calibration rules - Couillet's deterministic-equivalent criterion and the proposed derivative-based rule - reveals an interesting distinction not in terms of predictive performance, but in the qualitative behavior of the learned model parameters. Specifically, while both methods converge to similar predictive accuracies, the derivative-based direct loss calibration produces a more stable and consistent feature vector $\boldsymbol{\beta}$ under data perturbations.

This stability can be interpreted as a reduction in the variance component of the estimator. In high-dimensional ridge regression, the regularization parameter γ serves to balance the bias-variance trade-off: smaller values of γ allow the model to fit the data more closely but increase sensitivity to noise, whereas larger values enforce smoother, more regularized solutions. The proposed derivative-based rule tends to select slightly larger γ values compared to Couillet's approach (e.g., $\gamma_{\mathsf{D}} \approx 295$ versus $\gamma_{\mathsf{C}} \approx 71$ in the conducted Fashion-MNIST experiment), leading to a coefficient vector that varies less sharply with changes in the input data. Consequently, the resulting model exhibits reduced parameter variance, thereby enhancing robustness to perturbations and numerical instabilities without compromising generalization.

The perturbation experiments conducted in subsection 4.5.2 support this interpretation. When small localized changes were introduced to the training samples, the model calibrated using the derivative-based rule showed less adjustments in $\boldsymbol{\beta}$, whereas Couillet's calibration produced noticeably more fluctuation in the same coefficients. Yet, this increased stability did not translate into underfitting: the model remained sensitive to meaningful regions of the input space (for instance, the discriminative edges in the Fashion-MNIST digits) and maintained comparable test performance. This suggests that the proposed rule enforces a selective form of regularization - one that suppresses sensitivity to noise and irrelevant variations while preserving responsiveness to the underlying signal. Nevertheless, it is important to take into account a rather minimalistic limited-complexity setting of the experiment - performing a ReLU-activated binary classification using ridge regression on fashion-MNIST dataset.

In summary, the results suggest that the main distinction between the two calibration strategies lies not in predictive accuracy but in their regularization dynamics. The derivative-based rule biases the estimator toward smoother, more reproducible solutions-a property that may become particularly advantageous in high-dimensional problems where interpretability, reproducibility, or robustness to measurement noise are of primary concern.

5.2. Advantages of the Proposed Derivative-Based Calibration over Couillet's Rule

In this subsection we will discuss the benefits that a more stable learned feature vector gives us, settings in which these benefits might be crucial and any potential real-world applications of the developed method.

5.2.1. Conceptual Advantages - Theory

Interpretability. In the context of random-feature ridge regression, "interpretability" should not be understood in the classical sense of direct feature attribution, since the random mapping $\sigma(WX)$ obscures the original input-space coordinates. Instead, interpretability arises in a relative sense: a model whose coefficients remain consistent under repeated experiments or mild perturbations can be more reliably analyzed, visualized, and reasoned about.

A stable β implies that the model's internal representation - the effective filter applied to the random features - is not an artifact of random initialization or particular noise realizations. This consistency makes the model's decision mechanism more transparent and trustworthy: one can meaningfully inspect the average structure of β across runs, or study which directions in the feature space are systematically emphasized. By contrast, if β changes substantially with each retraining, any attempt to interpret or visualize the learned representation becomes unreliable. Thus, stability becomes a prerequisite for interpretability in this high-dimensional random-feature setting.

Reusability of the Learned Coefficients. A related advantage concerns the potential for *downstream reuse* of the learned coefficients β. In many applications, the ridge-regression layer serves as an intermediate component within a larger processing pipeline - for instance, as a feature extractor whose output is later used for clustering, transfer learning, or anomaly detection. When β is stable, it can be reused across slightly different datasets or experimental conditions without the need for retuning the regularization parameter or retraining from scratch. This property enables more reproducible and modular system design: the same model component can be integrated into different downstream tasks with predictable behavior.

In contrast, coefficients obtained through the Couillet calibration tend to vary more across retrainings, which limits their reuse beyond the specific dataset or random feature realization. The derivative-based rule, by reducing variance in β, implicitly supports the notion of a reusable "filter" that captures the core discriminative structure of the data rather than overfitting to local peculiarities.

5.2.2. Applied Advantages - Practice

Having established the conceptual benefits of stability in the preceding subsection, it is instructive to examine how this property manifests in practical, high-dimensional learning problems. In various real-world domains, researchers have identified coefficient or feature stability as a crucial component of model reliability, reproducibility, and interpretability. Stability ensures that the insights drawn from a model are not artifacts of random sampling or noise, but rather reflect persistent structure within the data itself. In this subsection, several representative studies are discussed to illustrate how the stability of learned representations, filters, or feature selections directly impacts the quality and trustworthiness of machine learning systems in applied contexts.

Feature Stability in High-Dimensional Learning. The empirical perspective on stability is thoroughly examined by Kalousis *et al.* [22], who conducted one of the earliest systematic analyses of feature selection stability across high-dimensional biological datasets. Their experiments covered several proteomics and microarray classification tasks, all characterized by a small number of samples ($n < 100$) and a very large number of features (p ranging from thousands to tens of thousands) - a

regime directly analogous to the high-dimensional random-feature setting studied in this thesis.

The authors evaluated a range of popular feature selection methods, including information gain, χ^2, symmetrical uncertainty, RELIEF, and support-vector-based ranking techniques (SVM and SVM-RFE). To quantify the sensitivity of each method to sampling variations, they introduced three complementary similarity measures for feature preferences: (i) S_W for correlation of feature weights, (ii) S_R for rank consistency via Spearman's ρ, and (iii) S_S for overlap between top-k selected feature subsets. These measures collectively define a "stability profile" for each algorithm, summarizing how much the selected features change when the model is retrained on different resampled subsets of the same data.

The results were striking. Even among algorithms that achieved nearly identical classification accuracy, stability varied dramatically. For instance, on the colon cancer dataset ($p = 2000$, $n = 62$), SVM-RFE exhibited overlap values S_S below 0.25 for top-$k = 30$ features, while simpler univariate methods such as information gain reached $S_S > 0.9$ under the same setup. Similar patterns appeared across all five datasets: in the ovarian and leukemia studies, univariate filters consistently achieved overlap above 0.85, while multivariate SVM-based selectors fluctuated heavily with small data perturbations, dropping to as low as $S_S = 0.1$. These results demonstrated that identical predictive scores can conceal highly unstable internal mechanisms.

From the perspective of this thesis, the analogy is clear. In the random-feature ridge regression studied here, two regularization calibration rules (Couillet's and the derivative-based one) may yield comparable prediction errors, yet differ sharply in the variance of their learned coefficients β. The derivative-based calibration thus plays a role analogous to preferring the more stable feature selector in Kalousis *et al.*: it favors solutions whose internal structure, the effective "filter" or feature weighting vector remains consistent under mild resampling or noise perturbations. Just as Kalousis and colleagues argued that stability is a prerequisite for reproducible and interpretable conclusions in biomedical settings, a stable β ensures reproducibility and trust in the learned representations in high-dimensional regression.

In short, the empirical message of Kalousis *et al.* reinforces the same principle that underlies this work: when dimensionality is high and data are scarce, predictive accuracy alone is not a sufficient measure of reliability. The internal stability of the learned coefficients becomes a distinguishing marker of models that capture genuine, reproducible structure rather than transient artifacts of random variation.

Stability and Redundancy in Neuroimaging Analysis. A second, domain-specific example of the importance of model stability is offered by Wang *et al.* [49], who addressed the problem of stable feature selection in extremely high-dimensional functional MRI (fMRI) data. This example is particularly interesting for our case as it also works with images. In such neuroimaging tasks, the number of voxels or network connections ($p \sim 10^3$–10^4) far exceeds the number of available subjects ($n \lesssim 100$), and the features are often highly correlated or redundant. The study proposed an algorithm combining *stability selection* with the *elastic net*, aiming not only to maintain high predictive accuracy but also to obtain reproducible and interpretable "biomarker" features that remain consistent across perturbations, label noise, and acquisition sites.

Wang *et al.* demonstrated that purely sparse methods such as ℓ_1-regularized logistic regression or SVM yield unstable and overly sparse solutions: when applied to fMRI voxels, they select scattered and inconsistent sets of discriminative features, sensitive to even minor variations in the data. By integrating stability selection (which aggregates results over subsamples of both samples and features) with elastic-net regularization (which enforces group-wise feature retention), their model achieved markedly higher robustness and interpretability. For example, in the synthetic dataset with injected label noise (up to ten wrong labels), their method maintained a nearly constant voxel-selection accuracy, while alternatives such as the ℓ_1-logistic model degraded sharply. Quantitatively, even with 10% label corruption, their approach preserved the correct identification of discriminative subregions, whereas univariate t-tests and standard SVMs produced false-positive activations or lost key regions entirely.

The advantages extended to real neuroimaging data. In a face-recognition fMRI experiment involving 26 subjects, Wang *et al.* successfully recovered five core regions implicated in visual and motor

aspects of facial processing - the occipital face area (OFA), fusiform face area (FFA), posterior superior temporal gyrus (pSTG), supplementary motor area (SMA), and sensorimotor cortex (SMC)- while competing methods either missed some of these regions or produced fragmented, spatially inconsistent activations. Furthermore, in a multi-center ADHD dataset where training and test data were collected at separate institutions (Peking University and New York University), the proposed method achieved a cross-center classification accuracy of 79.0% (AUC = 0.77), outperforming standard elastic net (72.6%) and randomized ℓ_1-logistic models (67.7%). The ability to generalize across centers highlights the robustness of the learned feature weighting vector under substantial data variation.

The parallels to the present thesis are clear. In both cases, stability of the learned coefficients - whether interpreted as voxel importances in fMRI or as the regression filter β in random-feature ridge regression serves as a safeguard against spurious structure and noise-driven variability. Just as Wang *et al.* showed that stable, redundant feature selection improves both interpretability and cross-domain robustness, the derivative-based calibration proposed in this work yields a β that remains consistent across perturbations, enabling more reproducible downstream analyses. In settings where the data are high-dimensional, correlated, and noisy, such stability directly translates into both scientific trustworthiness and operational reliability.

Stable Representation Learning in Industrial Fault Detection. The work of Michau and Fink [33] provides a particularly relevant example of how stability of internal representations translates into both interpretability and operational reliability in large-scale systems. Their study focuses on condition monitoring of industrial assets, where hundreds of heterogeneous sensors continuously record operating parameters under varying conditions. The challenge addressed is that only healthy-state data are typically available for training, while fault patterns are scarce or entirely absent. To handle this, the authors proposed an unsupervised feature learning and one-class classification architecture based on *Hierarchical Extreme Learning Machines* (HELM), integrating a stacked autoencoder for representation learning with a one-class classifier trained on the latent features.

The HELM framework can be seen as a random-feature architecture with analytical ridge-type weight estimation, conceptually close to the models investigated in this thesis. It learns a mapping from raw sensor space to a latent "health indicator," defined as the distance of the observation to the manifold of healthy data. Importantly, the system is trained with random input weights A, and its output coefficients β are determined in closed form through regularized least-squares (ridge or LASSO). This allows an explicit analysis of how the stability of β affects the consistency of the learned health indicators and, consequently, the reliability of fault detection decisions.

In their extensive experiments, Michau and Fink compared HELM with six alternative approaches—standalone one-class ELM and SVM classifiers, the same classifiers preceded by PCA, and a Deep Belief Network (DBN). On a simulated dataset with $D = 200$ sensors and five injected fault types, HELM achieved the highest average accuracy (95% for $n = 5$ intrinsic variables, $A_{\text{fault}} \approx 0.95$) and strong magnification coefficients (Mag ≈ 88–105), far exceeding the next-best competitors (SVM: $A \approx 0.94$, Mag ≈ 1.3; DBN: $A \approx 0.72$, Mag ≈ 1.2). These results quantitatively demonstrate that the feature-learning stage produces representations whose responses to perturbations—both random and structured—are highly consistent, yielding fault indicators that remain stable under varying operating conditions.

The findings are reinforced by the real-case application to a nine-month dataset from a hydrogen-cooled power plant generator with $D = 310$ sensors. HELM reached an accuracy of 95% with zero false positives and a magnification coefficient of 20, while the closest competitor (SVM) achieved 87% accuracy and Mag $= 1.04$. Importantly, HELM was able to detect early-stage degradation (day 169) months before the full short-circuit failure (day 247), providing a temporal margin that can translate directly into cost savings in maintenance and downtime prevention.

Conceptually, the relevance to this thesis lies in how HELM's stability emerges from a regularized random-feature mechanism. Like the derivative-based calibration introduced here, the HELM formulation stabilizes the learned β against small perturbations in the data, ensuring that the "health indicator" or model output evolves smoothly across retrainings. This stability, while mathematically grounded in

ridge and LASSO regularization, is functionally identical to the desired low-variance behavior of β in random-feature regression. In both contexts, the goal is not merely to predict correctly, but to ensure that the internal model structure—the coefficients themselves—remains reproducible and interpretable across runs, enabling engineers to trust the system's diagnosis or response over time and under changing conditions.

6

Conclusion

6.1. Summary and Final Remarks

This thesis introduced and analyzed a novel derivative-based, direct-loss calibration rule for selecting the regularization parameter in high-dimensional random-feature ridge regression. The method was developed as an alternative to the deterministic-equivalent canonical approach proposed by Couillet [6], offering a data-driven and theoretically grounded criterion derived directly from the loss-function dynamics.

The work began by revisiting the theoretical framework of ridge regression under random feature mappings, establishing the connection between the regularization parameter γ and the statistical bias–variance trade-off in high-dimensional settings. Building upon this foundation, a new calibration rule was formulated by enforcing the vanishing of the empirical loss derivative with respect to γ, yielding a direct, sample-based condition for optimal regularization.

The proposed method was validated through both synthetic and real-data experiments. In controlled simulations, the derivative-based rule achieved nearly identical predictive performance to Couillet's method calibration, confirming its theoretical consistency. Real-data evaluation on the Fashion-MNIST dataset further demonstrated that the proposed method matches or slightly surpasses Couillet's approach in terms of test accuracy, while producing a more stable coefficient vector β under perturbations of the training data. The perturbation analysis showed that this stability translates into smoother, more reproducible model behavior, which is an important property in high-dimensional problems where interpretability and robustness are as valuable as predictive accuracy.

A broader discussion connected these findings to practical contexts where model stability is a crucial requirement. By drawing parallels to studies on feature-selection stability in bioinformatics [22], neuroimaging [49], and industrial condition monitoring [33], the thesis highlighted that the benefits of coefficient stability extend far beyond synthetic benchmarks. In all such settings, the ability to learn consistent and reproducible representations enables more interpretable, trustworthy, and reusable models.

In summary, this research contributes both a new ridge regression hyperparameter calibration methodology and an expanded understanding of the role of stability in high-dimensional learning. The derivative-based rule provides a simple yet effective alternative to deterministic-equivalent approaches, combining strong empirical performance with enhanced reproducibility and interpretability of the learned coefficients.

6.2. Future Work

The experimental and computational framework adopted in this thesis was deliberately kept minimal in order to isolate and examine the behavior of the proposed calibration rule under controlled conditions. While this design allowed for a clear theoretical and empirical comparison with Couillet's deterministic-equivalent method, it also leaves several directions open for future exploration.

First, the dimensional ratios employed in the simulations denoted as $c_1 = p/n$ and $c_2 = N/n$ were limited to values not exceeding approximately 1.5, primarily due to computational constraints on the DelftBlue supercomputer [1]. As a result, the synthetic experiments operated on moderate data sizes, sufficient for statistical validity but not for fully exploring the asymptotic regime where random matrix predictions become most distinctive. Extending these experiments to higher dimensional ratios and larger sample sizes would help verify whether the observed stability and generalization properties of the derivative-based rule persist under more extreme high-dimensional conditions.

Second, the real-data validation was restricted to a single dataset (Fashion-MNIST) and to a binary classification task involving two visually distinct classes. Future work could expand the empirical validation along several complementary axes. Additional real-world datasets. For instance, those from text, sensor, or biomedical domains - would allow assessing the robustness of the proposed rule across modalities. Similarly, multi-class classification scenarios could reveal how the calibration behaves when the decision boundaries interact in more complex ways.

Another natural extension concerns the activation function used in the random feature mapping. All experiments in this work employed the ReLU nonlinearity, chosen for its analytical simplicity and widespread use. Investigating alternative activations, such as leaky ReLU, tanh, or even randomized polynomial kernels, could uncover how the choice of nonlinearity influences the derivative-based calibration and its resulting stability properties.

Finally, while the present study focused on classification, future research could revisit the same framework in a genuine regression context. Testing the proposed calibration rule on real-valued target problems would help establish whether its stabilizing effect on the learned coefficients $\boldsymbol{\beta}$ extends beyond classification tasks and remains beneficial in continuous-output ridge regression. Identifying suitable high-dimensional regression datasets would therefore be an important next step toward a more complete empirical characterization of the method.

In summary, future work should aim to broaden both the scale and scope of experimentation - ncreasing sample and feature counts, diversifying nonlinear mappings, and extending to regression settings to consolidate the findings of this thesis and to fully explore the potential of the derivative-based regularization rule in modern high-dimensional learning.

References

[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.

[2] Zhidong Bai and Jack W. Silverstein. *Spectral Analysis of Large Dimensional Random Matrices*. 2nd. Springer, 2010. DOI: https://doi.org/10.1007/978-1-4419-0661-8.

[3] Mikhail Belkin et al. "Reconciling Modern Machine Learning Practice and the Classical Bias–Variance Trade-off". In: *Proceedings of the National Academy of Sciences* 116.32 (2019), pp. 15849–15854. DOI: https://doi.org/10.1073/pnas.1903070116.

[4] Youngmin Cho and Lawrence K. Saul. "Kernel Methods for Deep Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 22. 2009, pp. 342–350. URL: https://proceedings.neurips.cc/paper_files/paper/2009/hash/013a006f03dbc5392effeb8f18fda755-Abstract.html.

[5] Romain Couillet and Mérouane Debbah. *Random Matrix Methods for Wireless Communications*. Cambridge University Press, 2011. DOI: https://doi.org/10.1017/CBO9780511994746.

[6] Romain Couillet and Zhenyu Liao. "Random Matrix Methods for Machine Learning". In: Cambridge University Press, 2022, pp. 299–306. DOI: https://doi.org/10.1017/9781009128490.

[7] Romain Couillet and Zhenyu Liao. *Random Matrix Methods for Machine Learning*. Cambridge University Press, 2022. DOI: https://doi.org/10.1017/9781009128490.

[8] George Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314. DOI: https://doi.org/10.1007/BF02551274.

[9] Jia Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2009), pp. 248–255. DOI: https://doi.org/10.1109/CVPR.2009.5206848.

[10] Edgar Dobriban and Weijie Su. "High-Dimensional Asymptotics of Prediction: Ridge Regression and Classification". In: *The Annals of Statistics* 46.1 (2018), pp. 247–279. DOI: https://doi.org/10.1214/17-AOS1549.

[11] Jeffrey L. Elman. "Finding Structure in Time". In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: https://doi.org/10.1207/s15516709cog1402_1.

[12] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman and Hall/CRC, 2019. DOI: https://doi.org/10.1201/9780429447273.

[13] Trevor Hastie et al. "Surprises in High-Dimensional Ridgeless Least Squares Interpolation". In: *The Annals of Statistics* 50.2 (2022), pp. 949–986. DOI: https://doi.org/10.1214/21-AOS2133.

[14] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90.

[15] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Computation* 18.7 (2006), pp. 1527–1554. DOI: https://doi.org/10.1162/neco.2006.18.7.1527.

[16] Arthur E. Hoerl and Robert W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 55–67. DOI: https://doi.org/10.1080/00401706.1970.10488634.

[17] John J. Hopfield. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities”. In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. DOI: https://doi.org/10.1073/pnas.79.8.2554.

[18] Kurt Hornik. “Approximation Capabilities of Multilayer Feedforward Networks”. In: *Neural Networks* 4.2 (1991), pp. 251–257. DOI: https://doi.org/10.1016/0893-6080(91)90009-T.

[19] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. “Extreme Learning Machine: Theory and Applications”. In: *Neurocomputing* 70.1–3 (2006), pp. 489–501. DOI: https://doi.org/10.1016/j.neucom.2005.12.126.

[20] Arthur Jacot, Franck Gabriel, and Clément Hongler. “Neural Tangent Kernel: Convergence and Generalization in Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)* 31 (2018). URL: https://arxiv.org/abs/1806.07572.

[21] Herbert Jaeger. “The “Echo State” Approach to Analysing and Training Recurrent Neural Networks”. In: *GMD Report* 148 (2001). URL: https://www.researchgate.net/publication/215385037.

[22] Alexandros Kalousis, Julien Prados, and Melanie Hilario. “Stability of Feature Selection Algorithms: A Study on High-Dimensional Spaces”. In: *Knowledge and Information Systems* 12.1 (2007), pp. 95–116. DOI: https://doi.org/10.1007/s10115-006-0040-8. URL: https://link.springer.com/article/10.1007/s10115-006-0040-8.

[23] Teuvo Kohonen. “Self-Organized Formation of Topologically Correct Feature Maps”. In: *Biological Cybernetics* 43.1 (1982), pp. 59–69. DOI: https://doi.org/10.1007/BF00337288.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 25. 2012, pp. 1097–1105. URL: https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), pp. 436–444. DOI: https://doi.org/10.1038/nature14539.

[26] Yann LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: https://doi.org/10.1162/neco.1989.1.4.541.

[27] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: https://doi.org/10.1109/5.726791.

[28] Cosme Louart, Zhenyu Liao, and Romain Couillet. “A Random Matrix Approach to Neural Networks”. In: *The Annals of Applied Probability* 28.2 (2018), pp. 1190–1248. DOI: https://doi.org/10.1214/17-AAP1328.

[29] Cosme Louart, Zhenyu Liao, and Romain Couillet. “A random matrix approach to neural networks”. In: *The Annals of Applied Probability* (2018). DOI: https://doi.org/10.1214/17-AAP1328.

[30] Wolfgang Maass, Thomas Natschläger, and Henry Markram. “Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations”. In: *Neural Computation* 14.11 (2002), pp. 2531–2560. DOI: https://doi.org/10.1162/089976602760407955.

[31] Vladimir A. Marchenko and Leonid A. Pastur. “Distribution of Eigenvalues for Some Sets of Random Matrices”. In: *Mathematics of the USSR-Sbornik* 1.4 (1967), pp. 457–483. DOI: https://doi.org/10.1070/SM1967v001n04ABEH001994.

[32] Song Mei and Andrea Montanari. “The Generalization Error of Random Features Regression: Precise Asymptotics and Double Descent Curve”. In: *Communications on Pure and Applied Mathematics* 75.4 (2022), pp. 667–766. DOI: https://doi.org/10.1002/cpa.21937.

[33] Gabriel Michau and Olga Fink. “Feature Learning for Fault Detection in High-Dimensional Condition-Monitoring Signals”. In: *arXiv preprint arXiv:1810.05550* (2019). URL: https://arxiv.org/abs/1810.05550.

[34] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.

[35] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 2010, pp. 807–814. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[36] Radford M. Neal. *Bayesian Learning for Neural Networks*. Vol. 118. Lecture Notes in Statistics. Springer, 1996. DOI: https://doi.org/10.1007/978-1-4612-0745-0.

[37] Ian Osband et al. “Deep Exploration via Bootstrapped DQN”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 29. 2016, pp. 4026–4034. URL: https://proceedings.neurips.cc/paper_files/paper/2016/hash/8d8818c8ce75b17d62e3e0fbde7d0b5c-Abstract.html.

[38] Jeffrey Pennington and Pratik Worah. “Nonlinear Random Matrix Theory for Deep Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017, pp. 2637–2646. URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/6070ef0e056174b4a3d8d005a36cf3c9-Abstract.html.

[39] Ali Rahimi and Benjamin Recht. “Random Features for Large-Scale Kernel Machines”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 20. 2008, pp. 1177–1184. URL: https://proceedings.neurips.cc/paper_files/paper/2007/hash/013a006f03dbc5392effeb8f18fda755-Abstract.html.

[40] Ali Rahimi and Benjamin Recht. “Weighted Sums of Random Kitchen Sinks: Replacing Minimization with Randomization in Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 21. 2008, pp. 1313–1320. URL: https://proceedings.neurips.cc/paper_files/paper/2008/hash/0efe32849d230d7f53049ddc4a4b0c60-Abstract.html.

[41] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: https://doi.org/10.1037/h0042519.

[42] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: https://doi.org/10.1038/323533a0.

[43] Andrew Saxe et al. “On Random Weights and Unsupervised Feature Learning”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. 2011, pp. 1089–1096. URL: https://icml.cc/Conferences/2011/papers/438_icmlpaper.pdf.

[44] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *International Conference on Learning Representations (ICLR)* (2015). URL: https://arxiv.org/abs/1409.1556.

[45] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: https://jmlr.org/papers/v15/srivastava14a.html.

[46] Andrey N. Tikhonov and Vasiliy Y. Arsenin. *Solution of Ill-Posed Problems*. Originally published in Russian, 1963. Winston and Sons, 1977.

[47] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017. URL: https://arxiv.org/abs/1706.03762.

[48] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *Journal of Machine Learning Research* 11 (2010), pp. 3371–3408. URL: https://www.jmlr.org/papers/v11/vincent10a.html.

[49] Yilun Wang et al. “A Novel Approach for Stable Selection of Informative Redundant Features from High Dimensional fMRI Data”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 24.11 (2015), pp. 1236–1248. DOI: https://doi.org/10.1109/TNSRE.2015.2474015. URL: https://arxiv.org/abs/1506.08301.

[50] Eugene P. Wigner. “Characteristic Vectors of Bordered Matrices with Infinite Dimensions”. In: *Annals of Mathematics* 62.3 (1955), pp. 548–564. DOI: https://doi.org/10.2307/1970079.

A

Source Code

A.1. Mean Squared Error against the regularization parameter γ

```
import numpy as np
import random
from scipy import linalg
from mpi4py import MPI

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
myrank = comm.Get_rank()


def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    if activation_function == 'linear':
        return t

    if activation_function == 'ReLu':
        return np.maximum(t, 0)

    if activation_function == 'sign':
        return np.sign(t)



def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    if activation_function == 'linear':
        return x.T@y

    if activation_function == 'ReLu':
        norm_x = np.linalg.norm(x, axis=0)  # Shape (n_x,)
        norm_y = np.linalg.norm(y, axis=0)  # Shape (n_y,)

        xTy = x.T @ y  # Shape (n_x, n_y)

        norm_prod = norm_x[:, np.newaxis] * norm_y[np.newaxis, :]  # Shape (n_x, n_y)
        norm_prod = np.maximum(norm_prod, 1e-10)  # Avoid division by zero
```

```
46
47          cos_theta = xTy / norm_prod
48          cos_theta = np.clip(cos_theta, -1 + 1e-10, 1 - 1e-10)  # Clamp values to [-1+1e-10,
                1-1e-10]
49
50          theta = np.arccos(-cos_theta)  # Shape (n_x, n_y)
51          sin_theta = np.sqrt(1 - cos_theta ** 2)
52
53          return (norm_prod) / (2 * np.pi) * (cos_theta * theta + sin_theta)
54
55      if activation_function == 'sign':
56          norm_x = np.linalg.norm(x, axis=0)  # Shape (n_x,)
57          norm_y = np.linalg.norm(y, axis=0)  # Shape (n_y,)
58
59          xTy = x.T @ y  # Shape (n_x, n_y)
60
61          norm_prod = np.outer(norm_x, norm_y)  # Shape (n_x, n_y)
62          norm_prod = np.maximum(norm_prod, 1e-10)  # Avoid division by zero
63
64          cos_theta = xTy / norm_prod
65          cos_theta = np.clip(cos_theta, -1 + 1e-10, 1 - 1e-10)  # Clamp values to [-1+1e-10,
                1-1e-10]
66
67          return (2 / np.pi) * np.arcsin(cos_theta)
68
69
70  def K_(x, y, delta, N, n_train):
71      """
72      Function to compute kernel approximation
73      :param x: first 'point' set
74      :param y: second 'point' set
75      :param delta: delta parameter
76      :return: approximation for the kernel matrix
77      """
78      k_ = (N/n_train)*((K(x, y))/(1+delta))
79      return k_
80
81
82  def Etrain(gamma, data, npN, monte_carlo_loops=30):
83      """
84      Actual training error compute (random-based)
85      :param gamma: ridge penalty value
86      :param data: train/test data
87      :param npN: dimensionality of the data and num of neurons
88      :param monte_carlo_loops: number of iterations for E value averaging for different
            generated W
89      :return: Training error
90      """
91      # Unpacking variables
92      X_train, X_test, Y_train, Y_test = data
93      n_train, n_test, p, N = npN
94
95      E_train_arr = []
96      p = X_train.shape[0]
97
98      for i in range(monte_carlo_loops):
99
100         # W = np.random.randn(N, p)
101         # Sigm = sigma(W@X_train)
102         # Q_y = np.linalg.inv((1/n_train)*Sigm.T@Sigm + gamma*np.eye(n_train))
103         #
104         # E_train = (gamma*gamma/n_train)*Y_train@np.linalg.matrix_power(Q_y, 2)@Y_train.T
105         # E_train_arr.append(E_train)
106
107         W = np.random.randn(N, p)
108         Sigm = sigma(W @ X_train)
109         Sigm_ = sigma(W @ X_test)
110
111         inv_tQ_r = linalg.solve(Sigm.T @ Sigm / n_train + gamma * np.eye(n_train), Y_train)
112         beta = Sigm / n_train @ inv_tQ_r
113
```

```
        E_train = np.linalg.norm(Y_train-Sigm.T@beta)**2/n_train
        E_train_arr.append(E_train)

    return np.mean(np.array(E_train_arr))


def Etest(gamma, data, npN, monte_carlo_loops=30):
    """
    Actual test error compute (random-based)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and number of neurons
    :param monte_carlo_loops: number of iterations for E value averaging for different
        generated W
    :return: Test error
    """
    # Unpacking variables
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    E_test_arr = []
    p = X_train.shape[0]

    for i in range(monte_carlo_loops):

        W = np.random.randn(N, p)
        Sigm = sigma(W@X_train)
        Sigm_ = sigma(W@X_test)

        inv_tQ_r = linalg.solve(Sigm.T@Sigm/n_train + gamma * np.eye(n_train), Y_train)
        beta = Sigm/n_train @ inv_tQ_r

        # Q_y = np.linalg.inv((1 / n_train) * Sigm.T @ Sigm + gamma * np.eye(n_train))

        # term1 = (1/n_test)*(Y_test@Y_test.T)
        # term2 = (2/(n_train*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Y_test.T)
        # term3 = (1/(n_train**2*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Sigm_.T@Sigm@Q_y@Y_train.
            T)


        E_test = np.linalg.norm(Y_test-Sigm_.T@beta)**2/n_test
        E_test_arr.append(E_test)

    ans = np.mean(np.array(E_test_arr))

    return ans


def find_delta(gamma, X_train, N, accuracy) -> float:
    """
    Helper-function that finds delta parameter for the resolvent Q iteratively
    :param gamma: ridge penalty value
    :param X_train: training set
    :param N: number of neurons in the hidden layer
    :param accuracy: accuracy for numerical delta finding
    :return: optimal value for delta parameter for the resolvent Q
    """
    n_train = X_train.shape[1]
    delta_prev = 1
    delta_next = 0
    while abs(delta_prev-delta_next) > accuracy:
        delta_prev = delta_next
        Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta_next)) + gamma*np.eye(
            n_train))
        delta_next = (1/n_train)*(np.trace(Q_@K(X_train, X_train)))
    return delta_next


def Etrain_(gamma, data, npN):
    """
```

```
    Estimated train error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated training error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = find_delta(gamma, X_train, N, delta_accuracy)
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))

    #E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.matrix.trace(Q_@K_@Q_))/((1 - 1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
    E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.trace(Q_@K_@Q_))/((1 - 1/N)*np.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)

    return E_train_


def Etest_(gamma, data, npN):
    """
    Estimated test error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated test error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = 0.0
    delta = find_delta(gamma, X_train, N, delta_accuracy)
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
    K_xX = (N/n_train)*(K(X_train, X_test)/(1+delta))
    K_XX = (N/n_train)*(K(X_test, X_test)/(1+delta))

    #E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.matrix.trace(K_XX) - (1/n_test)*np.matrix.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))
    E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.trace(K_XX) - (1/n_test)*np.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))

    return E_test_


def generate_synthetic_data(n, c1, c2, density=0.35, noise_level=0.1):
    p = int(c1 * n)
    N = int(c2 * n)
    n_train = int(n * 0.8)
    n_test = n - n_train

    # Generate random weights and sparse beta
    W = np.random.randn(N, p)
    b = np.random.choice([0, 1], size=(N, 1), p=[1 - density, density])

    # Generate data
    X_train = np.random.randn(p, n_train)
    Y_train = (sigma(W @ X_train).T @ b).flatten() + np.random.normal(0, noise_level, n_train)
    X_test = np.random.randn(p, n_test)
    Y_test = (sigma(W @ X_test).T @ b).flatten() + np.random.normal(0, noise_level, n_test)

    return X_train, X_test, Y_train, Y_test, W, b

```

```
245
246 def save_results_to_file(gammas, E_test_arr, E_train_arr, E_test_bar_arr, E_train_bar_arr,
        log_filename, result_filename):
247     """
248     A function designed for HPC to save the results in proper readable and plotable form.
249     :param gammas:
250     :param E_test_arr:
251     :param E_train_arr:
252     :param E_test_bar_arr:
253     :param E_train_bar_arr:
254     :param log_filename:
255     :param result_filename:
256     :return: none
257     """
258     with open(log_filename, 'w') as log_file:
259         for gamma, E_test, E_train, E_test_, E_train_ in zip(gammas, E_test_arr, E_train_arr,
                E_test_bar_arr,
260                                                              E_train_bar_arr):
261             log_file.write(
262                 f"for␣gamma␣=␣{gamma:.6e},␣E_train␣=␣{E_train:.6e},␣E_test␣=␣{E_test:.6e},␣
                        E_train_␣=␣{E_train_:.6e},␣E_test_␣=␣{E_test_:.6e}\n")
263
264     results = {
265         "gammas": gammas,
266         "E_test_arr": E_test_arr,
267         "E_train_arr": E_train_arr,
268         "E_test_bar_arr": E_test_bar_arr,
269         "E_train_bar_arr": E_train_bar_arr
270     }
271     np.savez(result_filename, **results)
272
273
274 activation_function = 'ReLu'
275
276 delta_accuracy = 1e-4
277 b_vector_density = 0.15 #%
278
279 n = 5000
280 c1 = 1.0 # p/n
281 c2 = 0.8 # N/n
282
283 p = int(c1 * n)
284 N = int(c2 * n)
285 n_train = int(n * 0.8)
286 n_test = n - n_train
287
288 npN = n_train, n_test, p, N
289 X_train, X_test, Y_train, Y_test, W, b = generate_synthetic_data(n=n, c1=c1, c2=c2, density=
        b_vector_density)
290 data = X_train, X_test, Y_train, Y_test
291 # data = get_data_MNIST_binary(npN)
292 gammas = [10**y for y in np.arange(-6, 6, 0.1)]
293
294 gamma_chunks = np.array_split(gammas, nprocs)
295 my_gammas = gamma_chunks[myrank]
296
297 E_test_arr = []
298 E_train_arr = []
299 E_test_bar_arr = []
300 E_train_bar_arr = []
301 log_entries = []
302
303 for g in my_gammas:
304     E_test = Etest(g, data, npN)
305     E_test_arr.append(E_test)
306
307     E_test_ = Etest_(g, data, npN)
308     E_test_bar_arr.append(E_test_)
309
310     E_train = Etrain(g, data, npN)
311     E_train_arr.append(E_train)
```

```
312
313     E_train_ = Etrain_(g, data, npN)
314     E_train_bar_arr.append(E_train_)
315
316     log_entries.append(
317         f"for␣gamma␣=␣{g}␣the␣error␣values␣are:␣E_train={E_train},␣E_test={E_test},␣E_train_
                =={E_train_},␣E_test_={E_test_}\n")
318
319 E_test_arr = comm.gather(E_test_arr, root=0)
320 E_train_arr = comm.gather(E_train_arr, root=0)
321 E_test_bar_arr = comm.gather(E_test_bar_arr, root=0)
322 E_train_bar_arr = comm.gather(E_train_bar_arr, root=0)
323 log_entries = comm.gather(log_entries, root=0)
324
325 if myrank == 0:
326     E_test_arr = [item for sublist in E_test_arr for item in sublist]
327     E_train_arr = [item for sublist in E_train_arr for item in sublist]
328     E_test_bar_arr = [item for sublist in E_test_bar_arr for item in sublist]
329     E_train_bar_arr = [item for sublist in E_train_bar_arr for item in sublist]
330     log_entries = [item for sublist in log_entries for item in sublist]
331
332     np.savez("results.npz",
333              gammas=gammas,
334              E_test_arr=E_test_arr,
335              E_train_arr=E_train_arr,
336              E_test_bar_arr=E_test_bar_arr,
337              E_train_bar_arr=E_train_bar_arr,
338              activation_function=activation_function,
339              density=b_vector_density,
340              n_train=n_train,
341              n_test=n_test,
342              N=N,
343              p=p)
344
345     with open("log.txt", "w") as f:
346         f.writelines(log_entries)
```

A.2. Optimal γ value and error behaviour with increasing amount of data

```python
import numpy as np
import random
from scipy import linalg
from mpi4py import MPI
import os
os.environ["OMP_NUM_THREADS"] = "16"

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
myrank = comm.Get_rank()


def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    return t


def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    return x.T@y


def K_(x, y, delta, N, n_train):
    """
    Function to compute kernel approximation
    :param x: first 'point' set
    :param y: second 'point' set
    :param delta: delta parameter
    :return: approximation for the kernel matrix
    """
    k_ = (N/n_train)*((K(x, y))/(1+delta))
    return k_


def Etrain(gamma, data, npN, monte_carlo_loops=30):
    """
    Actual training error compute (random-based)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and num of neurons
    :param monte_carlo_loops: number of iterations for E value averaging for different
        generated W
    :return: Training error
    """
    # Unpacking variables
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    E_train_arr = []
    p = X_train.shape[0]

    for i in range(monte_carlo_loops):

        # W = np.random.randn(N, p)
        # Sigm = sigma(W@X_train)
        # Q_y = np.linalg.inv((1/n_train)*Sigm.T@Sigm + gamma*np.eye(n_train))
        #
        # E_train = (gamma*gamma/n_train)*Y_train@np.linalg.matrix_power(Q_y, 2)@Y_train.T
```

```
        # E_train_arr.append(E_train)

        W = np.random.randn(N, p)
        Sigm = sigma(W @ X_train)
        Sigm_ = sigma(W @ X_test)

        inv_tQ_r = linalg.solve(Sigm.T @ Sigm / n_train + gamma * np.eye(n_train), Y_train)
        beta = Sigm / n_train @ inv_tQ_r

        E_train = np.linalg.norm(Y_train-Sigm.T@beta)**2/n_train
        E_train_arr.append(E_train)

    return np.mean(np.array(E_train_arr))


def Etest(gamma, data, npN, monte_carlo_loops=30):
    """
    Actual test error compute (random-based)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and number of neurons
    :param monte_carlo_loops: number of iterations for E value averaging for different
        generated W
    :return: Test error
    """
    # Unpacking variables
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    E_test_arr = []
    p = X_train.shape[0]

    for i in range(monte_carlo_loops):

        W = np.random.randn(N, p)
        Sigm = sigma(W@X_train)
        Sigm_ = sigma(W@X_test)

        inv_tQ_r = linalg.solve(Sigm.T@Sigm/n_train + gamma * np.eye(n_train), Y_train)
        beta = Sigm/n_train @ inv_tQ_r

        # Q_y = np.linalg.inv((1 / n_train) * Sigm.T @ Sigm + gamma * np.eye(n_train))

        # term1 = (1/n_test)*(Y_test@Y_test.T)
        # term2 = (2/(n_train*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Y_test.T)
        # term3 = (1/(n_train**2*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Sigm_.T@Sigm@Q_y@Y_train.
            T)


        E_test = np.linalg.norm(Y_test-Sigm_.T@beta)**2/n_test
        E_test_arr.append(E_test)

    ans = np.mean(np.array(E_test_arr))

    return ans


def find_delta(gamma, X_train, N, accuracy) -> float:
    """
    Helper-function that finds delta parameter for the resolvent Q iteratively
    :param gamma: ridge penalty value
    :param X_train: training set
    :param N: number of neurons in the hidden layer
    :param accuracy: accuracy for numerical delta finding
    :return: optimal value for delta parameter for the resolvent Q
    """
    n_train = X_train.shape[1]
    delta_prev = 1
    delta_next = 0
    while abs(delta_prev-delta_next) > accuracy:
        delta_prev = delta_next
```

```
136         Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta_next)) + gamma*np.eye(
                n_train))
137         delta_next = (1/n_train)*(np.trace(Q_@K(X_train, X_train)))
138     return delta_next
139
140
141
142 def Etrain_(gamma, data, npN):
143     """
144     Estimated train error compute (expectation based, deterministic)
145     :param gamma: ridge penalty value
146     :param data: train/test data
147     :param npN: dimensionality of the data and number of Neurons
148     :return: estimated training error
149     """
150     # Unpacking variables:
151     X_train, X_test, Y_train, Y_test = data
152     n_train, n_test, p, N = npN
153
154     delta = find_delta(gamma, X_train, N, delta_accuracy)
155     Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
156     K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
157
158     #E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.matrix.trace(Q_@K_@Q_))/((1 - 1/
            N)*np.matrix.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
159     E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.trace(Q_@K_@Q_))/((1 - 1/N)*np.
            trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
160
161     return E_train_
162
163
164 def Etest_(gamma, data, npN):
165     """
166     Estimated test error compute (expectation based, deterministic)
167     :param gamma: ridge penalty value
168     :param data: train/test
169     :param npN: dimensionality of the data and number of Neurons
170     :return: estimated test error
171     """
172     # Unpacking variables:
173     X_train, X_test, Y_train, Y_test = data
174     n_train, n_test, p, N = npN
175
176     delta = 0.0
177     delta = find_delta(gamma, X_train, N, delta_accuracy)
178     Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
179     K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
180     K_xX = (N/n_train)*(K(X_train, X_test)/(1+delta))
181     K_XX = (N/n_train)*(K(X_test, X_test)/(1+delta))
182
183     #E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
            Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.
            matrix.trace(K_XX) - (1/n_test)*np.matrix.trace((np.eye(n_train) + gamma*Q_)@(
            K_xX@K_xX.T@Q_)))
184     E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
            Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.trace(
            K_XX) - (1/n_test)*np.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))
185
186     return E_test_
187
188
189 def find_optimal_gamma(gammas, E_func, data, npN):
190     errors = [E_func(g, data, npN) for g in gammas]
191     return gammas[np.argmin(errors)], errors[np.argmin(errors)]
192
193
194 def generate_synthetic_data_regression(npN, density, d=1, noise_level=0.1):
195     """
196     Generating X randomly; Y according to the law Y = Σ^T * b + eps
197     :param npN: dimensionality of the data and number of neurons
198     :param d: dimensionality of the output vector
```

```
    :param noise_level: level of noisiness in Y
    :return: training and test data for both X and Y
    """

    # Unpacking values:
    n_train, n_test, p, N = npN

    X_train = np.random.randn(p, n_train)
    X_test = np.random.randn(p, n_test)

    # Generating Y according to the law: Y = Σ^T * b + eps
    dens = density
    b = np.array([np.random.choice([0, 1], size=d, p=[1-dens/100, dens/100]) for _ in range(N)])
    W = np.random.randn(N, p)
    Y_generator = lambda X: (sigma(W@X).T@b + np.random.normal(0, noise_level, (X.shape[1], d))).flatten()

    Y_train = Y_generator(X_train)
    Y_test = Y_generator(X_test)

    # Shuffling the dataset
    shuffle_train = np.random.permutation(n_train)
    shuffle_test = np.random.permutation(n_test)

    X_train = X_train[:, shuffle_train]
    Y_train = Y_train[shuffle_train]
    X_test = X_test[:, shuffle_test]
    Y_test = Y_test[shuffle_test]

    return X_train, X_test, Y_train, Y_test, W, b


# def log_results(log_file, n, p, N, optimal_gamma_Etest, optimal_gamma_Etest_bar, E_test, E_test_):
#     with open(log_file, 'a') as f:
#         f.write(f"n = {n}, p = {p}, N = {N}\n")
#         f.write(f"Optimal gamma for Etest: {optimal_gamma_Etest}\n")
#         f.write(f"Optimal gamma for Etest_bar: {optimal_gamma_Etest_bar}\n")
#         f.write(f"Etest: {E_test}\n")
#         f.write(f"Etest_bar: {E_test_}\n\n")


comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

delta_accuracy = 1e-3
b_vector_density = 50 #%

c1 = 1.0 # p/n
c2 = 0.8 # N/n

gammas = [10**y for y in np.arange(-7, 1, 0.1)]
n_values = np.arange(100, 3000, 100)

# Split the n_values among available ranks
n_split = np.array_split(n_values, size)
local_n_values = n_split[rank]

optimal_gammas_Etest = []
optimal_gammas_Etest_bar = []

Etest_values = []
Etest_bar_values = []


for i, n in enumerate(local_n_values):
    p = int(c1 * n)  # Calculate p based on the constant ratio c1
    N = int(c2 * n)  # Calculate N based on the constant ratio c2
    npN = (int(n * 0.8), int(n * 0.2), p, N)  # Recalculate n_train and n_test
```

```
267
268     print(f'Rank␣{rank}␣-␣Iteration␣{i+1}/{len(local_n_values)}')
269     print(f'Rank␣{rank}␣-␣Trying␣for␣n␣=␣{n},␣p␣=␣{p},␣N␣=␣{N}␣...')
270
271     # Generate synthetic data
272     X_train, X_test, Y_train, Y_test, W, b = generate_synthetic_data_regression(npN,
            b_vector_density)
273     data = X_train, X_test, Y_train, Y_test
274
275     # Find the optimal gammas that minimize Etest and Etest_bar
276     optimal_gamma_Etest, E_test = find_optimal_gamma(gammas, Etest, data, npN)
277     optimal_gamma_Etest_bar, E_test_ = find_optimal_gamma(gammas, Etest_, data, npN)
278
279     print(f'Rank␣{rank}␣-␣Gamma␣that␣minimizes␣Etest:␣{optimal_gamma_Etest}')
280     print(f'Rank␣{rank}␣-␣Gamma␣that␣minimizes␣Etest_bar:␣{optimal_gamma_Etest_bar}')
281     #
282     print(f'Rank␣{rank}␣-␣Etest:␣{E_test}')
283     print(f'Rank␣{rank}␣-␣Etest_bar:␣{E_test_}')
284
285     optimal_gammas_Etest.append(optimal_gamma_Etest)
286     optimal_gammas_Etest_bar.append(optimal_gamma_Etest_bar)
287
288     Etest_values.append(E_test)
289     Etest_bar_values.append(E_test_)
290
291     # log_results(log_file, n, p, N, optimal_gamma_Etest, optimal_gamma_Etest_bar, E_test,
            E_test_)
292
293 # Gather results from all ranks
294 optimal_gammas_Etest = comm.gather(optimal_gammas_Etest, root=0)
295 optimal_gammas_Etest_bar = comm.gather(optimal_gammas_Etest_bar, root=0)
296 Etest_values = comm.gather(Etest_values, root=0)
297 Etest_bar_values = comm.gather(Etest_bar_values, root=0)
298
299 if rank == 0:
300     optimal_gammas_Etest = np.concatenate(optimal_gammas_Etest)
301     optimal_gammas_Etest_bar = np.concatenate(optimal_gammas_Etest_bar)
302     Etest_values = np.concatenate(Etest_values)
303     Etest_bar_values = np.concatenate(Etest_bar_values)
304
305     np.savez('results_parallel.npz',
306              n_values=n_values,
307              optimal_gammas_Etest=optimal_gammas_Etest,
308              optimal_gammas_Etest_bar=optimal_gammas_Etest_bar,
309              Etest_values=Etest_values,
310              Etest_bar_values=Etest_bar_values,
311              c1=c1,
312              c2=c2,
313              b_vector_density=b_vector_density)
```

A.3. Normalized ridge error against regularization parameter γ

```python
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt


def sigma(t):
    return t  # Linear activation function


def generate_synthetic_data(n, c1, c2, density=0.9, noise_level=0.1):
    p = int(c1 * n)
    N = int(c2 * n)
    n_train = int(n * 0.8)
    n_test = n - n_train

    # Generate random weights and sparse beta
    W = np.random.randn(N, p)
    b = np.random.choice([0, 1], size=(N, 1), p=[1 - density, density])

    # Generate data
    X_train = np.random.randn(p, n_train)
    Y_train = (sigma(W @ X_train).T @ b).flatten() + np.random.normal(0, noise_level, n_train
        )
    X_test = np.random.randn(p, n_test)
    Y_test = (sigma(W @ X_test).T @ b).flatten() + np.random.normal(0, noise_level, n_test)

    return X_train, X_test, Y_train, Y_test, W, b


def compute_ridge_errors(gammas, X_train, Y_train, W, b):
    n_train = X_train.shape[1]
    Sigm = sigma(W @ X_train)
    SigmT_Sigm = Sigm.T @ Sigm
    ridge_errors = []

    for gamma in gammas:
        try:
            # Compute ridge solution
            beta_ridge = (Sigm / n_train) @ linalg.solve(
                SigmT_Sigm / n_train + gamma * np.eye(n_train),
                Y_train
            )

            # Compute normalized error
            error = np.linalg.norm(beta_ridge - b, 'fro') ** 2 / np.linalg.norm(b, 'fro') **
                2
            ridge_errors.append(error)

        except np.linalg.LinAlgError:
            ridge_errors.append(np.nan)

    return np.array(ridge_errors)


# Parameters (adjust these for faster testing)
n = 512  # Reduced for local testing
c1 = 1.0  # p/n
c2 = 0.7  # N/n
b_density = 0.95
gammas = [10**y for y in np.arange(-7, 7, 0.1)]

# Generate synthetic data
X_train, X_test, Y_train, Y_test, W, b = generate_synthetic_data(
    n=n, c1=c1, c2=c2, density=b_density
)

# Compute ridge errors
ridge_errors = compute_ridge_errors(gammas, X_train, Y_train, W, b)

```

```
# Plot results
plt.figure(figsize=(10, 6))
plt.semilogx(gammas, ridge_errors, 'b-', marker='o', markersize=5)
plt.xlabel('Regularization␣Strength␣ ()', fontsize=12)
plt.ylabel('Normalized␣Ridge␣Error', fontsize=12)
plt.title(f'Ridge␣Error␣vs␣Regularization␣Strength␣(n={n},␣p={int(c1␣*␣n)},␣N={int(c2␣*␣n)})'
    )
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

A.4. Error Metrics and γ value over number of data points

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg


##############################
# 1) Basic definitions
##############################

def sigma(t):
    # For this example, just use linear activation
    return t


def K(x, y):
    # Kernel function for the linear activation
    return x.T @ y


def find_delta(gamma, X_train, N, accuracy=1e-3):
    """
    Numerically solves for the delta parameter used in the deterministic equivalents.
    """
    n_train = X_train.shape[1]
    delta_prev = 1.0
    delta_next = 0.0
    while abs(delta_prev - delta_next) > accuracy:
        delta_prev = delta_next
        Q_ = np.linalg.inv((N / n_train) * K(X_train, X_train) / (1 + delta_next) + gamma * np.eye(n_train))
        delta_next = (1 / n_train) * np.trace(Q_ @ K(X_train, X_train))
    return delta_next


##############################
# 2) Error metrics
##############################

def Etest(gamma, X_train, Y_train, X_test, Y_test, N_loops=10):
    """
    Monte Carlo (random) test MSE:
      - Randomly draw W, compute Sigm = sigma(W X_train)
      - Solve ridge, apply to test set, average over N_loops
    """
    n_train = X_train.shape[1]
    n_test = X_test.shape[1]
    p = X_train.shape[0]

    E_test_vals = []
    for _ in range(N_loops):
        W = np.random.randn(N, p)
        Sigm = sigma(W @ X_train)  # shape (N, n_train)
        Sigm_test = sigma(W @ X_test)  # shape (N, n_test)

        # Solve for ridge coefficients: (1/n_train)*Sigm * inv( Sigm^T Sigm/n_train + gamma I ) * Y_train
        A = Sigm.T @ Sigm / n_train + gamma * np.eye(n_train)
        invA_Y = linalg.solve(A, Y_train, assume_a='pos')  # shape (n_train, )
        beta = (Sigm / n_train) @ invA_Y

        # Evaluate test error
        pred_test = Sigm_test.T @ beta  # shape (n_test,)
        err = np.mean((Y_test - pred_test) ** 2)
        E_test_vals.append(err)

    return np.mean(E_test_vals)


def Etest_det(gamma, X_train, Y_train, X_test, Y_test):
```

```
    """
    Deterministic equivalent test MSE (approximation).
    """
    n_train = X_train.shape[1]
    n_test = X_test.shape[1]
    N = int(c2 * (n_train + n_test))  # or pass as a parameter
    delta = find_delta(gamma, X_train, N)

    Q_ = np.linalg.inv((N / n_train) * K(X_train, X_train) / (1 + delta) + gamma * np.eye(
        n_train))
    K_ = (N / n_train) * K(X_train, X_train) / (1 + delta)
    K_xX = (N / n_train) * K(X_train, X_test) / (1 + delta)
    K_XX = (N / n_train) * K(X_test, X_test) / (1 + delta)

    # first "direct" MSE part: (1/n_test)* ||Y_test - K_xX^T Q_ Y_train||^2
    residual = Y_test - (K_xX.T @ (Q_ @ Y_train))
    partA = np.mean(residual ** 2)

    # second "correction" part from random matrix theory
    num = (1 / N) * (Y_train.T @ Q_ @ K_ @ Q_ @ Y_train)
    den = (1 - (1 / N) * np.trace(K_ @ Q_ @ K_ @ Q_))

    partB = num / den * ((1 / n_test) * np.trace(K_XX)
                         - (1 / n_test) * np.trace((np.eye(n_train) + gamma * Q_) @ (K_xX @
                             K_xX.T @ Q_)))

    return partA + partB


def Lridge(gamma, X_train, Y_train, b):
    """
    Ridge 'coefficient error':  || beta_hat - b ||^2 / ||b||^2
    with beta_hat found by ridge. We *don't* average over W here,
    since 'b' depends on a *specific* W used to generate data.
    """
    n_train = X_train.shape[1]
    Sigm = sigma(W_gen @ X_train)  # use the same W that generated 'b'

    A = Sigm.T @ Sigm / n_train + gamma * np.eye(n_train)
    invA_Y = linalg.solve(A, Y_train, assume_a='pos')
    beta = (Sigm / n_train) @ invA_Y

    return np.linalg.norm(beta - b, 'fro') ** 2 / np.linalg.norm(b, 'fro') ** 2


##############################
# 3) Local runner
##############################

# Smaller test parameters
c1 = 1.0  # p/n
c2 = 0.7  # N/n
gammas = [10 ** x for x in np.arange(-7, 5, 0.5)]  # e.g. gamma from 1e-3 to 1e3

n_values = [500, 600, 700]  # smaller range
results_gamma_etest = []
results_gamma_etest_ = []
results_gamma_lridge = []

results_etest_vals = []
results_etest_vals_ = []
results_lridge_vals = []

for n in n_values:
    # Dimensions
    n_train = int(n * 0.8)
    n_test = n - n_train
    p = int(c1 * n)
    N = int(c2 * n)

    # Generate synthetic data for this n
```

```
    # NOTE: store W_gen, b so we can use the same underlying W in Lridge
    density = 50  # e.g. 50%
    noise_level = 0.1

    # Create a random W_gen for generating the data
    W_gen = np.random.randn(N, p)


    def Y_generator(X):
        # We create a random 'b' with certain density:
        b_local = np.random.choice([0, 1], size=(N, 1),
                                   p=[1 - density / 100, density / 100])
        # We'll keep track of b_local so we can measure Lridge
        # but for clarity, let's store it outside in a closure
        return sigma(W_gen @ X).T @ b_local, b_local


    # Make X and Y
    X_tr = np.random.randn(p, n_train)
    Y_tr_gen, b_local = Y_generator(X_tr)
    # Add noise
    Y_tr = Y_tr_gen.ravel() + noise_level * np.random.randn(n_train)

    X_te = np.random.randn(p, n_test)
    Y_te_gen, _ = Y_generator(X_te)
    Y_te = Y_te_gen.ravel() + noise_level * np.random.randn(n_test)

    # Shuffle
    perm_tr = np.random.permutation(n_train)
    X_tr = X_tr[:, perm_tr]
    Y_tr = Y_tr[perm_tr]

    perm_te = np.random.permutation(n_test)
    X_te = X_te[:, perm_te]
    Y_te = Y_te[perm_te]


    # We define a small function that can compute each metric for a given gamma
    def measure_all_metrics(g):
        # test error (random-based)
        e_test = Etest(g, X_tr, Y_tr, X_te, Y_te, N_loops=5)
        # test error (det-based)
        e_test_approx = Etest_det(g, X_tr, Y_tr, X_te, Y_te)
        # ridge error in param space
        # but we need W_gen and b_local from above
        # put them in global scope or closure
        return e_test, e_test_approx


    # Now search for optimal gamma for each metric
    best_e_test = np.inf
    best_e_test_g = None

    best_e_test_approx = np.inf
    best_e_test_approx_g = None

    best_lridge = np.inf
    best_lridge_g = None

    for g in gammas:
        # Etest
        val_test = Etest(g, X_tr, Y_tr, X_te, Y_te, N_loops=5)
        if val_test < best_e_test:
            best_e_test = val_test
            best_e_test_g = g

        # Etest_det
        val_test_det = Etest_det(g, X_tr, Y_tr, X_te, Y_te)
        if val_test_det < best_e_test_approx:
            best_e_test_approx = val_test_det
            best_e_test_approx_g = g
```

```
208
209         # Lridge
210         # We need references to W_gen, b_local in scope
211         # so let's define them as global, or do it inline:
212         Sigm_gen = sigma(W_gen @ X_tr)
213         A = Sigm_gen.T @ Sigm_gen / n_train + g * np.eye(n_train)
214         invA_Y = linalg.solve(A, Y_tr, assume_a='pos')
215         beta_est = (Sigm_gen / n_train) @ invA_Y
216         this_lridge = (np.linalg.norm(beta_est - b_local, 'fro') ** 2
217                        / np.linalg.norm(b_local, 'fro') ** 2)
218         if this_lridge < best_lridge:
219             best_lridge = this_lridge
220             best_lridge_g = g
221
222     # store results
223     results_gamma_etest.append(best_e_test_g)
224     results_etest_vals.append(best_e_test)
225
226     results_gamma_etest_.append(best_e_test_approx_g)
227     results_etest_vals_.append(best_e_test_approx)
228
229     results_gamma_lridge.append(best_lridge_g)
230     results_lridge_vals.append(best_lridge)
231
232 ##############################
233 # 4) Plot results
234 ##############################
235
236 fig, axs = plt.subplots(1, 2, figsize=(12, 5))
237
238 # (a) Plot the optimal gamma for each metric vs n
239 axs[0].plot(n_values, results_gamma_etest, 'bo-', label='Gamma for Etest')
240 axs[0].plot(n_values, results_gamma_etest_, 'mo-', label='Gamma for Etest_ approx')
241 axs[0].plot(n_values, results_gamma_lridge, 'ro-', label='Gamma for Lridge')
242 axs[0].set_xscale('log')
243 axs[0].set_yscale('log')
244 axs[0].set_xlabel('n')
245 axs[0].set_ylabel('Optimal gamma (log scale)')
246 axs[0].legend()
247 axs[0].set_title('Optimal gamma vs. n')
248
249 # (b) Plot the *minimum* error achieved vs n (for each metric)
250 axs[1].plot(n_values, results_etest_vals, 'b*-', label='Min Etest')
251 axs[1].plot(n_values, results_etest_vals_, 'm*-', label='Min Etest_ approx')
252 axs[1].plot(n_values, results_lridge_vals, 'r*-', label='Min Lridge')
253 axs[1].set_xscale('log')
254 axs[1].set_yscale('log')
255 axs[1].set_xlabel('n')
256 axs[1].set_ylabel('Error (log scale)')
257 axs[1].legend()
258 axs[1].set_title('Minimum achieved error vs. n')
259
260 plt.tight_layout()
261 plt.show()
```

A.5. Derivative-based loss functions against γ parameter (different noise levels)

```python
import numpy as np
import matplotlib.pyplot as plt

def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    return t


def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    return x.T@y


def K_(x, y, delta, N, n_train):
    """
    Function to compute kernel approximation
    :param x: first 'point' set
    :param y: second 'point' set
    :param delta: delta parameter
    :return: approximation for the kernel matrix
    """
    k_ = (N/n_train)*((K(x, y))/(1+delta))
    return k_


def find_delta(gamma, X_train, N, accuracy) -> float:
    """
    Helper-function that finds delta parameter for the resolvent Q iteratively
    :param gamma: ridge penalty value
    :param X_train: training set
    :param N: number of neurons in the hidden layer
    :param accuracy: accuracy for numerical delta finding
    :return: optimal value for delta parameter for the resolvent Q
    """
    n_train = X_train.shape[1]
    delta_prev = 1
    delta_next = 0
    while abs(delta_prev-delta_next) > accuracy:
        delta_prev = delta_next
        Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta_next)) + gamma*np.eye(
            n_train))
        delta_next = (1/n_train)*(np.trace(Q_@K(X_train, X_train)))
    return delta_next


def Etrain_(gamma, data, npN):
    """
    Estimated train error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated training error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = find_delta(gamma, X_train, N, delta_accuracy)
```

```
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))

    #E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.matrix.trace(Q_@K_@Q_))/((1 - 1/
        N)*np.matrix.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
    E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.trace(Q_@K_@Q_))/((1 - 1/N)*np.
        trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)

    return E_train_


def Etest_(gamma, data, npN):
    """
    Estimated test error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated test error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = 0.0
    delta = find_delta(gamma, X_train, N, delta_accuracy)
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
    K_xX = (N/n_train)*(K(X_train, X_test)/(1+delta))
    K_XX = (N/n_train)*(K(X_test, X_test)/(1+delta))

    #E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
        Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.
        matrix.trace(K_XX) - (1/n_test)*np.matrix.trace((np.eye(n_train) + gamma*Q_)@(
        K_xX@K_xX.T@Q_)))
    E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
        Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.trace(
        K_XX) - (1/n_test)*np.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))

    return E_test_


def estimate_variance_RSS(Y, Sigm, n, gamma):
    QyT = np.linalg.solve(Sigm.T @ Sigm / n + gamma * np.eye(n), Y.T)
    beta_hat = Sigm / n @ QyT

    y_est = beta_hat.T @ Sigm

    # variance already squared
    variance = ((Y - y_est) @ (Y - y_est).T)/n  # sigm^2
    return variance


def dL_dy_expected(gamma, X, Y, W, variance):
    """
    "Oracle". Conditional expectation w.r.t. epsilon, taken
    from an unfolded derivative of Loss function ||beta-b||**2
    by gamma (set equal to 0). Some 'oracle' prediction for loss.
    :param gamma: gamma parameter (1x1)
    :param X: input data (pxn)
    :param Y: output data (dxn)
    :param W: weights matrix (Nxp)
    :param eps: noise vector (dxn)
    :return: calculated value of the derivative
    """
    Sigm = sigma(W @ X) # Size Nxn
    n = X.shape[1]

    Q = (1/n * (Sigm.T @ Sigm) + gamma * np.eye(n))
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv
```

```
131
132     dL_dy_exp = gamma*(Y @ Q_inv3 @ Y.T) - variance * np.trace(Q_inv2)
133     return dL_dy_exp
134
135
136 def dL_dy_estimated(gamma, X, Y, W, variance=1):
137     """
138     Calculation of the unfolded derivative of Loss function ||beta-b||**2
139     under expectation w.r.t. epsilon, with estimated variance instead of real 'oracle' one.
140     :param gamma: gamma parameter (1x1)
141     :param X: input data (pxn)
142     :param Y: output data (dxn)
143     :param W: weights matrix (Nxp)
144     :param eps: noise vector (dxn)
145     :return: calculated value of the derivative
146     """
147     Sigm = sigma(W @ X)  # Size Nxn
148     n = X.shape[1]
149
150     Q = (1 / n * (Sigm.T @ Sigm) + gamma * np.eye(n))
151     Q_inv = np.linalg.inv(Q)
152     Q_inv2 = Q_inv @ Q_inv
153     Q_inv3 = Q_inv2 @ Q_inv
154
155     estimated_var = estimate_variance_RSS(Y, Sigm, n, gamma)
156
157     dL_dy_exp = gamma * (Y @ Q_inv3 @ Y.T) - estimated_var * np.trace(Q_inv2)
158     return dL_dy_exp
159
160
161 def generate_synthetic_data_regression(npN, density, d=1, noise_level=0.1):
162     """
163     Generates data for the model:
164         Y = b^T * Sigma + eps
165     with Sigma = W @ X and b has 'density'% of 1's.
166
167     Shapes:
168       X_train: (p, n_train)
169       X_test:  (p, n_test)
170       W:       (N, p)
171       b:       (N, d)    -- if d=1 => shape (N,1)
172       Y_train, e_train: (d, n_train)
173       Y_test,  e_test:  (d, n_test)
174
175     :param npN: (n_train, n_test, p, N)
176     :param density: percentage for b's 1's
177     :param d: dimension of output (1 => scalar outputs)
178     :param noise_level: std-dev of Gaussian noise
179     :return: (X_train, X_test, Y_train, Y_test, W, b, e_train, e_test)
180     """
181     n_train, n_test, p, N = npN
182
183     # 1) Generate X
184     X_train = np.random.randn(p, n_train)  # (p, n_train)
185     X_test = np.random.randn(p, n_test)  # (p, n_test)
186
187     # 2) Generate b (N x d), if d=1 => (N,1)
188     b = np.random.choice([0, 1],
189                          size=(N, d),
190                          p=[1 - density / 100, density / 100])
191
192     # 3) Generate W (N x p)
193     W = np.random.randn(N, p)
194
195     def make_data(X):
196         """
197         Given X with shape (p, n), returns:
198           Y:  (d, n)
199           e:  (d, n)
200           Sigm: (N, n)  [might be useful if needed]
201         """
```

```
        # Sigma = W @ X => (N, n)
        Sigm = sigma(W @ X)

        # Noise eps => (d, n)
        e = np.random.normal(loc=0, scale=noise_level, size=(d, X.shape[1]))

        # b^T shape => (d, N), so b^T @ Sigm => (d, n)
        Y = (b.T @ Sigm) + e

        return Y, e, Sigm

    # 4) Make training data (Y_train_full, e_train_full)
    Y_train_full, e_train_full, Sigm_train = make_data(X_train)

    # 5) Make test data (Y_test_full, e_test_full)
    Y_test_full, e_test_full, Sigm_test = make_data(X_test)

    # 6) Shuffle columns (the "sample" axis) in X, Y, e
    #    X is (p, n), Y,e are (d, n). We shuffle axis=1 for each.
    idx_train = np.random.permutation(n_train)
    idx_test = np.random.permutation(n_test)

    X_train = X_train[:, idx_train]
    Y_train_full = Y_train_full[:, idx_train]
    e_train_full = e_train_full[:, idx_train]

    X_test = X_test[:, idx_test]
    Y_test_full = Y_test_full[:, idx_test]
    e_test_full = e_test_full[:, idx_test]

    # 7) Return them all in consistent shapes
    # No flattening is required; Y and e remain (d,n).
    return X_train, X_test, Y_train_full, Y_test_full, W, b, e_train_full, e_test_full


def L_ridge(gamma, X, Y, W, b):
    n = X.shape[1]
    Sigm = sigma(W @ X)

    QyT = np.linalg.solve(Sigm.T@Sigm/n + gamma*np.eye(n), Y.T)
    beta = Sigm/n @ QyT

    numerator = np.linalg.norm(beta - b,'fro')**2
    denominator = np.linalg.norm(b, 'fro')**2

    return numerator/denominator


def plot_estimated_vs_oracle_different_xranges():
    """
    Left subplot:
      dL/dgamma (estimated RSS) over gamma in [1e-5, 1e3] (step 0.005 in log10).
    Right subplot:
      dL/dgamma (oracle) over gamma in [1e-3, 1e2] (step 0.005 in log10).
    We loop over noise_levels=[1,2,3,4,5].
    """

    # Basic scenario
    noise_levels = [1,2,3,4,5]

    # c1, c2 => p/n=1.0, N/n=0.8
    n = 200
    c1 = 1.0
    c2 = 0.8

    p = int(c1*n)
    N = int(c2*n)
    n_train = int(0.8*n)
    n_test = n - n_train
    npN = (n_train, n_test, p, N)

```

```
    # Different gamma ranges
    gammas_est = [10**y for y in np.arange(-4, 3, 0.005)]   # ~1e-5 to ~1e3
    gammas_oracle = [10**y for y in np.arange(-4, 2, 0.005)]# ~1e-3 to ~1e2

    fig, (ax_est, ax_oracle) = plt.subplots(1, 2, figsize=(12,5))

    # Loop over noise
    for nl in noise_levels:
        # Generate data for each noise
        X_train, X_test, Y_train, Y_test, W, b, e_train, e_test = \
            generate_synthetic_data_regression(npN, density=50, d=1, noise_level=nl)

        # Evaluate dL/dgamma (est) over gammas_est
        dL_est_vals = []
        for g in gammas_est:
            val_est = dL_dy_estimated(g, X_train, Y_train, W, variance=1.0)
            dL_est_vals.append(val_est.item() if hasattr(val_est, 'item') else float(val_est)
                )

        # Evaluate dL/dgamma (oracle) over gammas_oracle
        dL_oracle_vals = []
        # We pass in the real variance = nl^2
        for g in gammas_oracle:
            val_oracle = dL_dy_expected(g, X_train, Y_train, W, variance=nl**2)
            dL_oracle_vals.append(val_oracle.item() if hasattr(val_oracle, 'item') else float
                (val_oracle))

        # Plot them
        ax_est.loglog(gammas_est, dL_est_vals, label=f"noise={nl}")
        ax_oracle.loglog(gammas_oracle, dL_oracle_vals, label=f"noise={nl}")

    # Label/Legend subplots
    ax_est.set_title("Estimated Derivative (RSS) - Gamma in [1e-5,1e3]")
    ax_est.set_xlabel("Gamma (log scale)")
    ax_est.set_ylabel("dL/dGamma (EstVar)")
    ax_est.grid(True, which='both', ls='--', alpha=0.7)
    ax_est.legend()

    ax_oracle.set_title("Oracle Derivative - Gamma in [1e-3,1e2]")
    ax_oracle.set_xlabel("Gamma (log scale)")
    ax_oracle.set_ylabel("dL/dGamma (Oracle)")
    ax_oracle.grid(True, which='both', ls='--', alpha=0.7)
    ax_oracle.legend()

    fig.tight_layout()
    plt.show()


if __name__ == "__main__":
    plot_estimated_vs_oracle_different_xranges()
```

A.6. Direct loss with γ derived from minimizing different loss functions (including the derivative-based)

```
import numpy as np
import matplotlib.pyplot as plt

def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    return t


def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    return x.T@y


def K_(x, y, delta, N, n_train):
    """
    Function to compute kernel approximation
    :param x: first 'point' set
    :param y: second 'point' set
    :param delta: delta parameter
    :return: approximation for the kernel matrix
    """
    k_ = (N/n_train)*((K(x, y))/(1+delta))
    return k_


def find_delta(gamma, X_train, N, accuracy) -> float:
    """
    Helper-function that finds delta parameter for the resolvent Q iteratively
    :param gamma: ridge penalty value
    :param X_train: training set
    :param N: number of neurons in the hidden layer
    :param accuracy: accuracy for numerical delta finding
    :return: optimal value for delta parameter for the resolvent Q
    """
    n_train = X_train.shape[1]
    delta_prev = 1
    delta_next = 0
    while abs(delta_prev-delta_next) > accuracy:
        delta_prev = delta_next
        Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta_next)) + gamma*np.eye(
            n_train))
        delta_next = (1/n_train)*(np.trace(Q_@K(X_train, X_train)))
    return delta_next


def Etrain_(gamma, data, npN):
    """
    Estimated train error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated training error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = find_delta(gamma, X_train, N, delta_accuracy)
```

```
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))

    #E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.matrix.trace(Q_@K_@Q_))/((1 - 1/
        N)*np.matrix.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
    E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.trace(Q_@K_@Q_))/((1 - 1/N)*np.
        trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)

    return E_train_


def Etest_(gamma, data, npN):
    """
    Estimated test error compute (expectation based, deterministic)
    :param gamma: ridge penalty value
    :param data: train/test
    :param npN: dimensionality of the data and number of Neurons
    :return: estimated test error
    """
    # Unpacking variables:
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    delta = 0.0
    delta = find_delta(gamma, X_train, N, delta_accuracy)
    Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
    K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
    K_xX = (N/n_train)*(K(X_train, X_test)/(1+delta))
    K_XX = (N/n_train)*(K(X_test, X_test)/(1+delta))

    #E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
        Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.
        matrix.trace(K_XX) - (1/n_test)*np.matrix.trace((np.eye(n_train) + gamma*Q_)@(
        K_xX@K_xX.T@Q_)))
    E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
        Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.trace(
        K_XX) - (1/n_test)*np.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))

    return E_test_


def estimate_variance_RSS(Y, Sigm, n, gamma):
    QyT = np.linalg.solve(Sigm.T @ Sigm / n + gamma * np.eye(n), Y.T)
    beta_hat = Sigm / n @ QyT

    y_est = beta_hat.T @ Sigm

    # variance already squared
    variance = ((Y - y_est) @ (Y - y_est).T)/n  # sigm^2
    return variance


def dL_dy_expected(gamma, X, Y, W, variance):
    """
    "Oracle". Conditional expectation w.r.t. epsilon, taken
    from an unfolded derivative of Loss function ||beta-b||**2
    by gamma (set equal to 0). Some 'oracle' prediction for loss.
    :param gamma: gamma parameter (1x1)
    :param X: input data (pxn)
    :param Y: output data (dxn)
    :param W: weights matrix (Nxp)
    :param eps: noise vector (dxn)
    :return: calculated value of the derivative
    """
    Sigm = sigma(W @ X) # Size Nxn
    n = X.shape[1]

    Q = (1/n * (Sigm.T @ Sigm) + gamma * np.eye(n))
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv
```

```
131
132     dL_dy_exp = gamma*(Y @ Q_inv3 @ Y.T) - variance * np.trace(Q_inv2)
133     return dL_dy_exp
134
135
136 def dL_dy_estimated(gamma, X, Y, W, variance=1):
137     """
138     Calculation of the unfolded derivative of Loss function ||beta-b||**2
139     under expectation w.r.t. epsilon, with estimated variance instead of real 'oracle' one.
140     :param gamma: gamma parameter (1x1)
141     :param X: input data (pxn)
142     :param Y: output data (dxn)
143     :param W: weights matrix (Nxp)
144     :param eps: noise vector (dxn)
145     :return: calculated value of the derivative
146     """
147     Sigm = sigma(W @ X)  # Size Nxn
148     n = X.shape[1]
149
150     Q = (1 / n * (Sigm.T @ Sigm) + gamma * np.eye(n))
151     Q_inv = np.linalg.inv(Q)
152     Q_inv2 = Q_inv @ Q_inv
153     Q_inv3 = Q_inv2 @ Q_inv
154
155     estimated_var = estimate_variance_RSS(Y, Sigm, n, gamma)
156
157     dL_dy_exp = gamma * (Y @ Q_inv3 @ Y.T) - estimated_var * np.trace(Q_inv2)
158     return dL_dy_exp
159
160
161 def generate_synthetic_data_regression(npN, density, d=1, noise_level=0.1):
162     """
163     Generates data for the model:
164         Y = b^T * Sigma + eps
165     with Sigma = W @ X and b has 'density'% of 1's.
166
167     Shapes:
168       X_train: (p, n_train)
169       X_test:  (p, n_test)
170       W:       (N, p)
171       b:       (N, d)    -- if d=1 => shape (N,1)
172       Y_train, e_train: (d, n_train)
173       Y_test,  e_test:  (d, n_test)
174
175     :param npN: (n_train, n_test, p, N)
176     :param density: percentage for b's 1's
177     :param d: dimension of output (1 => scalar outputs)
178     :param noise_level: std-dev of Gaussian noise
179     :return: (X_train, X_test, Y_train, Y_test, W, b, e_train, e_test)
180     """
181     n_train, n_test, p, N = npN
182
183     # 1) Generate X
184     X_train = np.random.randn(p, n_train)  # (p, n_train)
185     X_test = np.random.randn(p, n_test)  # (p, n_test)
186
187     # 2) Generate b (N x d), if d=1 => (N,1)
188     b = np.random.choice([0, 1],
189                          size=(N, d),
190                          p=[1 - density / 100, density / 100])
191
192     # 3) Generate W (N x p)
193     W = np.random.randn(N, p)
194
195     def make_data(X):
196         """
197         Given X with shape (p, n), returns:
198           Y:  (d, n)
199           e:  (d, n)
200           Sigm: (N, n)  [might be useful if needed]
201         """
```

```
        # Sigma = W @ X => (N, n)
        Sigm = sigma(W @ X)

        # Noise eps => (d, n)
        e = np.random.normal(loc=0, scale=noise_level, size=(d, X.shape[1]))

        # b^T shape => (d, N), so b^T @ Sigm => (d, n)
        Y = (b.T @ Sigm) + e

        return Y, e, Sigm

    # 4) Make training data (Y_train_full, e_train_full)
    Y_train_full, e_train_full, Sigm_train = make_data(X_train)

    # 5) Make test data (Y_test_full, e_test_full)
    Y_test_full, e_test_full, Sigm_test = make_data(X_test)

    # 6) Shuffle columns (the "sample" axis) in X, Y, e
    #    X is (p, n), Y,e are (d, n). We shuffle axis=1 for each.
    idx_train = np.random.permutation(n_train)
    idx_test = np.random.permutation(n_test)

    X_train = X_train[:, idx_train]
    Y_train_full = Y_train_full[:, idx_train]
    e_train_full = e_train_full[:, idx_train]

    X_test = X_test[:, idx_test]
    Y_test_full = Y_test_full[:, idx_test]
    e_test_full = e_test_full[:, idx_test]

    # 7) Return them all in consistent shapes
    # No flattening is required; Y and e remain (d,n).
    return X_train, X_test, Y_train_full, Y_test_full, W, b, e_train_full, e_test_full


def L_ridge(gamma, X, Y, W, b):
    n = X.shape[1]
    Sigm = sigma(W @ X)

    QyT = np.linalg.solve(Sigm.T@Sigm/n + gamma*np.eye(n), Y.T)
    beta = Sigm/n @ QyT

    numerator = np.linalg.norm(beta - b,'fro')**2
    denominator = np.linalg.norm(b, 'fro')**2

    return numerator/denominator


def plot_estimated_vs_oracle_different_xranges():
    """
    Left subplot:
      dL/dgamma (estimated RSS) over gamma in [1e-5, 1e3] (step 0.005 in log10).
    Right subplot:
      dL/dgamma (oracle) over gamma in [1e-3, 1e2] (step 0.005 in log10).
    We loop over noise_levels=[1,2,3,4,5].
    """

    # Basic scenario
    noise_levels = [1,2,3,4,5]

    # c1, c2 => p/n=1.0, N/n=0.8
    n = 200
    c1 = 1.0
    c2 = 0.8

    p = int(c1*n)
    N = int(c2*n)
    n_train = int(0.8*n)
    n_test = n - n_train
    npN = (n_train, n_test, p, N)

```

```
    # Different gamma ranges
    gammas_est = [10**y for y in np.arange(-4, 3, 0.005)]   # ~1e-5 to ~1e3
    gammas_oracle = [10**y for y in np.arange(-4, 2, 0.005)]# ~1e-3 to ~1e2

    fig, (ax_est, ax_oracle) = plt.subplots(1, 2, figsize=(12,5))

    # Loop over noise
    for nl in noise_levels:
        # Generate data for each noise
        X_train, X_test, Y_train, Y_test, W, b, e_train, e_test = \
            generate_synthetic_data_regression(npN, density=50, d=1, noise_level=nl)

        # Evaluate dL/dgamma (est) over gammas_est
        dL_est_vals = []
        for g in gammas_est:
            val_est = dL_dy_estimated(g, X_train, Y_train, W, variance=1.0)
            dL_est_vals.append(val_est.item() if hasattr(val_est, 'item') else float(val_est)
                )

        # Evaluate dL/dgamma (oracle) over gammas_oracle
        dL_oracle_vals = []
        # We pass in the real variance = nl^2
        for g in gammas_oracle:
            val_oracle = dL_dy_expected(g, X_train, Y_train, W, variance=nl**2)
            dL_oracle_vals.append(val_oracle.item() if hasattr(val_oracle, 'item') else float
                (val_oracle))

        # Plot them
        ax_est.loglog(gammas_est, dL_est_vals, label=f"noise={nl}")
        ax_oracle.loglog(gammas_oracle, dL_oracle_vals, label=f"noise={nl}")

    # Label/Legend subplots
    ax_est.set_title("Estimated Derivative (RSS) - Gamma in [1e-5,1e3]")
    ax_est.set_xlabel("Gamma (log scale)")
    ax_est.set_ylabel("dL/dGamma (EstVar)")
    ax_est.grid(True, which='both', ls='--', alpha=0.7)
    ax_est.legend()

    ax_oracle.set_title("Oracle Derivative - Gamma in [1e-3,1e2]")
    ax_oracle.set_xlabel("Gamma (log scale)")
    ax_oracle.set_ylabel("dL/dGamma (Oracle)")
    ax_oracle.grid(True, which='both', ls='--', alpha=0.7)
    ax_oracle.legend()

    fig.tight_layout()
    plt.show()


if __name__ == "__main__":
    plot_estimated_vs_oracle_different_xranges()
```

A.7. Convergence Study

```python
import numpy as np
from scipy import linalg

def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    return t


def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    return x.T@y


def K_(x, y, delta, N, n_train):
    """
    Function to compute kernel approximation
    :param x: first 'point' set
    :param y: second 'point' set
    :param delta: delta parameter
    :return: approximation for the kernel matrix
    """
    k_ = (N/n_train)*((K(x, y))/(1+delta))
    return k_


def Etrain(gamma, data, npN, monte_carlo_loops=30):
    """
    Actual training error compute (random-based)
    :param gamma: ridge penalty value
    :param data: train/test data
    :param npN: dimensionality of the data and num of neurons
    :param monte_carlo_loops: number of iterations for E value averaging for different
        generated W
    :return: Training error
    """
    # Unpacking variables
    X_train, X_test, Y_train, Y_test = data
    n_train, n_test, p, N = npN

    E_train_arr = []
    p = X_train.shape[0]

    for i in range(monte_carlo_loops):

        # W = np.random.randn(N, p)
        # Sigm = sigma(W@X_train)
        # Q_y = np.linalg.inv((1/n_train)*Sigm.T@Sigm + gamma*np.eye(n_train))
        #
        # E_train = (gamma*gamma/n_train)*Y_train@np.linalg.matrix_power(Q_y, 2)@Y_train.T
        # E_train_arr.append(E_train)

        W = np.random.randn(N, p)
        Sigm = sigma(W @ X_train)
        Sigm_ = sigma(W @ X_test)

        inv_tQ_r = linalg.solve(Sigm.T @ Sigm / n_train + gamma * np.eye(n_train), Y_train)
        beta = Sigm / n_train @ inv_tQ_r

        E_train = np.linalg.norm(Y_train-Sigm.T@beta)**2/n_train
        E_train_arr.append(E_train)
```

```
69
70      return np.mean(np.array(E_train_arr))
71
72
73  def Etest(gamma, data, npN, monte_carlo_loops=20):
74      """
75      Actual test error compute (random-based)
76      :param gamma: ridge penalty value
77      :param data: train/test data
78      :param npN: dimensionality of the data and number of neurons
79      :param monte_carlo_loops: number of iterations for E value averaging for different
            generated W
80      :return: Test error
81      """
82      # Unpacking variables
83      X_train, X_test, Y_train, Y_test = data
84      n_train, n_test, p, N = npN
85
86      E_test_arr = []
87      p = X_train.shape[0]
88
89      for i in range(monte_carlo_loops):
90
91          W = np.random.randn(N, p)
92          Sigm = sigma(W@X_train)
93          Sigm_ = sigma(W@X_test)
94
95          inv_tQ_r = linalg.solve(Sigm.T@Sigm/n_train + gamma * np.eye(n_train), Y_train)
96          beta = Sigm/n_train @ inv_tQ_r
97
98          # Q_y = np.linalg.inv((1 / n_train) * Sigm.T @ Sigm + gamma * np.eye(n_train))
99
100         # term1 = (1/n_test)*(Y_test@Y_test.T)
101         # term2 = (2/(n_train*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Y_test.T)
102         # term3 = (1/(n_train**2*n_test))*(Y_train@Q_y@Sigm.T@Sigm_@Sigm_.T@Sigm@Q_y@Y_train.
                T)
103
104
105         E_test = np.linalg.norm(Y_test-Sigm_.T@beta)**2/n_test
106         E_test_arr.append(E_test)
107
108     ans = np.mean(np.array(E_test_arr))
109
110     return ans
111
112
113 def find_delta(gamma, X_train, N, accuracy) -> float:
114     """
115     Helper-function that finds delta parameter for the resolvent Q iteratively
116     :param gamma: ridge penalty value
117     :param X_train: training set
118     :param N: number of neurons in the hidden layer
119     :param accuracy: accuracy for numerical delta finding
120     :return: optimal value for delta parameter for the resolvent Q
121     """
122     n_train = X_train.shape[1]
123     delta_prev = 1
124     delta_next = 0
125     while abs(delta_prev-delta_next) > accuracy:
126         delta_prev = delta_next
127         Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta_next)) + gamma*np.eye(
                n_train))
128         delta_next = (1/n_train)*(np.trace(Q_@K(X_train, X_train)))
129     return delta_next
130
131
132
133 def Etrain_(gamma, data, npN):
134     """
135     Estimated train error compute (expectation based, deterministic)
136     :param gamma: ridge penalty value
```

```
137     :param data: train/test data
138     :param npN: dimensionality of the data and number of Neurons
139     :return: estimated training error
140     """
141     # Unpacking variables:
142     X_train, X_test, Y_train, Y_test = data
143     n_train, n_test, p, N = npN
144 
145     delta = find_delta(gamma, X_train, N, delta_accuracy)
146     Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
147     K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
148 
149     #E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.matrix.trace(Q_@K_@Q_))/((1 - 1/
            N)*np.matrix.trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
150     E_train_ = ((gamma**2)/n_train)*(Y_train@Q_@((((1/N)*np.trace(Q_@K_@Q_))/((1 - 1/N)*np.
            trace(K_@Q_@K_@Q_)))*K_ + np.eye(n_train))@Q_@Y_train.T)
151 
152     return E_train_
153 
154 
155 def Etest_(gamma, data, npN):
156     """
157     Estimated test error compute (expectation based, deterministic)
158     :param gamma: ridge penalty value
159     :param data: train/test
160     :param npN: dimensionality of the data and number of Neurons
161     :return: estimated test error
162     """
163     # Unpacking variables:
164     X_train, X_test, Y_train, Y_test = data
165     n_train, n_test, p, N = npN
166 
167     delta = 0.0
168     delta = find_delta(gamma, X_train, N, delta_accuracy)
169     Q_ = np.linalg.inv((N/n_train)*(K(X_train, X_train)/(1+delta)) + gamma*np.eye(n_train))
170     K_ = (N/n_train)*(K(X_train, X_train)/(1+delta))
171     K_xX = (N/n_train)*(K(X_train, X_test)/(1+delta))
172     K_XX = (N/n_train)*(K(X_test, X_test)/(1+delta))
173 
174     #E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
            Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.matrix.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.
            matrix.trace(K_XX) - (1/n_test)*np.matrix.trace((np.eye(n_train) + gamma*Q_)@(
            K_xX@K_xX.T@Q_)))
175     E_test_ = (1/n_test)*np.sum((Y_test.T - K_xX.T@Q_@Y_train.T)**2) + (((1/N)*(
            Y_train@Q_@K_@Q_@Y_train.T))/((1-1/N)*np.trace(K_@Q_@K_@Q_)))*((1/n_test)*np.trace(
            K_XX) - (1/n_test)*np.trace((np.eye(n_train) + gamma*Q_)@(K_xX@K_xX.T@Q_)))
176 
177     return E_test_
178 
179 
180 def estimate_variance_RSS(Y, Sigm, n, gamma):
181     QyT = np.linalg.solve(Sigm.T @ Sigm / n + gamma * np.eye(n), Y.T)
182     beta_hat = Sigm / n @ QyT
183 
184     y_est = beta_hat.T @ Sigm
185 
186     # variance already squared
187     variance = ((Y - y_est) @ (Y - y_est).T)/n  # sigm^2
188     return variance
189 
190 
191 def dL_dy_real(gamma, X, Y, W, eps):
192     """
193     Real unfolded derivative of Loss function ||beta-b||**2 by gamma (set equal to 0)
194     :param gamma: gamma parameter (1x1)
195     :param X: input data (pxn)
196     :param Y: output data (dxn)
197     :param W: weights matrix (Nxp)
198     :param eps: noise vector (dxn)
199     :return: calculated value of the derivative
200     """
```

```
    Sigm = sigma(W @ X) # Size Nxn
    n = X.shape[1]

    Q = (1/n) * (Sigm.T @ Sigm) + gamma * np.eye(n)
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv

    dL_dy = gamma*(Y @ Q_inv3 @ Y.T) - (Y @ Q_inv2 @ eps.T)
    return dL_dy


def dL_dy_expected(gamma, X, Y, W, variance):
    """
    "Oracle". Conditional expectation w.r.t. epsilon, taken
    from an unfolded derivative of Loss function ||beta-b||**2
    by gamma (set equal to 0). Some 'oracle' prediction for loss.
    :param gamma: gamma parameter (1x1)
    :param X: input data (pxn)
    :param Y: output data (dxn)
    :param W: weights matrix (Nxp)
    :param eps: noise vector (dxn)
    :return: calculated value of the derivative
    """
    Sigm = sigma(W @ X) # Size Nxn
    n = X.shape[1]

    Q = (1/n * (Sigm.T @ Sigm) + gamma * np.eye(n))
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv

    dL_dy_exp = gamma*(Y @ Q_inv3 @ Y.T) - variance * np.trace(Q_inv2)
    return dL_dy_exp


def dL_dy_estimated(gamma, X, Y, W, variance=1):
    """
    Calculation of the unfolded derivative of Loss function ||beta-b||**2
    under expectation w.r.t. epsilon, with estimated variance instead of real 'oracle' one.
    :param gamma: gamma parameter (1x1)
    :param X: input data (pxn)
    :param Y: output data (dxn)
    :param W: weights matrix (Nxp)
    :param eps: noise vector (dxn)
    :return: calculated value of the derivative
    """
    Sigm = sigma(W @ X)  # Size Nxn
    n = X.shape[1]

    Q = (1 / n * (Sigm.T @ Sigm) + gamma * np.eye(n))
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv

    estimated_var = estimate_variance_RSS(Y, Sigm, n, gamma)

    dL_dy_exp = gamma * (Y @ Q_inv3 @ Y.T) - estimated_var * np.trace(Q_inv2)
    return dL_dy_exp


def dL_dy_estimated_fixed_rss(gamma, X, Y, W, variance, gamma_rss):
    """
    "Estimated" derivative: dL/dgamma, but the RSS-based variance is computed
    at 'gamma_rss'. Then the derivative uses the current 'gamma' for the Q
    in the standard formula:
        derivative = gamma*(Y Q_inv^3 Y^T) - est_var * trace(Q_inv^2)
    where est_var is from 'estimate_variance_RSS(Y, Sigm, n, gamma_rss)'.
    """
    n = X.shape[1]
    Sigm = W @ X  # shape (N, n)
```

```
272
273     # Build Q for *this gamma*:
274     Q = (1.0/n) * (Sigm.T @ Sigm) + gamma*np.eye(n)
275     Q_inv = np.linalg.inv(Q)
276     Q_inv2 = Q_inv @ Q_inv
277     Q_inv3 = Q_inv2 @ Q_inv
278
279     # We do the RSS-based variance at 'gamma_rss'
280     # Must call 'estimate_variance_RSS':
281     est_var = estimate_variance_RSS(Y, Sigm, n, gamma_rss)
282
283     # derivative:
284     term1 = gamma * (Y @ Q_inv3 @ Y.T)   # shape (1,1)
285     term2 = est_var * np.trace(Q_inv2)
286     dL = term1 - term2
287
288     # Convert to float
289     if hasattr(dL, "item"):
290         return float(dL.item())
291     return float(dL)
292
293
294 def generate_synthetic_data_regression(npN, density, d=1, noise_level=0.1):
295     """
296     Generates data for the model:
297         Y = b^T * Sigma + eps
298     with Sigma = W @ X and b has 'density'% of 1's.
299
300     Shapes:
301       X_train: (p, n_train)
302       X_test:  (p, n_test)
303       W:       (N, p)
304       b:       (N, d)    -- if d=1 => shape (N,1)
305       Y_train, e_train: (d, n_train)
306       Y_test,  e_test:  (d, n_test)
307
308     :param npN: (n_train, n_test, p, N)
309     :param density: percentage for b's 1's
310     :param d: dimension of output (1 => scalar outputs)
311     :param noise_level: std-dev of Gaussian noise
312     :return: (X_train, X_test, Y_train, Y_test, W, b, e_train, e_test)
313     """
314     n_train, n_test, p, N = npN
315
316     # 1) Generate X
317     X_train = np.random.randn(p, n_train)  # (p, n_train)
318     X_test = np.random.randn(p, n_test)  # (p, n_test)
319
320     # 2) Generate b (N x d), if d=1 => (N,1)
321     b = np.random.choice([0, 1],
322                          size=(N, d),
323                          p=[1 - density / 100, density / 100])
324
325     # 3) Generate W (N x p)
326     W = np.random.randn(N, p)
327
328     def make_data(X):
329         """
330         Given X with shape (p, n), returns:
331           Y:  (d, n)
332           e:  (d, n)
333           Sigm: (N, n)  [might be useful if needed]
334         """
335         # Sigma = W @ X => (N, n)
336         Sigm = sigma(W @ X)
337
338         # Noise eps => (d, n)
339         e = np.random.normal(loc=0, scale=noise_level, size=(d, X.shape[1]))
340
341         # b^T shape => (d, N), so b^T @ Sigm => (d, n)
342         Y = (b.T @ Sigm) + e
```

```
        return Y, e, Sigm

    # 4) Make training data (Y_train_full, e_train_full)
    Y_train_full, e_train_full, Sigm_train = make_data(X_train)

    # 5) Make test data (Y_test_full, e_test_full)
    Y_test_full, e_test_full, Sigm_test = make_data(X_test)

    # 6) Shuffle columns (the "sample" axis) in X, Y, e
    #    X is (p, n), Y,e are (d, n). We shuffle axis=1 for each.
    idx_train = np.random.permutation(n_train)
    idx_test = np.random.permutation(n_test)

    X_train = X_train[:, idx_train]
    Y_train_full = Y_train_full[:, idx_train]
    e_train_full = e_train_full[:, idx_train]

    X_test = X_test[:, idx_test]
    Y_test_full = Y_test_full[:, idx_test]
    e_test_full = e_test_full[:, idx_test]

    # 7) Return them all in consistent shapes
    # No flattening is required; Y and e remain (d,n).
    return X_train, X_test, Y_train_full, Y_test_full, W, b, e_train_full, e_test_full


def find_optimal_gamma_grid(gammas, f, data, npN):
    """
    Minimizes f(gamma, data, npN) by scanning over 'gammas'.
    Returns (best_gamma, best_value).
    """
    vals = [f(g, data, npN) for g in gammas]
    i_min = np.argmin(vals)
    return float(gammas[i_min]), float(vals[i_min])


def find_zero_derivative_grid(gammas, deriv_func):
    """
    Finds gamma that yields derivative closest to zero in absolute value,
    scanning 'gammas'. Returns (best_gamma, best_deriv_value).
    The caller is expected to provide a function deriv_func(g).
    """
    vals = [deriv_func(g) for g in gammas]
    abs_vals = np.abs(vals)
    i_min = np.argmin(abs_vals)
    return float(gammas[i_min]), float(vals[i_min])


def L_ridge(gamma, X, Y, W, b):
    n = X.shape[1]
    Sigm = sigma(W @ X)

    QyT = np.linalg.solve(Sigm.T@Sigm/n + gamma*np.eye(n), Y.T)
    beta = Sigm/n @ QyT

    numerator = np.linalg.norm(beta - b,'fro')**2
    denominator = np.linalg.norm(b, 'fro')**2

    return numerator/denominator

def find_optimal_gamma_grid(gammas, f, data, npN):
    """
    Minimizes f(gamma, data, npN) by scanning over 'gammas'.
    Returns (best_gamma, best_value).
    best_gamma is a float, best_value is float.
    """
    vals = [f(g, data, npN) for g in gammas]
    i_min = np.argmin(vals)
    return float(gammas[i_min]), float(vals[i_min])

```

```
414
415 if __name__ == "__main__":
416
417     # ---------- fixed experiment parameters ----------
418     delta_accuracy      = 1e-3
419     b_vector_density    = 50          # (% of ones in true b)
420     noise_level         = 8           #   of additive noise
421     real_variance       = noise_level**2
422     c1, c2              = 1.0, 0.8    # p/n and N/n
423     n_values            = [100, 200, 300]
424     num_repeats         = 10          # averages per n
425     gammas_grid         = [10**x for x in np.arange(-6, 3, 0.01)]
426     # -------------------------------------------------
427
428     # containers
429     n_arr               = []
430     gEtrain_arr, gEtest_arr  = [], []
431     gOracle_arr, gEstVar_arr = [], []
432
433     lrEtrain_arr,  lrEtest_arr  = [], []
434     lrOracle_arr,  lrEstVar_arr = [], []
435
436     # ---------------- main loop on sample size ----------------
437     for n in n_values:
438
439         # running sums to form averages across repeats
440         sum_gEtrain = sum_gEtest = sum_gOracle = sum_gEstVar = 0.0
441         sum_lrEtrain = sum_lrEtest = sum_lrOracle = sum_lrEstVar = 0.0
442
443         for _ in range(num_repeats):
444
445             # dimensions for this n
446             p       = int(c1 * n)
447             N       = int(c2 * n)
448             n_train = int(0.8 * n)
449             n_test  = n - n_train
450             npN     = (n_train, n_test, p, N)
451
452             # synthetic data
453             X_tr, X_te, Y_tr, Y_te, W, b, _, _ = generate_synthetic_data_regression(
454                 npN, density=b_vector_density, d=1, noise_level=noise_level
455             )
456             data = (X_tr, X_te, Y_tr, Y_te)
457
458             # ------- four 's (grid search on the same grid) -------
459             # 1)  from  Ê_train
460             g_Etrain, _ = find_optimal_gamma_grid(gammas_grid, Etrain_, data, npN)
461
462             # 2)  from  Ê_test
463             g_Etest,  _ = find_optimal_gamma_grid(gammas_grid, Etest_,  data, npN)
464
465             # 3)  from oracle derivative  dL/ d_expected = 0
466             def d_oracle(g):   return dL_dy_expected(g, X_tr, Y_tr, W, real_variance)
467             g_Oracle, _ = find_zero_derivative_grid(gammas_grid, d_oracle)
468
469             # 4)  from -datadriven derivative  dL/ d_estimated = 0
470             def d_est(g):      return dL_dy_estimated(g, X_tr, Y_tr, W)
471             g_EstVar, _ = find_zero_derivative_grid(gammas_grid, d_est)
472             # -------------------------------------------------------
473
474             # ridge errors
475             lr_Etrain = L_ridge(g_Etrain, X_tr, Y_tr, W, b)
476             lr_Etest  = L_ridge(g_Etest,  X_tr, Y_tr, W, b)
477             lr_Oracle = L_ridge(g_Oracle, X_tr, Y_tr, W, b)
478             lr_EstVar = L_ridge(g_EstVar, X_tr, Y_tr, W, b)
479
480             # accumulate
481             sum_gEtrain += g_Etrain;  sum_lrEtrain += lr_Etrain
482             sum_gEtest  += g_Etest;   sum_lrEtest  += lr_Etest
483             sum_gOracle += g_Oracle;  sum_lrOracle += lr_Oracle
484             sum_gEstVar += g_EstVar;  sum_lrEstVar += lr_EstVar
```

```
485
486         # averages over repeats
487         n_arr.append(n)
488         gEtrain_arr.append(sum_gEtrain / num_repeats)
489         gEtest_arr.append( sum_gEtest  / num_repeats)
490         gOracle_arr.append(sum_gOracle / num_repeats)
491         gEstVar_arr.append(sum_gEstVar / num_repeats)
492
493         lrEtrain_arr.append(sum_lrEtrain / num_repeats)
494         lrEtest_arr.append( sum_lrEtest  / num_repeats)
495         lrOracle_arr.append(sum_lrOracle / num_repeats)
496         lrEstVar_arr.append(sum_lrEstVar / num_repeats)
497
498         print(f"n={n:4d}␣␣done.")
499
500     # --------------- save for later plotting ------------------
501     np.savez("conv_analysis_n100_200_300.npz",
502              n_values         = np.array(n_arr),
503
504              gamma_Etrain     = np.array(gEtrain_arr),
505              gamma_Etest      = np.array(gEtest_arr),
506              gamma_Oracle     = np.array(gOracle_arr),
507              gamma_EstVar     = np.array(gEstVar_arr),
508
509              lr_Etrain        = np.array(lrEtrain_arr),
510              lr_Etest         = np.array(lrEtest_arr),
511              lr_Oracle        = np.array(lrOracle_arr),
512              lr_EstVar        = np.array(lrEstVar_arr),
513
514              noise_level      = noise_level,
515              repeats          = num_repeats,
516              b_vector_density = b_vector_density,
517              c1 = c1, c2 = c2
518     )
```

A.8. Fashion-MNIST validation

```
import numpy as np
from scipy import linalg
from tensorflow.keras.datasets import mnist,fashion_mnist
import random
import time

SEED = 42                    # <-- one knob for reproducibility
np.random.seed(SEED)
random.seed(SEED)

def sigma(t):
    """
    Small sigma function of choice
    :param t: input
    :return: output
    """
    if activation_function == 'linear':
        return t

    if activation_function == 'ReLu':
        return np.maximum(t, 0)

    if activation_function == 'sign':
        return np.sign(t)


def K(x, y):
    """
    Kernel function. Depends on the choice of a small sigma function
    :param x: first 'point' set
    :param y: second 'point' set
    :return: matrix of measures of 'distances' between points
    """
    if activation_function == 'linear':
        return x.T@y

    if activation_function == 'ReLu':
        norm_x = np.linalg.norm(x, axis=0)  # Shape (n_x,)
        norm_y = np.linalg.norm(y, axis=0)  # Shape (n_y,)

        xTy = x.T @ y  # Shape (n_x, n_y)

        norm_prod = norm_x[:, np.newaxis] * norm_y[np.newaxis, :]  # Shape (n_x, n_y)
        norm_prod = np.maximum(norm_prod, 1e-9)  # Avoid division by zero

        cos_theta = xTy / norm_prod
        cos_theta = np.clip(cos_theta, -1 + 1e-9, 1 - 1e-9)  # Clamp values to [-1+1e-10, 1-1
            e-10]

        theta = np.arccos(-cos_theta)  # Shape (n_x, n_y)
        sin_theta = np.sqrt(1 - cos_theta ** 2)

        return (norm_prod) / (2 * np.pi) * (cos_theta * theta + sin_theta)

    if activation_function == 'sign':
        norm_x = np.linalg.norm(x, axis=0)  # Shape (n_x,)
        norm_y = np.linalg.norm(y, axis=0)  # Shape (n_y,)

        xTy = x.T @ y  # Shape (n_x, n_y)

        norm_prod = np.outer(norm_x, norm_y)  # Shape (n_x, n_y)
        norm_prod = np.maximum(norm_prod, 1e-10)  # Avoid division by zero

        cos_theta = xTy / norm_prod
        cos_theta = np.clip(cos_theta, -1 + 1e-10, 1 - 1e-10)  # Clamp values to [-1+1e-10,
            1-1e-10]

        return (2 / np.pi) * np.arcsin(cos_theta)

```

```
68
69 def K_(x, y, delta, N, n_train):
70     """
71     Function to compute kernel approximation
72     :param x: first 'point' set
73     :param y: second 'point' set
74     :param delta: delta parameter
75     :return: approximation for the kernel matrix
76     """
77     k_ = (N/n_train)*((K(x, y))/(1+delta))
78     return k_
79
80
81 def find_delta(gamma, K_train, N, n_train, accuracy) -> float:
82     """
83     Fixed-point solver for delta without forming any matrix inverse.
84     M(delta) = (N/n)*(K_train/(1+delta)) + gamma*I
85     delta_next = (1/n) * trace( M(delta)^{-1} @ K_train )
86     """
87     delta_prev = 1.0
88     delta_next = 0.0
89     I = np.eye(n_train)
90
91     while abs(delta_prev - delta_next) > accuracy:
92         delta_prev = delta_next
93         a = (N / n_train) / (1.0 + delta_prev)              # scaling factor
94         M = a * K_train + gamma * I                        # (n x n), SPD
95         cf = linalg.cho_factor(M, lower=True, check_finite=False)
96         # U = M^{-1} K_train  (solve for all columns at once)
97         U = linalg.cho_solve(cf, K_train, check_finite=False)
98         delta_next = (1.0 / n_train) * np.trace(U)
99
100    return float(delta_next)
101
102
103 def find_delta_from_eigs(gamma, eigvals, N, n_train, accuracy) -> float:
104    """
105    Fixed-point iteration for delta using only eigenvalues of K_train.
106    a = (N/n) / (1+delta)
107    delta_next = (1/n) * sum_i [ lambda_i / (a*lambda_i + gamma) ]
108    """
109    delta_prev = 1.0
110    delta_next = 0.0
111    while abs(delta_prev - delta_next) > accuracy:
112        delta_prev = delta_next
113        a = (N / n_train) / (1.0 + delta_prev)
114        delta_next = (1.0 / n_train) * float(np.sum(eigvals / (a * eigvals + gamma)))
115    return float(delta_next)
116
117
118 def Etest_(gamma, data, npN):
119    """
120    Eigen/SVD-based E_test (no explicit inverses).
121    Expects 'data' to be a dict:
122        {
123          "K_train": (n,n),
124          "K_xX":    (n,n_test),
125          "K_XX":    (n_test,n_test),
126          "Y_train": (n,),
127          "Y_test":  (n,),
128          "N":       int
129        }
130    npN = (n_train, n_test, p, N)  # N here is ignored; we take N from 'data'
131    """
132    n_train, n_test, p, _ = npN
133
134    K_train = data["K_train"]
135    K_xX    = data["K_xX"]
136    K_XX    = data["K_XX"]
137    y_tr    = data["Y_train"]
138    y_te    = data["Y_test"]
```

```
    N       = int(data["N"])

    # 1) Eigendecomposition of K_train (SPD → eigh is ideal)
    lam, V = np.linalg.eigh(K_train)  # lam: (n,), V: (n,n)

    # 2) Solve for delta using only eigenvalues
    delta = find_delta_from_eigs(gamma, lam, N, n_train, delta_accuracy)

    # Common scaling
    a = (N / n_train) / (1.0 + delta)      # scalar
    d = a * lam + gamma                    # (n,)

    # 3) Prediction term
    y_til = V.T @ y_tr                     # (n,)
    u = V @ (y_til / d)                    # (n,)
    K_xX_scaled = a * K_xX                 # (n, n_test)
    resid = y_te - K_xX_scaled.T @ u       # (n_test,)
    term_pred = (1.0 / n_test) * float(resid @ resid)

    # 4) Ratio term: num / denom
    # num = (1/N) * sum_i (a*lam_i / d_i^2) * (y_til_i)^2
    num = (1.0 / N) * float(np.sum((a * lam / (d * d)) * (y_til * y_til)))
    # denom = (1 - 1/N) * sum_i (a^2 * lam_i^2 / d_i^2)
    denom = (1.0 - 1.0 / N) * float(np.sum((a * a) * (lam * lam) / (d * d)))

    # 5) Bracket term
    # First piece: tr(a*K_XX)
    tr_aKXX = a * float(np.trace(K_XX))
    # Second piece: tr(Q S) + gamma tr(Q^2 S), with S = (a K_xX)(a K_xX)^T
    # If Z = V^T (a K_xX), then diag(V^T S V) = rowwise sum of Z^2
    Z = V.T @ K_xX_scaled                  # (n, n_test)
    diag_Sprime = np.sum(Z * Z, axis=1)    # (n,)
    tr_QS  = float(np.sum(diag_Sprime / d))
    tr_Q2S = float(np.sum(diag_Sprime / (d * d)))
    bracket = (tr_aKXX - (tr_QS + gamma * tr_Q2S)) / n_test

    # 6) Combine
    E_test_ = term_pred + (num / denom) * bracket
    return float(E_test_)


def estimate_variance_RSS(Y, Sigm, n, gamma):
    """
    RSS-based noise variance estimate using SVD of A = Sigm / sqrt(n).
    Equivalent to solving (Sigm^T Sigm / n + gamma I) z = Y, but faster and
    numerically stable.

    Returns a Python float.
    """
    A = Sigm / np.sqrt(n)                          # (N, n)
    U, S, Vt = np.linalg.svd(A, full_matrices=False)  # S: (r,), Vt: (r, n), r = rank
    y = np.asarray(Y).reshape(-1)                  # (n,)
    y_v = Vt @ y                                   # coords in span(A)
    r = S.shape[0]
    denom = S**2 + gamma                           # (r,)

    # Residual in span(A): y_v - (S^2/(S^2+gamma)) y_v = (gamma/(S^2+gamma)) y_v
    rss_span = np.sum((gamma * y_v / denom)**2)

    # Residual in null(A): unchanged (no fit there)
    if r < y.size:
        y_perp = y - Vt.T @ y_v
        rss_null = float(y_perp @ y_perp)
    else:
        rss_null = 0.0

    variance = (rss_span + rss_null) / n
    return float(variance)


def dL_dy_estimated(gamma, X, Y, W, variance=1):
```

```
    """
    Optimized derivative:
        dL/dgamma = gamma * (Y Q^{-3} Y^T) - est_var * tr(Q^{-2}),
    with Q = (Sigm^T Sigm)/n + gamma I, Sigm = sigma(W @ X).

    Uses SVD of A = Sigm / sqrt(n) to avoid explicit inverses.
    """
    # Design in hidden layer
    Sigm = sigma(W @ X)             # (N, n)
    n = X.shape[1]

    # SVD of A = Sigm / sqrt(n)  ->  Q = V diag(S^2 + gamma) V^T
    A = Sigm / np.sqrt(n)
    U, S, Vt = np.linalg.svd(A, full_matrices=False)  # U:(N,r), S:(r,), Vt:(r,n)
    r = S.shape[0]
    denom = S**2 + gamma

    # Coordinates of y in the V basis
    y = np.asarray(Y).reshape(-1)   # (n,)
    y_v = Vt @ y                    # (r,)
    if r < n:
        y_perp = y - Vt.T @ y_v
        y_perp_norm2 = float(y_perp @ y_perp)
    else:
        y_perp_norm2 = 0.0

    # First term: gamma * y^T Q^{-3} y
    term1 = gamma * ( np.sum((y_v**2) / (denom**3)) + (y_perp_norm2 / (gamma**3)) )

    # Noise variance estimate (same as before but SVD-based and exact)
    est_var = estimate_variance_RSS(y, Sigm, n, gamma)  # returns float

    # Second term: est_var * tr(Q^{-2})
    tr_Qinv2 = np.sum(1.0 / (denom**2)) + (n - r) * (1.0 / (gamma**2))
    term2 = est_var * tr_Qinv2

    return float(term1 - term2)


def dL_dy_estimated_fixed_rss(gamma, X, Y, W, variance, gamma_rss):
    """
    "Estimated" derivative: dL/dgamma, but the RSS-based variance is computed
    at 'gamma_rss'. Then the derivative uses the current 'gamma' for the Q
    in the standard formula:
        derivative = gamma*(Y Q_inv^3 Y^T) - est_var * trace(Q_inv^2)
    where est_var is from 'estimate_variance_RSS(Y, Sigm, n, gamma_rss)'.
    """
    n = X.shape[1]
    Sigm = sigma(W @ X)  # shape (N, n)

    # Build Q for *this gamma*:
    Q = (1.0/n) * (Sigm.T @ Sigm) + gamma*np.eye(n)
    Q_inv = np.linalg.inv(Q)
    Q_inv2 = Q_inv @ Q_inv
    Q_inv3 = Q_inv2 @ Q_inv

    # We do the RSS-based variance at 'gamma_rss'
    # Must call 'estimate_variance_RSS':
    est_var = estimate_variance_RSS(Y, Sigm, n, gamma_rss)

    # derivative:
    term1 = gamma * (Y @ Q_inv3 @ Y.T)   # shape (1,1)
    term2 = est_var * np.trace(Q_inv2)
    dL = term1 - term2

    # Convert to float
    if hasattr(dL, "item"):
        return float(dL.item())
    return float(dL)


```

```python
def make_two_class_ridge_from_arrays(
    init_data: np.ndarray,            # e.g. (60000, 28, 28) uint8
    init_labels: np.ndarray,          # e.g. (60000,) int
    selected_labels: list[int],       # exactly two labels, e.g. [1, 2]
    c1: float,                        # target p/n ratio (features-to-sample-size); n  round (p / c1)
    train_fraction: float = 0.8,      # desired train share of (train + test)
    cs: tuple[float, float] = (0.5, 0.5),  # per-class proportions (sum  1.0)
    seed: int | None = None           # RNG seed (None -> non-deterministic like authors)
):
    """
    Rebuild (X, y, X_test, y_test) for a 2-class MNIST/Fashion setup, mirroring the authors:
      1) Sort by label.
      2) Flatten to (p, init_n), rescale to [0,1].
      3) Global mean-center and scale so average ||x||^2 = p.
      4) Select the two classes, pool them, mean-center & scale AGAIN within the pooled subset so
         pooled average ||x||^2 = p.
      5) For each class: shuffle WITHIN class; TRAIN = first int(cs[i]*n);
         TEST  = columns [n : n + int(cs[i]*n_test)] (i.e., test slice starts at absolute index n).
      6) Labels: class 0 -> -1, class 1 -> +1; class blocks contiguous.

    Differences from the raw authors' script:
      - You control n via c1 (p/n), and we derive n_test from `train_fraction`:
            n_test = round( n * (1 - train_fraction) / train_fraction )
        This preserves their splitting scheme while giving you a clean 80/20 (or any) split.
    """
    # ----------------------------
    # A) Sort by label (authors do this)
    # ----------------------------
    idx_sorted = np.argsort(np.array(init_labels))
    labels_sorted = np.array(init_labels)[idx_sorted]
    imgs_sorted = np.array(init_data)[idx_sorted]  # (init_n, H, W)

    # ----------------------------
    # B) Flatten to (p, init_n), cast to float
    # ----------------------------
    H, W = imgs_sorted.shape[1], imgs_sorted.shape[2]
    p = H * W
    init_n = imgs_sorted.shape[0]
    data = imgs_sorted.reshape(init_n, p).T.astype(np.float64)  # (p, init_n); columns = samples

    # ----------------------------
    # C) Global rescale to [0, 1] (authors: data = data / data.max())
    # ----------------------------
    max_val = data.max()
    if max_val > 0:
        data /= max_val

    # ----------------------------
    # D) Global mean-center & renormalize so avg ||x||^2 = p
    # ----------------------------
    mean_data = np.mean(data, axis=1, keepdims=True)         # (p,1)
    centered_global = data - mean_data                       # (p, init_n)
    norm2_data = np.mean(np.sum(centered_global**2, axis=0)) # scalar
    scale_global = np.sqrt(p) / np.sqrt(norm2_data) if norm2_data > 0 else 1.0
    data_std = centered_global * scale_global                # (p, init_n)

    # ----------------------------
    # E) Extract the two classes and pool them
    # ----------------------------
    if len(selected_labels) != 2 or len(cs) != 2:
        raise ValueError("Provide exactly two labels and two class proportions cs=(c0, c1).")

    selected_data = []
    for lab in selected_labels:
        cols = (labels_sorted == lab)
        selected_data.append(data_std[:, cols])              # each: (p, n_class)

```

```
    # Pool both classes (fix authors' missing assignment in np.concatenate)
    cascade_selected = np.concatenate(selected_data, axis=1) # (p, n_pool)

    # ------------------------------
    # F) Recenter & renormalize AGAIN within the pooled two-class subset
    # ------------------------------
    mean_pool = np.mean(cascade_selected, axis=1, keepdims=True) # (p,1)
    centered_pool = cascade_selected - mean_pool
    norm2_pool = np.mean(np.sum(centered_pool**2, axis=0))
    scale_pool = np.sqrt(p) / np.sqrt(norm2_pool) if norm2_pool > 0 else 1.0

    for j in range(2):
        selected_data[j] = (selected_data[j] - mean_pool) * scale_pool

    # ------------------------------
    # G) Choose n from c1 = p/n  ->  n  round(p / c1)
    #    Then compute n_test from the desired train fraction:
    #       n / (n + n_test) = train_fraction  =>  n_test = n * (1 - tf) / tf
    # ------------------------------
    if c1 <= 0:
        raise ValueError("c1 must be positive (c1 = p/n).")
    if not (0 < train_fraction < 1):
        raise ValueError("train_fraction must be in (0, 1).")

    n = int(np.round(p / c1))
    n = max(1, n)
    n_test = int(np.round(n * (1.0 - train_fraction) / train_fraction))
    n_test = max(1, n_test)  # ensure at least one test column overall
    c1_actual = p / n
    train_share_actual = n / (n + n_test)

    # ------------------------------
    # H) Allocate X, X_test and compute per-class block sizes as authors do
    # ------------------------------
    rng = np.random.default_rng(seed)
    X = np.zeros((p, n), dtype=np.float64)
    X_test = np.zeros((p, n_test), dtype=np.float64)

    cs_arr = np.array(cs, dtype=float)
    train_bounds = (np.cumsum(np.concatenate([[0.0], cs_arr])) * n).astype(int)
    test_bounds  = (np.cumsum(np.concatenate([[0.0], cs_arr])) * n_test).astype(int)

    train_counts = np.diff(train_bounds)    # [int(cs[0]*n), int(cs[1]*n)]
    test_counts  = np.diff(test_bounds)     # [int(cs[0]*n_test), int(cs[1]*n_test)]

    # Each class must have >= n + test_counts[i] samples for the authors' indexing
    for i in range(2):
        need = n + test_counts[i]                    # because test slice is data_i[:, n : n
            + test_counts[i]]
        have = selected_data[i].shape[1]
        if have < need:
            raise ValueError(
                f"Class {i} (label {selected_labels[i]}) has {have} samples; need at least {
                    need} "
                f"to reproduce the authors' exact slicing (test starts at index n)."
            )

    # ------------------------------
    # I) Fill X and X_test: shuffle WITHIN each class, then slice as authors do
    # ------------------------------
    for i in range(2):
        perm = rng.permutation(selected_data[i].shape[1])
        data_i = selected_data[i][:, perm]

        # Train block for class i
        X[:, train_bounds[i]:train_bounds[i+1]] = data_i[:, :train_counts[i]]

        # Test block for class i (note: slice starts at absolute index n)
        X_test[:, test_bounds[i]:test_bounds[i+1]] = data_i[:, n : n + test_counts[i]]

    # ------------------------------
```

```
    # J) Labels: class 0 -> -1, class 1 -> +1; contiguous class blocks
    # ------------------------------
    y = np.concatenate([
        -np.ones(train_counts[0], dtype=np.int8),
        +np.ones(train_counts[1], dtype=np.int8),
    ])
    y_test = np.concatenate([
        -np.ones(test_counts[0], dtype=np.int8),
        +np.ones(test_counts[1], dtype=np.int8),
    ])

    info = dict(
        p=p, n=n, n_test=n_test,
        selected_labels=tuple(selected_labels),
        class_sizes=tuple(d.shape[1] for d in selected_data),
        train_counts=tuple(train_counts.tolist()),
        test_counts=tuple(test_counts.tolist()),
        c1_target=c1,
        c1_actual=c1_actual,
        train_fraction_target=train_fraction,
        train_fraction_actual=round(train_share_actual, 6),
        global_scale=scale_global,
        pooled_scale=scale_pool,
        note="Exact␣authors'␣preprocessing;␣test␣slice␣starts␣at␣index␣n␣within␣each␣class."
    )
    return X, y, X_test, y_test, info


def find_optimal_gamma_grid(gammas, f, data, npN):
    """
    Minimizes f(gamma, data, npN) by scanning over 'gammas'.
    Returns (best_gamma, best_value).
    """
    vals = [f(g, data, npN) for g in gammas]
    i_min = np.argmin(vals)
    return float(gammas[i_min]), float(vals[i_min])


def find_zero_derivative_grid(gammas, deriv_func):
    """
    Finds gamma that yields derivative closest to zero in absolute value,
    scanning 'gammas'. Returns (best_gamma, best_deriv_value).
    The caller is expected to provide a function deriv_func(g).
    """
    vals = [deriv_func(g) for g in gammas]
    abs_vals = np.abs(vals)
    i_min = np.argmin(abs_vals)
    return float(gammas[i_min]), float(vals[i_min])


def get_beta(W, X_train, Y_train, gamma):
    """
    SVD version (optional):
      Solve z = (Sigm^T Sigm / n + gamma I)^{-1} Y via SVD of A = Sigm / sqrt(n).
      Then beta = (Sigm / n) @ z.
    """
    Sigm = sigma(W @ X_train)                # (N, n)
    n = X_train.shape[1]
    A = Sigm / np.sqrt(n)                    # (N, n)

    # economy SVD
    U, S, Vt = np.linalg.svd(A, full_matrices=False)  # U:(N,r), S:(r,), Vt:(r,n), r=min(N,n)

    y_v = Vt @ Y_train                         # (r,)
    z_span = (Vt.T * (1.0 / (S**2 + gamma))) @ y_v   # V diag(1/(S^2+)) V^T Y in span(A)

    if Vt.shape[0] < n:                        # nullspace component (if n > r)
        y_perp = Y_train - Vt.T @ y_v
        z = z_span + (1.0 / gamma) * y_perp
    else:
        z = z_span
```

```
488
489     beta = (Sigm / n) @ z                       # (N,)
490     return beta
491
492
493
494 def build_kernels(X_train, X_test):
495     """
496     Compute unscaled kernels once. Scaling by 'a' happens inside Etest_.
497     """
498     K_train = K(X_train, X_train)
499     K_xX    = K(X_train, X_test)
500     K_XX    = K(X_test,  X_test)
501     return K_train, K_xX, K_XX
502
503
504
505 if __name__ == "__main__":
506     import os
507
508     t0 = time.perf_counter()
509
510     # ---------------- Experiment knobs ----------------
511     activation_function     = 'linear'    # 'linear' | 'ReLu' | 'sign'
512     delta_accuracy          = 1e-3
513     c1                      = 1.0
514     c2                      = 1.0
515     gammas_grid_full        = [10**x for x in np.arange(-6, 4, 0.01)]
516     testcase                = 'fashion'   # 'fashion' | 'MNIST'
517     train_fraction          = 0.8
518
519     RNG_SCENARIO_SEED       = SEED + 123
520     N_PIX_SCEN              = 10          # how many single-pixel scenarios
521     N_PATCH_SCEN            = 5           # how many 5x5 patch scenarios
522     PATCH_SIZE              = 5
523     DELTA_STD               = 0.5         # additive perturbation in standardized space
524     LOCAL_LOG_WIDTH         = 1.0         # +- decades around baseline
525     LOCAL_LOG_STEP          = 0.05        # grid resolution in log10
526     RESULTS_DIR             = "results"
527     os.makedirs(RESULTS_DIR, exist_ok=True)
528     # --------------------------------------------------
529
530     # --- Load data (two classes) ---
531     if testcase == 'MNIST':
532         selected_labels = [7, 9]
533         (init_data, init_labels), _ = mnist.load_data()
534     else:
535         selected_labels = [1, 2]
536         (init_data, init_labels), _ = fashion_mnist.load_data()
537
538     # --- Authors' preprocessing / split (deterministic with SEED) ---
539     X_train, y_train, X_test, y_test, info = make_two_class_ridge_from_arrays(
540         init_data=init_data,
541         init_labels=init_labels,
542         selected_labels=selected_labels,
543         c1=c1,
544         train_fraction=train_fraction,
545         seed=SEED
546     )
547
548     # --- Sizes & model shapes ---
549     p        = info['p']
550     n_train  = info['n']
551     n_test   = info['n_test']
552     N        = int(round(c2 * n_train))
553     W        = np.random.randn(N, p)          # fixed W for all scenarios
554
555     # Sanity
556     assert X_train.shape == (p, n_train)
557     assert X_test.shape  == (p, n_test)
558     assert y_train.shape == (n_train,)
```

```
    assert y_test.shape  == (n_test,)

    # --- Precompute baseline kernels once ---
    K_train_base, K_xX_base, K_XX_base = build_kernels(X_train, X_test)

    # --- Baseline: gammas (full grid once), betas, and caches ---
    data_base = {
        "K_train": K_train_base,
        "K_xX":    K_xX_base,
        "K_XX":    K_XX_base,
        "Y_train": y_train,
        "Y_test":  y_test,
        "N":       N
    }
    npN = (n_train, n_test, p, N)

    gamma_Etest_base, _ = find_optimal_gamma_grid(gammas_grid_full, Etest_, data_base, npN)
    def d_est(g): return dL_dy_estimated(g, X_train, y_train, W)
    gamma_direct_base, _ = find_zero_derivative_grid(gammas_grid_full, d_est)

    beta_Etest_base  = get_beta(W, X_train, y_train, gamma_Etest_base)
    beta_direct_base = get_beta(W, X_train, y_train, gamma_direct_base)

    # Helpers: local gamma grids around baseline (± LOCAL_LOG_WIDTH decades)
    def local_grid(g0, width=LOCAL_LOG_WIDTH, step=LOCAL_LOG_STEP):
        g0 = float(g0)
        g0_log = np.log10(g0)
        return 10.0**np.arange(g0_log - width, g0_log + width + 1e-12, step)

    # Random generator for scenario design
    rng = np.random.default_rng(RNG_SCENARIO_SEED)

    # --- Utility: run a single scenario given a perturbed X_train ---
    def run_scenario(X_train_pert, scenario_meta, tag):
        # Kernels for this scenario
        K_tr, K_xX, K_XX = build_kernels(X_train_pert, X_test)
        data_s = {
            "K_train": K_tr,
            "K_xX":    K_xX,
            "K_XX":    K_XX,
            "Y_train": y_train,
            "Y_test":  y_test,
            "N":       N
        }

        # Local gamma search around baseline gammas (fast, robust)
        gE_grid = local_grid(gamma_Etest_base)
        gD_grid = local_grid(gamma_direct_base)

        gE, _ = find_optimal_gamma_grid(gE_grid, Etest_, data_s, npN)
        def d_est_local(g): return dL_dy_estimated(g, X_train_pert, y_train, W)
        gD, _ = find_zero_derivative_grid(gD_grid, d_est_local)

        # Betas
        beta_E  = get_beta(W, X_train_pert, y_train, gE)
        beta_D  = get_beta(W, X_train_pert, y_train, gD)

        # Save all artifacts
        out = {
            "testcase": testcase,
            "activation_function": activation_function,
            "selected_labels": np.array(selected_labels, dtype=int),
            "SEED": SEED,
            "rng_scenario_seed": RNG_SCENARIO_SEED,
            "p": p, "n_train": n_train, "n_test": n_test, "N": N,
            "c1": c1, "c2": c2, "train_fraction": train_fraction,

            "gamma_Etest_base": float(gamma_Etest_base),
            "gamma_direct_base": float(gamma_direct_base),
            "gamma_Etest_scen": float(gE),
            "gamma_direct_scen": float(gD),
```

```
            "beta_Etest_base": beta_Etest_base,
            "beta_direct_base": beta_direct_base,
            "beta_Etest_scen": beta_E,
            "beta_direct_scen": beta_D,

            "scenario_meta": scenario_meta
        }
        fname = os.path.join(RESULTS_DIR, f"{tag}.npz")
        np.savez(fname, **out)
        print(f"[saved] {fname}")

    # --- 10 single-pixel scenarios (std-space) ---
    # choose unique pixels
    pix_ids = rng.choice(p, size=N_PIX_SCEN, replace=False)
    # random signs per scenario (+1 or -1)
    pix_signs = rng.choice([-1.0, 1.0], size=N_PIX_SCEN)

    for idx, (pix, sgn) in enumerate(zip(pix_ids, pix_signs), start=1):
        Xp = X_train.copy()
        Xp[pix, :] += sgn * DELTA_STD
        meta = {
            "type": "pixel",
            "feature_indices": np.array([int(pix)], dtype=int),
            "delta_std": float(DELTA_STD),
            "sign": float(sgn),
            "applied_space": "standardized_training_only"
        }
        tag = f"pert_pix_{idx:02d}_{testcase}_af-{activation_function}_pix{int(pix)}_d{
            DELTA_STD:+.2f}sgn{int(sgn)}_seed{SEED}"
        run_scenario(Xp, meta, tag)

    # --- 5 patch (5x5) scenarios (std-space) ---
    # infer H, W (MNIST/Fashion are 28x28; derive from p if possible)
    H_px = W_px = int(round(np.sqrt(p)))
    if H_px * W_px != p:
        H_px = W_px = 28

    max_r = H_px - PATCH_SIZE
    max_c = W_px - PATCH_SIZE
    rc_pairs = set()
    while len(rc_pairs) < N_PATCH_SCEN:
        rc_pairs.add((int(rng.integers(0, max_r + 1)), int(rng.integers(0, max_c + 1))))
    rc_pairs = list(rc_pairs)
    patch_signs = rng.choice([-1.0, 1.0], size=N_PATCH_SCEN)

    for idx, ((r0, c0), sgn) in enumerate(zip(rc_pairs, patch_signs), start=1):
        # build flat indices of the patch
        feats = []
        for dr in range(PATCH_SIZE):
            rr = r0 + dr
            cc0 = c0
            row_start = rr * W_px
            feats.extend(range(row_start + cc0, row_start + cc0 + PATCH_SIZE))
        feats = np.array(feats, dtype=int)

        Xp = X_train.copy()
        Xp[feats, :] += sgn * DELTA_STD
        meta = {
            "type": "patch",
            "patch_size": int(PATCH_SIZE),
            "top_left_rc": (int(r0), int(c0)),
            "feature_indices": feats,
            "delta_std": float(DELTA_STD),
            "sign": float(sgn),
            "applied_space": "standardized_training_only"
        }
        tag = (f"pert_patch_{idx:02d}_{testcase}_af-{activation_function}_r{r0}c{c0}"
               f"_sz{PATCH_SIZE}_d{DELTA_STD:+.2f}sgn{int(sgn)}_seed{SEED}")
        run_scenario(Xp, meta, tag)
```

```
700     total_sec = time.perf_counter() - t0
701     print(f"[TOTAL] {total_sec:.3f} s")
702     print(f"Baseline gammas: Etest={gamma_Etest_base:.4g}, direct={gamma_direct_base:.4g}")
```