Business Informatics Degree Program
Digital Business Department

2D Image Classification with Random Convolutional Kernels: A ROCKET-Based Approach

BACHELORTHESIS
to obtain the academic degree of Bachelor of Science
Faculty of Electrical Engineering and Computer Science
Ravensburg-Weingarten University of Applied Sciences

presented by
FELICITAS LOCK
Matriculation number: 36434
from Markdorf

reviewed by
Prof. Dr. rer. nat. Markus Schneider
Dr. rer. nat Christopher Bonenberger

Submitted on: February 17th, 2026

Contents

Declaration of Authorship

I hereby declare under oath that I have written the present work independently and have not used any sources or aids other than those indicated, that all parts of the work that have been taken verbatim or in substance from other sources are identified as such, and that the work has not been submitted to any examination board in the same or similar form.

F. Lock

Markdorf, February 15th, 2026

Abbreviations

CNN Convolutional Neural Network. 4, 7–9, 11, 12, 16, 21, 24, 27, 29, 35, 37, 45–47, 53–56, 58

HOG Histogram of Oriented Gradients. 4, 15, 53

LPB Local Binary Pattern. 15

LSPV Local Share of Positive Values. 18, 21–23, 38, 46, 55

Max Max-Pooling. 10, 18, 21, 22, 27, 28, 36, 38, 42, 46, 49, 50, 55

MIPV Mean Index of Positive Values. 18, 21–23, 27, 28, 38, 42, 46, 55

ML Machine Learning. 3

MPV Mean Positive Value. 18, 21, 22, 27, 28, 38, 42, 46, 55

PPV Proportion of Positive Values. 10, 18, 21–23, 27, 28, 36, 38, 42, 46, 49, 50, 55

ROCKET RandOm Convolutional Kernel Transform. 3–12, 14–24, 26–28, 30–33, 35–38, 41–43, 45–47, 49, 52–59

SCNN Shallow Convolutional Neural Network. 16, 17, 53

SVM Support Vector Machine. 7, 8, 11, 12, 15–17, 19, 21, 27, 35, 37, 45–47, 52, 53, 55, 56, 58

1 Introduction

Image classification has become a central task in Machine Learning (ML), with deep learning models achieving remarkable accuracy across various datasets. Classic machine learning algorithms, not only on images but also on time series, often require expensive feature engineering steps, resulting in a demand for high computing effort and correspondingly powerful hardware [1]. Moreover, these models often come with large memory requirements and limited interpretability. Exploring alternative approaches that are computationally efficient, lightweight, and conceptually simple is therefore of growing interest.

RandOm Convolutional Kernel Transform (ROCKET) is a recent method originally developed for time-series classification that offers high efficiency through large banks of random convolutional kernels combined with simple aggregation features. Investigating whether this idea can be transferred to image data may allow insights into efficient, non-deep alternatives for image classification.

1.1 Research Objectives

The main objective of this thesis is to investigate whether ROCKET-style feature extraction can be effectively adapted to image classification.

To address this, the thesis will:

- **(i)** Design a ROCKET-inspired 2D feature extraction pipeline for images using random convolutional kernels and simple aggregation.
- **(ii)** Analyze the impact of kernel size, dilation, number of kernels, and feature selection.
- **(iii)** Evaluate the method on MNIST and CIFAR-10.
- **(iv)** Compare the results to simple classical and shallow model baselines.

1.2 Scope and Exclusion of the Thesis

This work studies the applicability and effectiveness of ROCKET-style feature extraction for image classification using 2D convolutions and aggregation-based features. The scope explicitly includes:

In scope

(i) Large banks of random 2D convolutional kernels with varying kernel sizes and dilations, applied to grayscale and RGB images.

(ii) Aggregation-based features extracted from kernel responses (e.g., PPV, maximum value, mean positive value, and mean index of positive values).

(iii) The use of a linear Ridge classifier trained on frozen feature representations.

(iv) Evaluation on MNIST and CIFAR-10 using consistent training and testing protocols.

(v) Empirical analysis of architectural and hyperparameter choices through controlled ablation studies.

Out of scope

(i) Deep neural network architectures beyond a lightweight Convolutional Neural Network (CNN) baseline. Complex models such as ResNets or Vision Transformers are excluded.

(ii) Learned feature representations and end-to-end training. All convolutional kernels remain fixed after random initialization.

(iii) Heavy handcrafted feature descriptors (e.g., Histogram of Oriented Gradients (HOG)).

(iv) Exhaustive hyperparameter optimization. The focus lies on understanding core ROCKET-style design factors rather than achieving maximal accuracy.

(v) Recent ROCKET variants such as MiniROCKET and MultiROCKET, which focus on kernel sampling optimization and computational efficiency in time series classification. This work pursues a complementary direction by transferring ROCKET-style random convolutions to the two-dimensional image domain.

1.3 Research Questions

This thesis addresses the following research questions:

(i) How well does a ROCKET-style 2D feature extraction approach perform on image classification tasks compared to simple baseline models on MNIST and CIFAR-10?

(ii) How do key design choices, such as kernel size, dilation, number of kernels, and selected aggregation features, influence classification performance and computational cost?

(iii) What are the practical advantages, limitations, and application scenarios of the proposed 2D-ROCKET approach, and how do the experimental findings inform future methodological improvements or potential use cases?

1.4 Thesis Structure

The thesis is organized as follows:

Chapter 2 introduces the theoretical background, including image classification fundamentals and the ROCKET method.
Chapter 3 reviews related work on efficient and non-deep classification approaches.
Chapter 4 presents the methodology provides an overview of the methodological questions, considerations and approach of the work.
Chapter 5 reports the experimental setup and technical details, as well as necessary information to reproduce the presented results.
Chapter 6 describes the experiments and specific settings for each ablation.
Chapter 7 presents the results, analysis of the results and additionally answers the research questions stated in Chapter 4.
Chapter 8 provides an in-depth discussion, and Chapter 9 concludes the thesis and outlines directions for future work.

Throughout the methodological and experimental chapters, a consistent structure is maintained: after introducing the baseline models, successive experimental evaluations and ablation studies are conducted. Insights and results from these analyses are then used to develop and refine the final classification pipeline. The chosen structure aims to move from baselines through increasingly complex experiments to the optimized final pipeline. This approach guides the reader throughout the thesis and clarifies the progression of the research.

2 Theoretical Background

To provide the necessary theoretical foundation for understanding ROCKET, this chapter first reviews classical image classification methods and convolutional neural networks. It then delves deeper into the theoretical foundations and characteristics of ROCKET. The goal is to give the reader the background needed to understand how ROCKET relates to hand-crafted feature pipelines and hierarchical learned representations.

2.1 Convolutional Kernels

Convolutional kernels are a fundamental building block in image processing and image classification, where they are commonly used to detect local patterns within images.

Convolutional kernels, also referred to as filters, are small matrices that are applied to images to perform specific operations such as edge detection, detail enhancement, or noise reduction.

During convolution, a kernel is systematically slid across the image. At each spatial position, a mathematical operation, typically an element-wise multiplication followed by a summation, is performed between the kernel and the underlying image region. The resulting value is assigned to the corresponding output pixel. This procedure corresponds to the discrete form of convolution [2].

The main parameters of a convolutional kernel include its size, weights, bias, dilation, and padding. The kernel weights determine the contribution of each input pixel to the convolution result, while an optional bias term allows for an additional constant offset. While a kernel has the same dimensional structure as the input data, it is typically much smaller in spatial extent [3]. Padding, stride, and dilation control the spatial application of the kernel, affecting the output resolution and the receptive field of the convolution. Padding determines whether the image border is extended during convolution and thus directly affects the output width. Stride defines the kernel step size and therefore the output resolution. Lastly, dilation increases the spacing between kernel weights, allowing larger image areas to be covered by the same kernel [4], [5], [6].

Example of a simple convolution To illustrate the convolution operation and the sliding-window mechanism, consider a 3×3 grayscale image and a 2×2 kernel. For simplicity, the following example uses unit stride, no padding, and no dilation.

$$\text{Input image } I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \text{Kernel } K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

The output size for a 3×3 input and a 2×2 kernel with unit stride is 2×2. At the top-left position, the convolution yields

$$(1 \cdot 1) + (2 \cdot 0) + (4 \cdot 0) + (5 \cdot -1) = -4.$$

The resulting scalar value is assigned to the corresponding pixel in the output feature map. Applying the kernel at all valid spatial locations produces the output

$$\text{Output feature map} = \begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}.$$

Kernels can be designed to achieve different objectives. Common examples include the Gaussian blur kernel, which smooths images and reduces noise, and the Sobel kernel, which emphasizes intensity gradients and is widely used for edge detection [2].

Random Convolutional Kernels The concept of random convolutional kernel fundamentals is particularly relevant for working with ROCKET, as they constitute the core mechanism underlying the ROCKET method. Random convolutional kernels are kernels whose weights are initialized randomly and remain fixed, rather than being learned during training. They can be used to extract generic features from input data in a computationally efficient way. In contrast to learned kernels, random kernels do not require gradient-based optimization [3].

2.2 Existing Image Classification Approaches

Image classification, a core task of machine vision, has historically been solved using classical machine learning methods and hand-crafted feature descriptions. Support Vector Machine (SVM)s, in particular, have proven to be robust and theoretically sound models, enabling effective classification across many feature spaces. The recent development of neural networks, especially CNNs, has made it possible to learn features directly from image data, thus replacing manual feature development [7], [8].

In this work, SVMs serve as a representative classical approach, while CNNs are understood as a representation of the deep learning approach.

Support Vector Machines are supervised learning models for classification and regression. The primary objective is to determine a hyperplane in high-dimensional feature space that separates data points of different classes. That hyperplane aims to have the greatest possible margin, as defined by the distance to the nearest points from either class, known as *support vectors* [9].
Formally, given a labeled dataset $\{(x_i, y_i)\}_{i=1}^{N}$ with features $x_i \in \mathbb{R}^d$ and labels $y_i \in \{-1, 1\}$, the SVM solves the optimization problem [9]

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to } y_i(w^\top x_i + b) \geq 1,\ i = 1, \dots, N$$

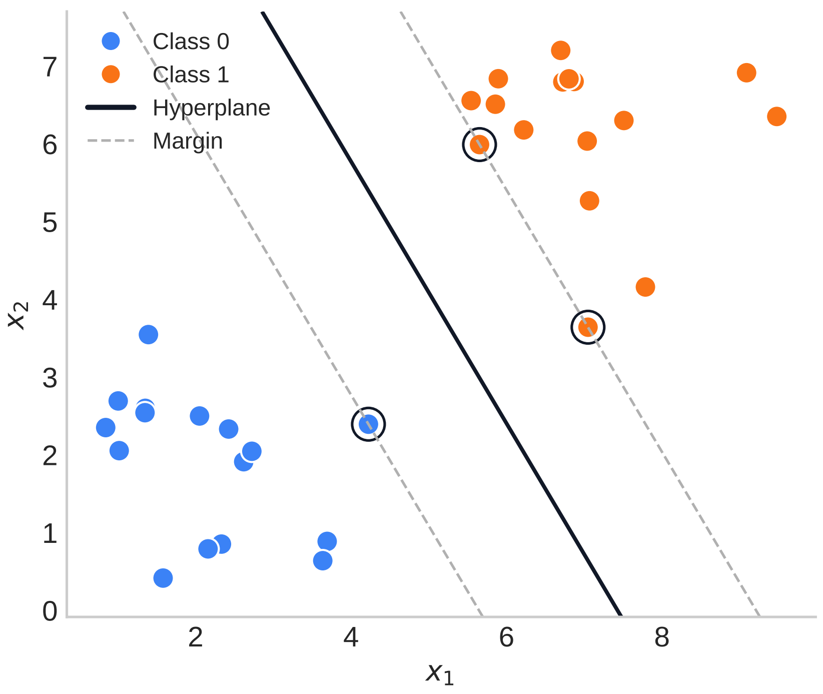


Figure 2.1: Visualization of a support vector machine.

where w defines the hyperplane and b is the bias. The solution maximizes the margin $\frac{2}{\|w\|}$.

SVMs were initially devised for binary classification but are now routinely extended to multi-class settings. For data that is not linearly separable, the kernel trick allows mapping inputs into a higher-dimensional feature space, making linear separation possible without the explicit computation of the mapping [9].

Convolutional Neural Networks are a class of deep learning models specifically designed for image data and have achieved state-of-the-art performance on common benchmark datasets [7]. While classical image processing methods rely on manually designed features, a deep CNN automatically selects and combines relevant features during the learning process. By layering multiple layers, increasingly complex and abstract features can be learned, from local edges to complete object shapes at higher levels. Hence, only the basic structure is described here to provide a general understanding.

First, convolutional layers detect simple local patterns, such as edges and corners. Afterwards, convolutional layers build on these features to recognize increasingly complex structures. These can be textures, shapes, and object parts. Pooling layers then reduce the spatial dimensions that are produced by the convolutional layer. Fully connected layers at the top of the network integrate the information from all previous layers to make the final classification [10].

2.3 ROCKET Basic Idea

ROCKET was originally designed for time series classification with the goal to extract fast and efficient meaningful features from time series. While CNNs learn convolutional

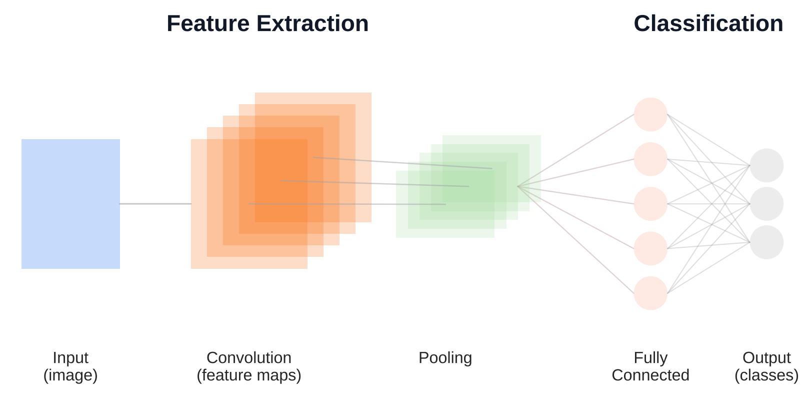


Figure 2.2: Visualization of a simple CNN architecture.

filters through training, ROCKET instead uses a large set of randomly generated convolutional kernels. This approach enables to capture patterns while significantly reducing computational costs [3].

Conceptually, ROCKET uses randomly generated one-dimensional convolutional kernels to extract features from time series. These randomly initialized filters enable the model to capture both simple and complex structures within the data without explicit feature engineering [3].
ROCKET typically relies on a large number of random convolutional kernels, often around 10,000 or more [3]. This redundancy might compensate for the fact that individual random kernels may not be informative on their own. From a theoretical perspective, this idea can be linked to the *Johnson–Lindenstrauss Lemma*, which states that a sufficiently large number of random projections can preserve pairwise distances between samples with high probability [11]. Thus, employing many random convolutional kernels ensures that the resulting feature space maintains much of the structure of the original data while remaining computationally efficient [3].

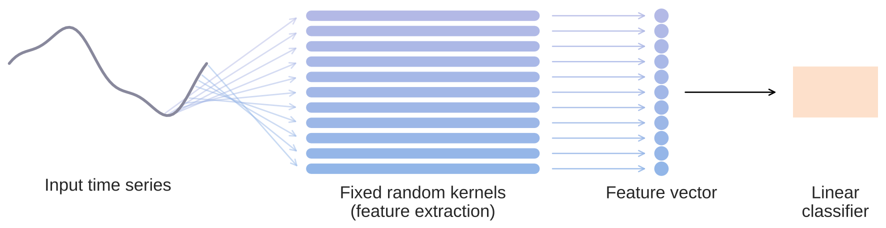


Figure 2.3: Visualization of ROCKET architecture.

Differentiation between Rocket and random convolutional CNNs Although ROCKET relies on convolutional kernels, it is conceptually distinct from random or untrained convolutional neural networks. Random CNNs typically retain a multi-layer architecture with stacked convolutions. Also, normally they consist of nonlinear activation functions and pooling operations. Thereby, they impose an implicit hierarchical structure on the data. In contrast, ROCKET consists of a single, flat feature extraction stage using a large ensemble of independently generated random kernels. No nonlinear activations are learned or stacked, and no hierarchical representations are formed. Instead, kernel responses are summarized by simple, fixed statistical measures and directly passed to a linear classifier. As a result, ROCKET should be understood as a randomized feature mapping rather than a neural network architecture, despite its use of convolutional operations [3].

Feature Extraction For each kernel, there are two main features extracted: Max-Pooling (Max) and Proportion of Positive Values (PPV). Max over the kernel response captures the strongest activation, while PPV measures how often the filter responds positively to the time series [3].
These are defined as

$$\begin{aligned} f_{\max} &= \max_t (x * k)[t] \\ f_{\text{PPV}} &= \frac{1}{L - l + 1} \sum_t \nVdash((x * k)[t] > 0) \end{aligned} \tag{2.1}$$

and lead to the end result being a large, but linearly manageable, feature vector.

Mathematical Basis The core operation in ROCKET is the convolution of a time series x with a randomly generated kernel k of length l

$$(x * k)[t] = \sum_{i=0}^{l-1} x[t + i] \cdot k[i]$$

where x is the input time series, k is the randomly generated kernel of length l and t indexes the positions in the time series.

Convolution Example Consider the input time series and kernel

$$x = [1, 2, 3, 4], \qquad k = [1, -1]$$

where x is the input time series and k the kernel.
The convolution $(x * k)$ is then computed at each valid position t with

$$\begin{aligned} (x * k)[0] &= 1 \cdot 1 + 2 \cdot (-1) = -1 \\ (x * k)[1] &= 2 \cdot 1 + 3 \cdot (-1) = -1 \\ (x * k)[2] &= 3 \cdot 1 + 4 \cdot (-1) = -1. \end{aligned}$$

This results in the convolved series

$$(x * k) = [-1, -1, -1].$$

Feature Extraction Example From the convolved series, two summary features are computed. First, max-pooling extracts the maximum activation value,

$$f_{\max} = \max([-1, -1, -1]) = -1.$$

Second, the proportion of PPV is defined as the fraction of strictly positive responses,

$$f_{\text{PPV}} = \frac{\#\{\text{positive values}\}}{3} = 0.$$

These features from all kernels are concatenated to form a large feature vector. The concatenated feature vectors are then fed into a linear classifier, typically a Ridge regression classifier, which can efficiently learn to discriminate between classes [3].

2.4 Theoretical Time Complexity

In the practical application of machine learning methods, not only predictive accuracy, but also algorithmic efficiency is important. This work places an emphasis on time complexity. When processing large image datasets or operating in resource-constrained environments, scalability becomes a critical factor. Since scalability is largely determined by time complexity, it serves as a criterion for method selection and evaluation. The following discussion is based on general theoretical assumptions for each method and does not refer to specific implementation details, which may lead to additional overhead or optimizations. Instead, the aim of this work is to provide a broad overview. Concrete runtime characteristics of the actual implementations are discussed separately in Chapter 7.

Linear SVMs typically scale roughly as $\mathcal{O}(N \cdot d)$ for N samples with d features, while kernel SVMs can scale between $\mathcal{O}(N^2 \cdot d)$ and $\mathcal{O}(N^3 \cdot d)$ [12].

CNNs require, for a single convolutional layer with input of size $H \times W$ and C_{in} channels and C_{out} filters of size $k_h \times k_w$,

$$\mathcal{O}\big(H \cdot W \cdot C_{\text{in}} \cdot k_h \cdot k_w \cdot C_{\text{out}}\big)$$

multiply-accumulate operations in the forward pass [7]. Pooling and activation layers typically contribute only linearly to the overall computational complexity and are therefore negligible in Big-O terms compared to convolution operations.

ROCKETs ROCKETs main operations are 1D convolutions followed by simple feature aggregation. Let N denote the number of time series, L the length of each series, K the number of kernels, and l the length of each kernel.

Convolution Each kernel is convolved with each time series. As stated above, for a kernel of length l, the discrete convolution at position t is

$$(x * k)[t] = \sum_{i=0}^{l-1} x[t+i] \cdot k[i],$$

where x is the input series and k the kernel [13]. A single kernel on a single series requires therefore $\mathcal{O}(L \cdot l)$ operations, and extending to K kernels and N series gives

$$\text{Cost}_{\text{conv}} = \mathcal{O}(N \cdot K \cdot L \cdot l).$$

[3].

Feature aggregation For each convolved series, each of the two summary statistics requires $\mathcal{O}(L)$ operations. Thus, the total aggregation cost is

$$\text{Cost}_{\text{agg}} = \mathcal{O}(N \cdot K \cdot L).$$

Total complexity Since typically $l \ll L$, convolution dominates the runtime, resulting in an overall computational complexity of

$$\text{Total Cost} \approx \mathcal{O}(N \cdot K \cdot L \cdot l),$$

as reported in the paper of Dempster et al. [3]

Method	**Main Computational Steps**	**Theoretical Time Complexity**
Linear SVM	Training on N samples with d features	$\mathcal{O}(N \cdot d)$
Kernel SVM	Training with kernel matrix on N samples	$\mathcal{O}(N^2 \cdot d)$ to $\mathcal{O}(N^3 \cdot d)$
CNN (single layer)	Forward pass with input size $H \times W$, C_{in}, C_{out} filters of size $k_h \times k_w$	$\mathcal{O}(H \cdot W \cdot C_{in} \cdot k_h \cdot k_w \cdot C_{out})$
ROCKET (on time series)	Feature extraction: N series, K kernels of length l, aggregate statistics	$\mathcal{O}(N \cdot K \cdot L \cdot l)$

Table 2.1: Theoretical time complexity of classical machine learning, deep learning, and ROCKET methods.

Comparison of methods As shown in 2.1, linear SVMs are typically efficient, scaling linearly with the number of samples and features, whereas kernel SVMs can become prohibitive for large datasets due to their quadratic to cubic scaling [12]. CNNs require substantial computation during forward and backward passes, especially as model depth and input size increase [7]. By contrast, ROCKET achieves feature extraction with linear time complexity relative to the number of inputs and kernels, benefiting from not requiring any gradient-based optimization or training [3].

However, a direct complexity comparison between SVM- and CNN-based approaches versus ROCKET is not yet meaningful at this stage due to the differing input domains. The discussion here therefore serves only as a preliminary theoretical reference point. A controlled empirical and theoretical comparison is discussed in chapter 7.

Overall, the favorable computational properties of ROCKET are well established in the one-dimensional setting. The expectation is that the efficiency advantages observed for ROCKET in the 1D domain may transfer to spatially structured 2D inputs. Therefore, ROCKET could enable competitive large-scale feature extraction without the training overhead of deep networks. While this extension is exploratory rather than theoretically guaranteed, it is motivated by the strong empirical performance of ROCKET in high-dimensional time-series classification [3].

2.5 Datasets

In the following section the used datasets are reviewd. These are used for the tests and setup of the pipeline, as well as for the evaluation.

MNIST is a widely used benchmark in the field of image classification. It contains 60,000 grayscale images of handwritten digits (0–9) with a resolution of 28x28 pixels. The dataset is split into 60,000 training images and 10,000 test images. Each image is labeled with the corresponding digit it represents. Due to its simplicity and moderate size, MNIST is commonly used for proof-of-concept experiments and evaluating the basic performance of novel classification methods [14].

Class	**Label**	**Number of training images**
0	zero	5,923
1	one	6,742
2	two	5,958
3	three	6,131
4	four	5,842
5	five	5,421
6	six	5,918
7	seven	6,265
8	eight	5,851
9	nine	5,949

Table 2.2: Class distribution in the MNIST training set A.1.

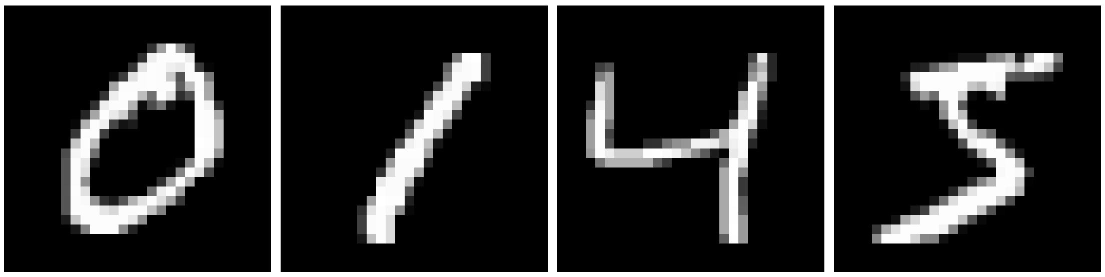


Figure 2.4: MNIST example images

CIFAR-10 is used as a more challenging benchmark for image classification compared to MNIST. It consists of 60,000 color images of 32x32 pixels across 10 classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 test images [15]. Since CIFAR-10 provides a more complex and diverse set of images, it is suitable to evaluate the robustness and generalization ability of the proposed ROCKET-based method.

Class	Label	Number of training images
0	airplane	5,000
1	automobile	5,000
2	bird	5,000
3	cat	5,000
4	deer	5,000
5	dog	5,000
6	frog	5,000
7	horse	5,000
8	ship	5,000
9	truck	5,000

Table 2.3: Class distribution in the CIFAR-10 training set A.1.

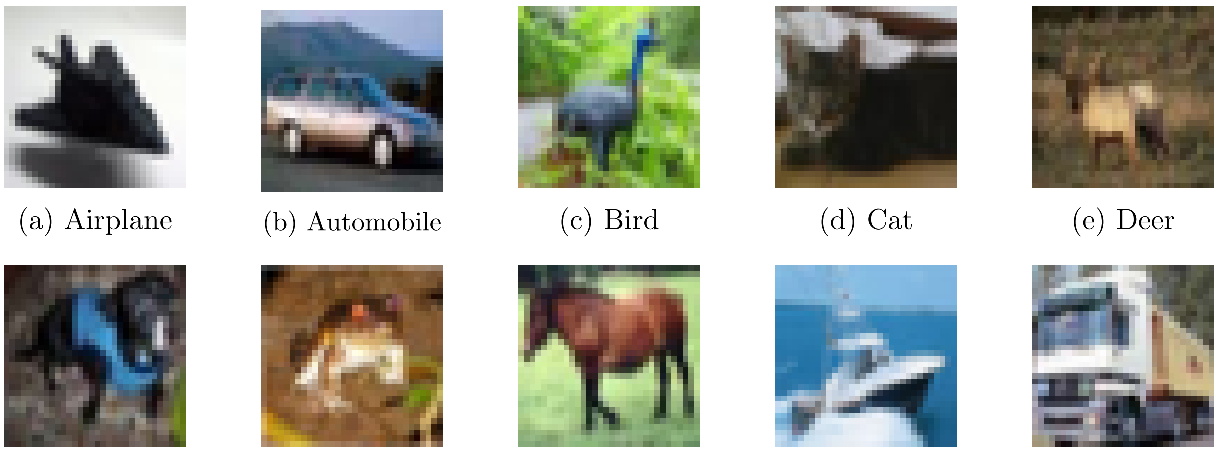

(a) Airplane (b) Automobile (c) Bird (d) Cat (e) Deer
(f) Dog (g) Frog (h) Horse (i) Ship (j) Truck

Figure 2.5: CIFAR-10 example images

3 Related Work

Although research on ROCKET adaptations to images is still limited, several related approaches are worth considering. These methods provide both classification strategies and potential inspiration for the proposed ROCKET adaptation.

3.1 Lightweight Image Classification Approaches

Before the deep learning era, image classification was commonly performed by extracting handcrafted features and training classical machine learning classifiers such as Support Vector Machines, Random Forests, or k-Nearest Neighbors [8]. Classical feature engineering focuses on local structures and simple pooled statistics, using edge and gradient filters (e.g., Sobel, Canny) to capture orientation and boundaries [16]. Descriptors derived from gradients, such as HOG, summarize shape and contour patterns [17].

HOG and Local Binary Pattern (LPB) are classical handcrafted feature descriptors. HOG computes gradient magnitudes and directions in small spatial cells, accumulating them into histograms. It effectively encodes object contours and is robust to illumination changes [18]. LPB encodes local texture by thresholding each pixel against its neighbors to form a binary pattern, summarized over image regions as histograms [19]. Both descriptors are simple, interpretable, and widely used in conjunction with linear classifiers, providing a strong baseline before the deep learning era.

Support Vector Machines were a standard classifier for image recognition tasks prior to the deep learning era [8]. They were most effective when used in conjunction with handcrafted features such as HOG or LPB [17], [18]. These pipelines typically consist of a feature extraction stage, followed by an SVM as the final classifier [20]. Linear SVMs are more transparent than deep networks, as their decision function is a weighted sum of features [21]. Non-linear SVMs, however, lose some interpretability due to kernel transformations [21]. Extensions to kernelized SVMs enabled non-linear decision boundaries, although often with a much higher computational cost on larger datasets [9].

Furthermore, SVM performance is highly sensitive to hyperparameter choices such as the regularization parameter C and kernel-specific parameters. This typically requires careful tuning, usually via cross-validated grid search [9]. Standard SVMs can also struggle with class imbalance. This is due to symmetric hinge loss, and a single misclassification cost tends to bias the decision boundary toward the majority class. In addition, training and inference costs increase with the number of support vectors, which can be substantial on large datasets [22].

However, SVMs provide overall a well-understood baseline for image classification, leveraging handcrafted local features with a simple linear or kernel classifier. This makes them a natural reference point when evaluating alternative feature representations, including random convolutional approaches such as ROCKET.

Convolutional Neural Networks continue to dominate state-of-the-art image classification due to their strong representational power. On the other hand, full-scale deep networks can be computationally expensive and memory-intensive [7]. This can limit their applicability in scenarios with constrained resources [23]. Lightweight convolutional networks aim to bridge this gap by delivering competitive performance while reducing computational costs and memory requirements [24].

Lightweight CNNs often use manually designed architectures with few convolutional layers, simple pooling operations, and standard nonlinearities. Techniques such as depthwise separable, group, or dilated convolutions reduce computational cost while preserving an effective receptive field. Residual connections can further improve training stability in slightly deeper networks. Overall, these networks are computationally efficient and require less training effort than deep architectures, making them practical baselines for evaluating alternative feature-extraction strategies [23], [24].

Within this category, Shallow Convolutional Neural Network (SCNN)s provide a concrete example. SCNNs consist of significantly less complex architecture and fewer parameters. They employ, for example, fewer convolutional layers and filters than deep networks. This makes them computationally efficient while still learning task-specific features from data [25]. Despite their simplicity, SCNNs can achieve competitive performance compared to deeper CNNs on small-scale image datasets [26]. Empirical studies, such as PCANet, demonstrate that analytically derived or unsupervised filters in shallow stacks can produce useful image representations with minimal training overhead [24]. While SCNNs learn a small set of convolutional filters from data, ROCKET-style adaptations rely instead on large ensembles of randomly initialized kernels combined with simple statistical aggregation. Due to their focus on local feature transformations, both approaches are well suited for comparison in lightweight image-classification settings [3].

Automated methods like Neural Architecture Search (NAS) offer another way to design efficient, lightweight networks with minimal human effort [24]. These searches operate at multiple granularities, from refining individual layers to arranging repeated cells or stages across the network. While NAS can yield highly optimized architectures for specific tasks, the search itself is often computationally intensive and can still benefit from expert guidance [24].

Overall, lightweight convolutional methods, including SCNNs, illustrate the design space for efficient image feature extraction. They highlight the importance of balancing computational cost, model complexity, and representational power. Notably, the success of shallow and fixed- or random-filter approaches shows that meaningful feature representations can be obtained without heavy end-to-end training. This observation directly motivates the adaptation of ROCKET's random-kernel and statistics-based paradigm to two-dimensional image data. ROCKET provides a potentially low-cost, interpretable, and flexible alternative to fully learned CNNs.

Pipeline	**Feature Type**	**Classifier**	**Trainable Parameters**	**Main Advantages**
HOG + SVM	Handcrafted gradients	SVM (linear or kernel)	Only classifier	Robust, interpretable, efficient on small-/medium data
LBP + SVM	Handcrafted textures	SVM (linear or kernel)	Only classifier	Good for micro-textures, simple, fast
Shallow CNN	Learned filters (few layers)	Linear/ Classical	Filters + classifier	Moderate training, mix of learning & efficiency
Random CNN (untrained)	Random stacked filters	Linear/SVM	Only classifier	No training, moderate depth, some DL structure
2D-ROCKET	Large ensemble of random kernels	Linear/ RidgeClassifier	Only classifier	No training, simple statistics, efficient, interpretable, strong on structure

Table 3.1: Comparison of typical image feature extraction pipelines and classifiers.

Both SVM-based pipelines and SCNNs provide reference points for evaluating ROCKET adaptations in the 2D domain. SVMs represent classical feature-based approaches, while SCNNs exemplify learned but lightweight convolutional representations. Comparing ROCKET to these baselines highlights its ability to efficiently extract and aggregate local features without extensive parameter learning [3].

3.2 Variations and extensions of ROCKET

Several ROCKET variants now exist, of which selected ones will be listed below.

MiniROCKETs' main improvements address computational efficiency and determinism while maintaining competitive classification accuracy. It is characterized by the use of a fixed kernel size of 9 and a highly constrained kernel structure. Kernel weights are restricted to two values, $-1, 2$, with a fixed number of positive and negative weights per kernel. This design allows convolution operations avoiding multiplications, leading to a significant reduction in computational cost [27]. Unlike the original ROCKET, which extracts both the Max and PPV per kernel, MiniROCKET uses only PPV, discarding the maximum statistic. As a result, MiniROCKET produces exactly one feature per kernel. For a configuration with 10,000 kernels, this yields 10,000 features, compared to 20,000 features in the original ROCKET framework. Deterministic kernel generation ensures reproducible outputs across runs. The design of MiniROCKET represents a trade-off between computational efficiency and feature flexibility [27], [28].

MultiROCKET extends the enhancements introduced in MiniROCKET to further improve classification performance. The central idea is to include feature extraction from first-order differences and to introduce three additional pooling operations. In addition to PPV, MultiROCKET computes the Mean Positive Value (MPV), the Mean Index of Positive Values (MIPV), and the Local Share of Positive Values (LSPV). These additions aim to capture inter-channel interactions and exploit multi-dimensional signal structure, which is particularly interesting for two-dimensional image data [29].

MultiROCKET-Hydra builds upon MultiROCKET by introducing a dictionary-inspired component. Kernels are organized in groups, and Hydra records how often a particular kernel within its group produces the highest activation. This produces a frequency vector similar to a dictionary representation. MultiROCKET-Hydra demonstrates superior performance compared to vanilla MultiROCKET in time series classification benchmarks, particularly for large datasets and tasks involving structured patterns, like image outlines. Its underlying principle resonates with the motivation for 2D-ROCKET: capturing structured spatial interactions across multiple channels while keeping computation manageable [30].

Variant	Feature types	Features (10k kernels)	Relevance to 2D images
MiniROCKET	PPV only	10,000	Efficient, deterministic; minimal feature cost
MultiROCKET	PPV, MPV, MIPV, LSPV	50,000	Captures inter-channel / multi-dimensional info; motivates spatial statistics
MR-Hydra	MultiROCKET + kernel group counts	50,000+	Adds frequency/-dictionary info; structured activation patterns relevant for 2D

Table 3.2: Summary of ROCKET variants, their features, and relevance for image data.

3.3 ROCKET on Images

While most work with random convolutions focuses on 1D time series, extending their success to images opens the door for highly efficient, deep-learning-independent pipelines in vision applications.

The idea of using random convolutional kernels for image classification is not new. Approaches that utilize randomized features in combination with linear classifiers were already presented in the late 2000s. Rahimi and Recht propose a technique to map data into a randomized low-dimensional feature space, in which the inner product approximates a specific kernel, allowing fast linear learning for non-linear problems [31].

Recent studies show that convolutional architectures with random filters can generate high-quality features for image classification without learning weights. For example, Xue and Roshan apply recognition with deep random architectures, followed by global pooling and a linear classifier, achieving competitive accuracy on benchmarks such as CIFAR-10 and STL-10 [32].

Another example is found in the work of Ren and Luo, where random filters followed by ReLU activation, pooling, and a linear SVM achieved competitive performance on facial expression classification (approx. 84%) without learned weights [33]. These results suggest that random convolutional representations are informative for visual tasks even in the absence of filter learning.

During the research, it became clear that many modern approaches to image classification with randomized convolutions still rely on deep learning architectures, such as stacked convolutional blocks and pooling operations.
In contrast, ROCKET represents a fundamentally different approach. It completely dispenses with learned neural network weights. Instead, ROCKET uses massive ensembles of simple, independently initialized random kernels, whose outputs are aggregated via

straightforward summary statistics as features for a classic linear classifier [3].

Adapting this to two-dimensional image data provides a conceptually distinct, non-deep-learning baseline, highlighting the potential of random features for pattern recognition without any learned representations or architectural bias.

3.4 Classification of the Thesis

There remains a notable gap for low-effort, computationally efficient image-classification algorithms that deliver useful accuracy with minimal training. Modern state-of-the-art solutions often rely on deep, parameter-heavy convolutional networks requiring large labeled datasets, substantial compute, and specialized hardware. Many real-world applications, however, demand lightweight solutions, e.g., on-device vision for IoT, real-time inspection systems, or rapid prototyping in domains with scarce labeled data. Here, 2D-ROCKET refers to the application of ROCKET's random-kernel and statistics-based paradigm directly to two-dimensional image inputs, using random convolutional filters and simple summary statistics for linear classification in the two-dimensional space.

Applying time-series methods directly to images is limited, as 1D kernels cannot capture 2D spatial or multi-channel correlations. Adapting ROCKET to 2D, while preserving its positive properties (random kernels, cheap and interpretable statistics, minimal learning), might enable efficient and robust feature extraction suitable for small to medium datasets. This approach occupies a middle ground between heavy deep learning stacks and classical handcrafted pipelines, combining computational and sample efficiency with image-appropriate spatial processing. Building on these insights, this work presents and evaluates an explicit 2D adaptation of ROCKET for image classification, aiming to systematically assess its strengths and limitations relative to both classical and deep learning approaches.

4 Methodology

This study investigates the added value of a 2D adaptation of the ROCKET framework under fully controlled and reproducible conditions. The proposed 2D-ROCKET pipeline is evaluated on **MNIST** (28×28 grayscale) and **CIFAR-10** (32×32 RGB). The approach is then compared against lightweight baselines: a minimal 1D-ROCKET pipeline, a linear SVM, and a CNN.
Rather than aiming for state-of-the-art performance, this work focuses on isolating the effects of spatially structured random convolutions, as well as on feature design choices. Additionally, it addresses assessing the potential of a ROCKET pipeline in two dimensions. Across all experiments, global defaults such as device configuration, random seeds, dataset normalization, batch size, and evaluation protocol are fixed and documented in Chapter 5.

4.1 Research Questions

Each experiment family is designed to answer a specific research question:

Experiment	**Research Question**
1D Feature Ablation	Do additional summary statistics (MPV, MIPV, LSPV) improve performance over the original PPV+Max features on flattened inputs?
1D vs. 2D Kernels	Does preserving spatial structure with 2D convolutions outperform flattened 1D kernels at comparable budgets?
2D Padding Ablation	How do different padding strategies—*same*, *valid*, and a per-kernel *random* mix—affect final classifier accuracy and the resulting feature dimensionality?
Preprocessing Banks	Do kernels applied to preprocessed channels yield better features than kernels over dataset-normalized raw inputs, and is there an additive benefit when concatenating separate kernel banks over raw and preprocessed channels?
Kernel Configuration	Which kernel size and dilation combinations perform best under a fixed kernel count?
Kernel Count Scaling	How does performance scale with the number of kernels?
Final Evaluation	What accuracy is achieved by the final 2D-ROCKET configuration on each dataset? How does it perform compared to the defined baselines?

4.2 Experimental Strategy and Ablation Logic

The methodology follows a staged ablation strategy, where each experiment isolates a single design decision and informs subsequent choices. Decisions are propagated forward, ensuring that the final configuration is justified by empirical evidence.

1. **Feature Set (1D):** The original ROCKET feature set (PPV and Max) is compared against extensions incorporating MPV, MIPV, and LSPV. The best-performing combination defines the feature template for subsequent 2D experiments.
2. **1D vs. 2D Kernels:** Using the selected feature set, 1D and 2D kernels are compared to evaluate whether preserving spatial structure leads to improved performance. Comparisons are performed at matched kernel counts rather than matched feature dimensionality.
3. **Padding Strategy (2D):** The notebook evaluates three padding modes `same`, `valid`, and a per-kernel `random` mix. For `valid` padding the implementation filters out kernels whose effective receptive field exceeds the image height or width, which can reduce the extracted feature dimensionality.
4. **Preprocessing Banks:** Kernels are evaluated on two preprocessing regimes implemented in the notebook: dataset-normalized loader output and per-image, per-channel standardization. The ablation builds separate kernel banks for raw RGB and for the selected preprocessed channels. Experiment specs split a fixed total kernel budget across banks.
5. **Kernel Configuration (Phase 1):** Kernel sizes and dilation patterns are ablated to identify the most effective configuration at a fixed kernel count.
6. **Kernel Count Scaling (Phase 2):** The selected configuration is evaluated across increasing kernel counts to quantify performance gains and computational trade-offs.

4.3 Experimental Protocol

Data Loading and Preprocessing All datasets are normalized using fixed, dataset-specific statistics applied in the data loader. Model-specific preprocessing used for feature extraction is applied inside the feature extraction pipeline and can be forced. By default a heuristic may skip per-image normalization when the loader already applies dataset-level normalization. No data augmentation is applied in the base experiments.

Feature Design Since the original ROCKET feature set focuses on simple summary statistics such as the proportion of PPV and Max, additional statistics are included to capture broader spatial information across the input.

Inspired by the idea of the MultiROCKET approach the MPV is evaluated, which summarizes the average strength of positive kernel responses. This provides complementary information about the typical activation magnitude and reduces sensitivity to single extreme responses. It is particularly relevant for image data, where informative patterns are distributed over two-dimensional regions rather than strictly sequentially as in time series [29].

To complement these statistics, the MIPV is investigated, which measures the average spatial location of positive kernel responses. This statistic can be interpreted as a first-order moment of the activation distribution and provides coarse localization information [29].

Additionally, to further characterize the spatial distribution of kernel activations, the LSPV is investigated. Unlike PPV and MIPV, which summarize global frequency and mean position, LSPV measures the proportion of positive responses within predefined local regions. This statistic aims to capture coarse spatial concentration patterns of activations while remaining computationally inexpensive and consistent with the design philosophy of ROCKET [29].

Feature Extraction and Classification Extracted features are standardized as part of the cla Extracted features are standardized within the classification pipeline. The Ridge pipeline uses a StandardScaler, fitted on the training set and applied to both training and test features, to ensure zero mean and unit variance before classifier fitting. Note that dataset-level image normalization with fixed constants for MNIST/CIFAR) is a separate step that normalizes raw images before feature extraction. Per-image preprocessing is an additional, optional step applied during feature extraction and can be forced to override the loader normalization. All classifications are performed using a Ridge classifier. During ablations, a small grid over the regularization parameter α is evaluated.

4.4 Fairness and Reproducibility

Reproducibility and fairness protocols are enforced to ensure that all experimental results are comparable and scientifically valid. They apply to both the study-wide and within-experiment levels. For specific technical information see Chapter 5.

Across-Experiment Fairness All models and experiments share key global settings: fixed random seeds, uniform dataset normalization, identical train/test split protocols, and standardized evaluation metrics. No data augmentation, ensemble methods, or external pretraining are used. This aims to ensure that baseline comparisons reflect only differences in feature design or model architecture.

Within-Experiment Fairness To isolate the impact of specific design choices, such as kernel configuration, feature set, or preprocessing regime, all variants within an ablation or experiment are run with matched settings for data subsampling, batch size, random seed, and classifier configuration. This guarantees that observed performance differences are attributable solely to the intended modification and not to auxiliary changes in data or training conditions.

Seeding and Determinism All experiments use a fixed random seed, with a default of 42. Deterministic CuDNN settings are enabled, and benchmarking is disabled to ensure reproducibility across runs. When subsampling is applied, per-class indices are generated deterministically once per dataset and reused across all experimental variants.

4.5 Evaluation Metrics

Accuracy is used as the primary evaluation metric. Both MNIST and CIFAR-10 are balanced multi-class classification tasks with uniform misclassification costs. This allows accuracy to be used as a metric without the risk of distortion of the metric by unbalanced classes. In this setting, accuracy provides a clear and interpretable measure of overall classification performance. Furthermore, it allows direct comparison across experimental variants.
The accuracy A is defined as the proportion of correctly classified samples over all samples as

$$A = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\hat{y}_i = y_i)$$

where N is the total number of test samples, y_i the true label, $\hat{y}_i$ the predicted label, and $\mathbb{I}$ is the indicator function.

Per-class accuracy and confusion matrices are partially checked as secondary metrics to identify class-specific effects and ensure that performance gains are not driven by a small subset of classes.
Training time denotes the time required for the classifier fit (and full network training for CNNs). It is reported to capture computational efficiency, reflecting classifier fitting. Feature extraction times for ROCKET are not systematically logged.

4.6 Non-Goals and Explicit Exclusions

To keep conclusions attributable and the scope focused, several aspects are explicitly excluded. The classifier family is not benchmarked as a research question, meaning Ridge regression is fixed to isolate feature quality. Hyperparameter optimization beyond small, predefined grids is out of scope. No data augmentation, task-specific priors, or external pretraining is used. The work does not aim at end-to-end representation learning but focuses on random convolutional feature extraction. Rather than aiming for state-of-the-art performance, this work focuses on isolating the effects of spatially structured random convolutions and feature design choices.

4.7 Threats to Validity and Limitations

Single-seed evaluation Experiments use a fixed seed for determinism. While optional sensitivity runs can report mean and standard deviation, single-seed results may under-represent variance. To reduce the computation time of the experiments and enable fast iterations, only a single seed is used for each run.

Fixed classifier The use of a fixed Ridge classifier isolates feature quality but excludes potential gains from alternative classifiers.

No augmentation or pretraining The absence of augmentation and pretraining ensures fair comparisons but may understate achievable performance.

Kernel and padding choices Kernel sizes, dilations, padding strategies, and budgets are explored within constrained grids. Conclusions are valid within these ranges.

Computational environment Deterministic settings may slightly reduce throughput compared to non-deterministic configurations.

5 Technical Details

This chapter documents the shared technical backbone for all experiments: devices, seeding, data loading, normalization, ROCKET feature mechanics, and classifier setup. Details about alternative padding regimes, feature sets, and kernel configurations evaluated in ablations are provided in Chapter 4 and 6.

5.1 Experimental Environment

Experiments run on GPU (CUDA) when available. Otherwise, CPU is used. Determinism and seeding are enforced uniformly (see A.2). Deterministic algorithm flags and environment variables are enabled to improve reproducibility. However, full bit-wise determinism on CUDA depends on driver/hardware support and certain BLAS/CuBLAS operations may remain non-deterministic in practice.

Parameter	**Value / Setting**
Device	`cuda` (if available), else `cpu`
Random seed	42 (`random`, `numpy`, `torch`)
Batch size	512
Num. workers (DataLoader)	2
Normalization (MNIST)	mean 0.1307, std 0.3081
Normalization (CIFAR-10)	mean (0.4914, 0.4822, 0.4465), std (0.2470, 0.2435, 0.2610)
ROCKET-specific	
Classifier (ROCKET)	Ridge
Ridge alpha (ROCKET)	grid search for ablations and final pipeline
Feature statistics (ROCKET)	PPV, Max (additional features in some ablations)

Table 5.1: Summary of key global experimental constants and ROCKET-specific settings.

When subsampling is used (e.g., quick-mode baselines), per-class indices are generated deterministically and reused across models to ensure fairness and reduce variance unrelated to the tested factors.

5.2 Datasets and Normalization

Two standard computer vision datasets are used:

MNIST: 28×28 grayscale images; dataset-level normalization with mean (0.1307) and std (0.3081).

CIFAR-10: 32×32 RGB images; dataset-level normalization with mean $(0.4914, 0.4822, 0.4465)$ and std $(0.2470, 0.2435, 0.2610)$.

For each ablation, baseline and the final pipeline the default dataset splits (MNIST 60k/10k, CIFAR-10 50k/10k) are used.
The specific values for the mean and standard deviation were calculated separately and correspond to the commonly used values. After manual calculation, these values were fixed and applied to all experiments and setups.

Dataset-level normalization via `torchvision` ensures a comparable input scale across all models (CNN, SVM, ROCKET). It functions as the standard conditioning for these benchmarks. It reduces covariate shift between channels and stabilizes downstream training and evaluation without introducing dataset-specific learnable transforms.

For the preprocessing ablation, a deterministic per-image, per-channel z-score standardization is applied before convolutional feature extraction as

$$x' = \frac{x - \mu}{\max(\sigma, \varepsilon)},$$

with $\varepsilon = 10^{-6}$ to avoid division by very small variances. This step was implemented to make random kernel responses depend on structure rather than absolute brightness/contrast, yielding scale-invariant statistics (PPV, Max, MPV, $\text{MIPV}_y/\text{MIPV}_x$). Furthermore, it also improves numerical stability (clamping avoids division by near-zero variance regions) and aligns feature distributions for the global `StandardScaler` used prior to the Ridge classifier. The two normalization steps operate at different levels and are therefore not overlap.

5.3 ROCKET Feature Mechanics

Kernel initialization Random kernels are sampled with variance $\sqrt{2/\text{fan-in}}$ for weights and biases drawn from $\mathcal{N}(0, 0.1)$. Stride is fixed to 1. With per-image z-score inputs (mean 0, std 1) and a positive-activation mask (PPV, Max, MPV), He initialization preserves forward variance for ReLU-like statistics. This makes the convolution outputs roughly of size 1, which helps the features stay stable and avoids very small or very large values. For specific implementation details see Appendix A.4.

Small random bias $\mathcal{N}(0, 0.1)$. A zero bias on zero-mean inputs can yield PPV values clustered around ~ 0.5 due to symmetry. The small, random bias helps kernels respond differently. This gives a more varied PPV values without changing the overall signal too much.

Stride = 1. Full-resolution sampling preserves spatial granularity needed for location-sensitive features (MIPV$_y$, MIPV$_x$) and yields consistent receptive-field coverage across kernel sizes/dilations. Larger strides would downsample activation maps, altering PPV or MPV statistics and confounding comparisons.

Minimal 1D-ROCKET The minimal 1D-ROCKET implementation, which serves as the basis for all further steps, is a minimal independent implementation, based on the method described in the paper by Dempster et al. [3] The purpose of this choice was to gain a complete understanding of the algorithm, ensure full transparency and control over subsequent experiments. Key properties of the original ROCKET design, including random kernel generation, the use of only Max and PPV summary statistics, and a linear classifier, are preserved to maintain comparability. This minimal pipeline serves as a clean, reproducible baseline. It allows systematic exploration of extended feature sets, 2D kernels, and other design choices without introducing unintended deviations from the original method.

In order to make only minimal adjustments, images are flattened (row-wise). For each (kernel size k, dilation d), 1D convolutions, based on the template by Dempster et al., use random padding and therefore select whether to use same or valid. In 'random' padding mode, the implementation generates a pad map that selects between 'same' and 'valid' once for each kernel group (k,d) (at the start of extraction) using a pseudo-random RNG (seed-controlled). This makes the selection deterministically reproducible with the same seed, and the padding decision remains constant across all batches of a run. In 'random' padding mode the implementation generates a pad map via a pseudo-random generator seeded with the global seed. The per-kernel choice between 'same' and 'valid' is therefore reproducible for a given seed.

Key components of the 1D ROCKET implementation (kernel sampling, padding strategy, and feature statistics) are illustrated in the Appendix A.10.

Standardization and classifier There are three distinct normalization/standardization steps in the pipeline:

1. **Dataset-level image normalization (loader):** Fixed, dataset-specific constants are applied. This normalizes raw pixel values before any feature extraction and ensures comparable input scales across models.

2. **Optional per-image preprocessing (during feature extraction):** An optional, ablation-specific step performs deterministic per-image, per-channel z-score standardization inside the feature extraction pipeline. This step can be forced even when the loader already applies dataset normalization and is intended to make kernel responses scale-invariant. It is applied at the image level immediately before convolutions.

3. **Feature standardization (classifier pipeline):** The extracted ROCKET features are standardized as part of the classification pipeline: `build_ridge(...)` inserts a `StandardScaler` (fitted on training features and applied to train/test)

before the `RidgeClassifier`. This ensures feature means and variances are appropriate for the linear classifier and is distinct from the image-level normalizations above.

These steps operate at different levels (image vs. feature) and are complementary. Loader normalization controls input statistics for the convolution stage. The optional per-image preprocessing enforces invariance for individual images during extraction, and the pipeline scaler conditions the final feature matrix for stable classifier training.

Classification uses a `RidgeClassifier`. For the ablation experiments, all candidates $\alpha \in \{1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400\}$ were evaluated. The 'best' α highlighted in the text and tables was selected based on the test set evaluation. This approach can lead to optimistic bias. For unbiased final measurements, it is recommend selecting α on a separate validation set or by cross-validation on the training data.

5.4 SVM and CNN Baselines (Technical Skeleton)

Linear SVM A linear Support Vector Machine is trained on flattened and standardized pixel values. For high-dimensional datasets such as CIFAR-10, a random projection step reduces the dimensionality while approximately preserving distances. The value of the regularization parameter C=0.1 is used.

Lightweight CNN The small CNN used in this work consists of two convolutional layers with 32 and 64 channels, respectively, each followed by ReLU activation and max-pooling. The feature maps are then flattened and passed through a small multilayer perceptron (linear–ReLU–linear) to produce class logits. No data augmentation or Batch Normalization is applied. Training is performed for a fixed number of 10 epochs.

5.5 Evaluation and Reproducibility

The following points summarize the key aspects considered, that were implemented in order to a achieve reproducable evaluations.

Primary metric: test accuracy.

Secondary diagnostics: training time (classifier fit; feature extraction time reported in ablations as relevant), confusion matrix, per-class accuracy.

Reproducibility: fixed seeds, deterministic flags, and consistent subsample indices (when used) ensure repeatability.

Artifacts: CSV logs of results and configuration; optional feature caches, confusion matrices and predictions can be saved as needed.

Note on hyperparameter selection: The 'best α' specified in the results tables was selected from the tested grid based on the test accuracy. For final, unbiased reporting, hyperparameter tuning on a validation set or using CV is required.

5.6 Environment and Versions

The runtime environment is documented in order to support exact reproducibility.

Runtime and OS Google Colab High-Memory runtime (Linux). GPU used when available.

Hardware Hardware: Experiments were run in a Colab High-Memory Runtime. All runs reported here ran on an NVIDIA A100 GPU (approx. 40–80 GiB VRAM, depending on the image).

Software versions Python 3.12; PyTorch 2.9+cu128; Torchvision 0.24+cu128; scikit-learn 1.6; NumPy 2.0; CUDA/cuDNN versions per Colab image. Exact versions should be recorded at run time.

Specific Setup for this Work: Python: 3.12.12 (main, Oct 10 2025, 08:52:57) [GCC 11.4.0] PyTorch: 2.9.0+cu128 Torchvision: 0.24.0+cu128 scikit-learn: 1.6.1 NumPy: 2.0.2 CUDA: 12.8 cuDNN: 91002

5.7 Instructions for Reproduction

1. Use the same runtime (Colab High-Memory) and ensure a CUDA GPU (e.g., T4, V100, or A100) is available. Enable GPU in the notebook runtime.
2. Run the notebook (Rocket2D.ipynb) end-to-end with the default global settings (seed=42, deterministic CuDNN flags, batch size=512).
3. Execute experiment cells as needed:

   Final setup (2D-ROCKET): writes `/content/rocket_final_cache/final_kernel_results.csv`

   1D feature ablation: writes `/content/rocket1d_cache/feature_ablation_summary.csv`

   1D vs 2D comparison: writes `/content/rocket_dim_feature_cont/dim_feature_runs.csv`

   2D padding ablation: writes `/content/rocket_2d_padding_check_cache/padding_check_results.csv`

   Preprocessing banks: writes `/content/rocket_2d_rgb_preproc_banks_3024_cache/rgb_preproc_banks_3024_results.csv`

   Kernel config/count ablations: writes `/content/rocket_kernel_ablation_cache/kernel_ablation_results.csv`

4. The use of the recorded software versions and seed enables replication of the reported results. Figures and tables in the Results chapter can be generated directly from the CSV paths above.

Note: `get_loaders` uses `root='/content'` by default in the notebook. Some demo cells use `./data`. For reproducibility, stick to the default root or set a consistent data path.

6 Experiments

This chapter reports only *experiment-specific* settings and expected outcomes. Global assumptions (device, seeding, batch size, normalization, standardization, and classifier training) are fixed and documented in Chapter 5. Methodological rationale is summarized in Chapter 4. Each experiment is presented as a compact card that lists only the fields that deviate from the global defaults or are needed for clarification.

6.1 Card Format

For each experiment, there will be provided:

Objective (short summary of what the goal of the experiment is)
Varying configuration (concise table with parameters specific to the setup)
Procedure delta (what differs from the global workflow)
Outputs (CSV/artifacts paths)
Expected Results (brief overview of the expected results)

Items not listed in a card follow the global defaults.

6.2 Baselines

Minimal 1D-ROCKET **Objective** Provide a non-spatial ROCKET reference.
Varying configuration

1D bank	$N = 3024,\ K = [7, 9, 11],\ d\_\text{cap} = 64,\ \text{stride} = 1$
Features	ppv|max
Classifier α	grid $\{1000, \dots, 2400\}$

Outputs `/content/rocket1d_cache/rocket1d_results_pure_rocket_1d.csv`

Linear SVM **Objective** Provide a classical linear baseline on the full dataset.
Varying configuration

Pixels	flattened + StandardScaler
Dimensionality reduction	RP to 2048 components (CIFAR-10 only, optional)
Classifier	LinearSVC with $C \in \{0.1\}$, max_iter=20000

Procedure delta None. The full dataset used for training and testing.
Outputs Printed logs per dataset via `./svm_baseline_results_{dataset}.csv`.

Lightweight CNN **Objective** Provide a compact neural baseline on the full dataset.
Varying configuration

Architecture	Conv(32)-MP-Conv(64)-MP + MLP(128)
Training	epochs=10, Adam(lr=10^{-3}), CrossEntropyLoss

Procedure delta Full dataset used, no subsampling.
Outputs Printed logs via `/content/cnn_cache/cnn_results.csv`.

6.3 Feature Set

Objective Identify the best 1D feature set to inform the 2D template.
Varying configuration

Kernel bank (1D)	$N = 3024,\ K = [7, 9, 11],\ d\_\text{cap} = 64,\ \text{stride} = 1$
Variants	F1: ppv|max; F2: +mpv; F3: +mipv; F4: +lspv; F5: +mpv|mipv; F6: +mpv|lspv; F7: +mipv|lspv; F8: +mpv|mipv|lspv
Classifier α	grid $\{1000, \ldots, 2400\}$

Procedure delta Flatten images row-wise, random 1D padding.
Outputs `/content/rocket1d_cache/feature_ablation_summary.csv`
Expected Results Adding more features might lead to relevant information gain and therefore increase accuracy. However, certain feature combinations can also lead to redundancies or no significant gain in information, thus offering no improvement in prediction accuracy and only increasing computing time.

6.4 1D vs. 2D Kernels

Objective Compare 1D- versus 2D-ROCKET at comparable budgets.
Varying configuration

1D bank	$N = 3024$, K=[7,9,11], d_cap=64, features: F1 and ablation_best
2D bank	$N = 3024$, K=[7,9,11], D=$[1, 2, 3]$, padding=random, features: F1 and ablation_best

Procedure delta Follow global workflows and compare accuracy across matched α grids. The maximum dilation limit differs between 1D and 2D to account for the limitations of the effective receptive field.
Outputs `/content/rocket_dim_feature_cont/dim_feature_runs.csv`
Expected Results Accuracy may be gained from spatial structure, but it might lead to a significant increase in the computation time. However, since the 2D variant has a slightly higher feature dimensionality due to additional directional features, the results need to be evaluated in consideration of that fact.

6.5 Padding Strategy

Objective Compare *same* vs *valid* padding in 2D-ROCKET.
Varying configuration

2D bank	$N = 3024,\ K = [7, 9, 11],\ D = [1, 2, 3]$
Padding	random vs same vs valid (exclude (k, d) with eff. size $(k-1) \cdot d + 1$ exceeding image)
Features	baseline (ppv|max) and ablation_best (ppv|max|mpv|mipv_y|mipv_x)
Classifier α	grid $\{1000, \dots, 2400\}$

Procedure delta Swap padding regime, report feature dimension and accuracy.
Outputs `/content/rocket_2d_padding_check_cache/padding_check_results.csv`
Expected Results The padding strategies result in slightly different feature dimensions, as the effective feature map size differs. In the evaluation needs to be considered, that an increase might be due to a higher feature dimension and not directly due to the padding itself.

6.6 Preprocessing Banks

Objective Compare RGB-only vs. PREPROC-only vs. combined banks under equal total kernels.
Varying configuration

Total kernels	N_total = 3024
Specs	rgb_only(3024,0); preproc_only(0,3024); combo(1512,1512)
2D bank	$K = [7, 9, 11],\ D = [1]$, padding = *valid*
Preprocessing	`prev` (per-image z-score); channels=input channels
Features	ablation_best (ppv|max|mpv|mipv_y|mipv_x)
Classifier α	grid $\{1000, \dots, 2400\}$

Procedure delta By default 2D runs use loader normalization (PREPROC_NONE). Per-image z-score preprocessing (PREPROC_IMAGE_NORM) is applied only in the designated preprocessing ablation card. Extract features per bank, concatenate, standardize, grid over α.
Outputs `/content/rocket_2d_rgb_preproc_banks_3024_cache/rgb_preproc_banks_3024_results.csv`
Expected Results A combination of both RGB-only and preprocessed channels may lead to redundancy. Using only preprocessed features might help performance, but it could also leave out important details.

6.7 Kernel Configuration (Phase 1)

Objective Identify the best (K, D) per dataset at a fixed kernel count.

Kernel count	$N = 3024$
Configs	original_rocket original_rocket_dilated wide_dilated small_dilated mixed_moderate
Padding	valid
Features	ppv|max|mpv|mipv_y|mipv_x
Classifier α	grid $\{1000, \ldots, 2400\}$

Procedure delta Evaluate each config, exclude invalid (k, d) groups, and record accuracy and feature dimension.
Outputs `/content/rocket_kernel_ablation_cache/kernel_ablation_results.csv` (Phase1 rows)
Expected Results The best configuration may vary with the chosen dataset. Differences in accuracy are expected to be primarily driven by the interaction between kernel size, dilation, and the spatial structure of the dataset, with larger receptive fields benefiting more complex patterns.

6.8 Kernel Count (Phase 2)

Objective Quantify how increasing the kernel count affects both accuracy and runtime.
Varying configuration

Kernel counts	$N \in \{1000, 2000, 3024, 5000, 7000\}$
Config	chosen per dataset from Phase 1
Padding	valid
Features	ppv|max|mpv|mipv_y|mipv_x
Classifier α	grid $\{1000, \ldots, 2400\}$

Procedure delta For each N, sample kernels, extract features, standardize, and grid over α.
Outputs `/content/rocket_kernel_ablation_cache/kernel_ablation_results.csv` (Phase2 rows)
Expected Results With a higher feature dim and kernel count, the computation time might increase. It is expected that beyond a certain point, increasing the kernel count will lead to a less significant improvement in accuracy, but the computation time will increase disproportionately. Therefore, especially in this experiment, accuracy should be evaluated, including runtime costs.

6.9 Final Configuration (2D-ROCKET)

Objective Report final accuracy and per-class accuracy per dataset with the selected 2D configuration based on the test results for the given datasets.
Varying configuration

Features	ppv|max|mpv|mipv_y|mipv_x
Padding	valid
Kernel count (N)	5000
CIFAR-10 (K, D)	$K = [3, 5, 7, 9],\ D = [1, 2]$
MNIST (K, D)	$K = [3, 5, 7, 9],\ D = [1, 2]$
Classifier α	grid $\{1000, \ldots, 2400\}$

Procedure delta Apply per-image z-score, exclude invalid (k, d) groups for valid padding, extract 5 features/filter, and train Ridge at $\alpha = 1600$.
Outputs `/content/rocket_final_cache/final_kernel_results.csv`
Expected Results The final 2D-ROCKET pipeline will be evaluated primarily by test accuracy, per-class accuracy and computation time. It will be then compared directly to the defined baselines (minimal 1D-ROCKET, linear SVM, lightweight CNN). In addition to classification performance, resource metrics that matter in practice will be reported.This includes extracted feature dimensionality, classifier training time, and peak memory or estimated feature storage (GiB).

It is expected, that the 2D-ROCKET pipeline outperforms the minimal 1D-ROCKET baseline on datasets where spatial structure is informative. Furthermore, it is expected to be competitive with the CNN while offering a different trade-off: a faster classifier fitting but higher feature extraction cost. Compared to the linear SVM baseline the ROCKET features should yield substantially higher accuracy. Final conclusions will be drawn from a combined view of accuracy, per-class behaviour, and computational costs to assess practical trade-offs.

7 Results

All subsequent experimental choices are based exclusively on quantitative criteria, namely classification accuracy and training time. When appropriate, feature dimensionality and per-class accuracies for the baselines and the final pipeline are reported.

For simplicity, all subsequent experimental decisions ,with the exception of kernel configuration, were made jointly for both datasets. In cases of doubt, the decision was made in favor of CIFAR-10, as this dataset is more complex and offers more room for improvement.

Training times in the tables are defined inconsistently across experiments: some tables report only the classifier fit time (Ridge fit after feature extraction), while other tables (in particular the kernel-count ablation) report the total runtime including feature extraction. Each table caption indicates which timing is shown. Where necessary the text clarifies whether the value denotes "classifier fit time" or "total runtime (extraction + fit)". This decision is primarily based on the fact that the variable under investigation directly influences feature extraction (e.g., number/configuration of kernels). If extraction remains unchanged and only the classifier is evaluated, only the pure classifier fit time is reported.

All reported "best α" values were selected from the α grid sweep used in each experiment (see Chapter 4). For the final numbers, α was taken from the sweep using the full test split. It is important to note that this approach may be slightly optimistic.

7.1 Baselines Results

This section presents the results of the baseline runs, as well as the analysis of those results. They serve as a framework for the subsequent ablations of the ROCKET 2D adaptation and the final pipeline.

Minimal 1D-ROCKET Table 7.1 summarizes test accuracy, selected regularization strength, as well as the training time for each dataset.
The Minimal-ROCKET pipeline already achieves competitive performance on MNIST with nearly perfect accuracy. This shows that a simple random convolutional feature set captures the dominant patterns in grayscale handwritten digits when combined with PPV and Max statistics.

For CIFAR-10, performance is substantially lower. This is expected due to the increased complexity of color images and richer spatial structures, which are not fully captured by 1D flattened kernels. Nevertheless, the baseline still provides a reasonable starting point and a reproducible reference for subsequent 2D-ROCKET experiments. Training time is short for both datasets, reflecting the simplicity and efficiency of the minimal pipeline.

Dataset	**Accuracy**	**Best α**	**Train Time (s)**
MNIST	0.9858	1000	13.28
CIFAR-10	0.6258	1200	10.47

Table 7.1: Test accuracy, optimal regularization, and training time for the Minimal 1D-ROCKET baseline on MNIST and CIFAR-10.

SVM On MNIST, the SVM baseline achieves an overall accuracy of 0.9166, but performance varies between classes (some classes have substantially lower recall). On CIFAR-10, overall accuracy is 0.3790 and several classes are particularly challenging (e.g. class 3 shows low recall). Training time is substantially higher than for ROCKET and CNN due to the combination of high-dimensional inputs and slow SVM optimization.

Dataset	**Test Accuracy**	**Training Time (s)**
MNIST	0.9166	7816.6
CIFAR-10	0.3790	6890.1

Table 7.2: Test accuracy and training time for the SVM baseline.

CNN The CNN achieves near-perfect accuracy on MNIST with 0.9906. Per-class metrics indicate balanced performance across all digits. Confusion is minimal, with only a few misclassifications in digits 2, 5, and 8. On CIFAR-10, the CNN achieves a test accuracy of 0.6962. Performance varies more across classes, with the lowest F1-scores for classes 3 and 4. Confusion is more pronounced than on MNIST, reflecting the increased complexity of natural images.

Dataset	**Test Accuracy**	**Training Time (s)**
MNIST	0.9906	77.82
CIFAR-10	0.6962	79.27

Table 7.3: Test accuracy and training time for the small CNN on CIFAR-10.

7.2 Ablation Results

This section presents the results of comprehensive ablation studies designed to rigorously analyze the contribution of individual components within the 2D-ROCKET pipeline.

7.2.1 Feature Set

Dataset	**Feature Variant**	**Feature Dim**	**Best α**	**Accuracy**	**Train Time (s)**
MNIST	F1	6048	1000	0.9858	13.2
	F2	9072	1000	0.9879	20.1
	F3	9072	1000	0.9887	20.9
	F4	9072	1000	0.9882	22.7
	F5	12096	1000	0.9901	33.0
	F6	12096	1000	0.9891	32.5
	F7	12096	1000	0.9899	32.4
	F8	15120	1000	**0.9902**	44.8
CIFAR-10	F1	6048	1200	0.6258	11.0
	F2	9072	1000	0.6512	17.8
	F3	9072	2200	0.6367	19.3
	F4	9072	1800	0.6355	17.6
	F5	12096	1600	0.6568	27.8
	F6	12096	2200	0.6545	31.5
	F7	12096	2000	0.6452	29.6
	F8	15120	1800	**0.6592**	46.4

Table 7.4: Feature ablation results for MNIST and CIFAR-10. Best α, feature dimensions, accuracy, and training time are shown.

Analysis The ablation study shows the impact of different feature combinations on accuracy, feature dimensionality, and training time.

MNIST The accuracy generally increases with the inclusion of additional feature types. The second best test accuracy (0.9901) is achieved using the F5 feature set (PPV, Max, MPV, MIPV), with a corresponding feature dimension of 12096. Further extending the feature set (F8) increases the feature dimension to 15,120 and training time. At the same time, F8 improves accuracy compared to F5.

CIFAR-10 Accuracy improves as more feature types are combined. The best accuracy (0.6592) is reached with the full feature set F8 (PPV, Max, MPV, MIPV, LSPV) with 15120 features. Training time increases with feature dimension, showing a trade-off between computational cost and feature richness.
The full feature set F8 achieves the highest accuracy on CIFAR-10. However, the F5 configuration provides competitive performance across both datasets, using substantially fewer features and requiring less training time.

F5 is used for subsequent experiments because it provides a favorable accuracy vs. cost trade-off. It achieves near-top accuracy while substantially reducing feature dimensionality and training time compared to the full feature set F8. This makes F5 a pragmatic choice for systematic ablations under realistic compute budgets.

7.2.2 1D vs. 2D Kernels

Dataset	**Feature set**	**Kernel dim.**	**Accuracy**	**Feature dim.**	**best α**	**Train time (s)**
MNIST	ablation_best	1D	0.9901	12096	1000	36.2
MNIST	ablation_best	2D	**0.9914**	15120	2000	43.4
CIFAR-10	ablation_best	1D	0.6568	12096	1600	28.1
CIFAR-10	ablation_best	2D	**0.6680**	15120	1600	38.9

Table 7.5: Performance of the best ablation feature configurations for 1D- and 2D-ROCKET on MNIST and CIFAR-10.

Analysis Across both feature configurations, 2D kernels consistently outperform their 1D counterparts. The performance gain is modest: on MNIST the gain is about 0.13 percentage points (0.9901 to 0.9914) and on CIFAR-10 about 1.12 percentage points (0.6568 to 0.6680). Notably, 2D kernels achieve higher accuracy with increased feature dimensionality. The 1D ablation-best setup requires 12,096 features, the 2D variant uses 15,120 features. Feature extraction and training times are higher for 2D in both cases.

Subsequent experiments therefore focus on 2D kernels, since they consistently yield higher accuracy (in particular a clear benefit on CIFAR-10). This allows the following ablations to target spatial design choices.

7.2.3 Padding Strategy

Table 7.6 summarizes the best test accuracy, resulting feature dimensionality, selected regularization strength, and training time for each configuration, as observed in the actual experiments.

Dataset	**Padding**	**Accuracy**	**Feature Dim.**	**Best α**	**Train Time (s)**
CIFAR-10	random	0.6679	15,120	1600	37.8
CIFAR-10	same	0.6699	15,120	1200	37.4
CIFAR-10	valid	**0.6715**	15,120	1800	37.8
MNIST	random	0.9914	15,120	2000	44.0
MNIST	same	**0.9920**	15,120	1400	44.8
MNIST	valid	0.9919	13,385	1000	37.1

Table 7.6: Effect of 2D padding strategy on CIFAR-10 and MNIST. Values correspond to the best-performing regularization strength per dataset.

Analysis For CIFAR-10, valid padding slightly outperforms same padding in terms of test accuracy (0.6715 vs. 0.6699), with feature dimensionality and training time remaining approximately equal.

For MNIST, valid padding yields a slightly lower accuracy compared to same (0.9919 vs. 0.9920), but decreases feature dimensionality and training time significantly.

Both valid and same padding outperform the random approach in terms of accuracy. Given that MNIST is already near-saturation, the minor difference between valid and same is negligible. On the more challenging CIFAR-10 dataset, valid padding provides the highest accuracy without increasing computational cost.

Valid padding is selected for the remainder of the study, since it gives the best trade-off on CIFAR-10. Additionally, it reduces feature dimensionality for MNIST and with that the required computation time.

7.2.4 Preprocessing Banks

Table 7.7 summarizes the best test accuracy, resulting feature dimensionality, selected regularization strength, and training time for each configuration.

MNIST				
Input Configuration	Acc.	Dim.	Best α	Time (s)
Gray-only	**0.9919**	13,385	1000	36.88
Preprocessing-only	0.9917	13,385	1000	37.63
Gray + Preprocessing	0.9900	13,470	1000	37.78

CIFAR-10				
Input Configuration	Acc.	Dim.	Best α	Time (s)
RGB-only	**0.6715**	15,120	1800	38.67
Preprocessing-only	0.6531	15,120	2000	39.08
RGB + Preprocessing	0.6233	15,120	1800	38.93

Table 7.7: Comparison of RGB and preprocessing-based input representations using 2D-ROCKET kernels with `valid` padding. Values taken from experiment logs (best α per run).

Analysis For MNIST, the `gray_only` configuration achieves the highest accuracy (0.9919) with a feature dimension of 13 385 and a runtime of 36.88 s. `Preproc_only` follows with an accuracy of 0.9917, 13 385 features and a runtime of 37.63 s. `Mixed_half` performs the worst with an accuracy of 0.9900, while increasing the features to 13 470. For CIFAR-10, the `RGB-only` variant attains the highest accuracy of 0.6715.
The classifier fit times for the best α in each run are very similar across these variants (on the order of approximately 37 s), so timing differences do not drive the selection.
MNIST is near saturation, while CIFAR-10 is more informative for spatial/color features. Therefore, subsequent experiments use the RGB/Grey-only input.

7.2.5 Kernel Configuration and Kernel Count Ablation

Configuration	**Kernel Sizes**	**Dilations**	**Accuracy**	**Feature Dim.**
Original ROCKET	[7, 9, 11]	[1]	0.9927	15,120
Mixed, moderate dilation	[3, 5, 7, 9]	[1, 2]	**0.9934**	15,120
Small, progressive dilation	[1, 3, 5]	[1, 2, 3, 4]	0.9920	15,120
Original ROCKET (dilated)	[7, 9, 11]	[1, 2, 4]	0.9916	11,545
Current best	[1, 3, 7, 12]	[1, 2, 3]	0.9913	13,790

Table 7.8: Kernel configuration ablation results on MNIST using 2D-ROCKET kernels (Phase 1). Best accuracy per configuration is reported from the logs.

Configuration	**Kernel Sizes**	**Dilations**	**Accuracy**	**Feature Dim.**
Original ROCKET	[7, 9, 11]	[1]	0.6991	15,120
Original ROCKET (dilated)	[7, 9, 11]	[1, 2, 4]	0.6750	11,835
Small, progressive dilation	[1, 3, 5]	[1, 2, 3, 4]	0.7027	15,120
Wide, dilated	[1, 3, 7, 12]	[1, 2, 3]	0.6989	13,705
Mixed, moderate dilation	[3, 5, 7, 9]	[1, 2]	**0.7189**	15,120

Table 7.9: Kernel configuration ablation results on CIFAR-10 using 2D-ROCKET kernels (Phase 1). Best accuracy per configuration is reported from the logs.

Analysis On MNIST, the mixed-moderate configuration ([3,5,7,9], dilations [1,2]) achieved the highest accuracy in the Phase 1 runs (0.9934). The original ROCKET configuration ([7,9,11], dilation 1) reached 0.9927, with other configurations in the 0.9913–0.9934 range. Feature dimensionalities remain comparable across most variants. The mixed-moderate design is therefore adopted for Phase 2 to evaluate scaling with kernel count.
For CIFAR-10, the mixed-size configuration with moderate dilation ([3, 5, 7, 9], dilation [1, 2]) also achieves the highest accuracy (0.7189). Other configurations perform worse, with accuracies ranging from 0.6750 to 0.7027. This indicates that using a mix of kernel sizes combined with moderate dilation is particularly beneficial for CIFAR-10.
The mixed-moderate kernel configuration is carried forward to Phase 2 because Phase 1 showed it to be the most effective design for CIFAR-10. This allows Phase 2 to measure scaling effects under a practically useful configuration.

Phase 2: Kernel Count Ablation In the second phase, the best-performing kernel configuration from Phase 1 (mixed sizes with moderate dilation) is evaluated across different kernel counts to assess scalability and performance saturation.

Analysis Increasing the number of kernels improves classification accuracy on both MNIST and CIFAR-10. However, gains diminish while computational costs increase significantly.

For CIFAR-10, the highest accuracy reported in the Phase 2 runs is 0.7372 at 7,000 kernels; the 5,000-kernel configuration yields 0.7342 with substantially fewer features and lower training time.

On MNIST, after 3,000 to 5,000 kernels, accuracy gains are minimal compared to the additional computation time.
These results show that kernel configuration strongly impacts performance. Mixed kernel sizes with moderate dilation consistently outperform both large undilated kernels and heavily dilated small kernels. Performance scales predictably with the number of kernels. Marginal gains beyond roughly 5,000 kernels come at a disproportionately higher computational cost.

As kernel count increases accuracy with diminishing returns, $N = 5000$ is selected for the final pipeline as a pragmatic compromise. It attains near-peak accuracy within the tested number of kernels, while keeping feature dimensionality and total runtime at reasonable levels.

Number of Kernels	**Accuracy**	**Feature Dim.**	**Training Time (s)**
1,000	0.9899	5,000	10.0
2,000	0.9931	10,000	23.5
3,024	0.9934	15,120	44.4
5,000	0.9941	25,000	102.9
7,000	**0.9951**	35,000	198.5

Table 7.10: Kernel count ablation study on MNIST using the mixed-moderate configuration. Training time denotes classifier fit time measured for the best α observed in the sweep (notebook logs).

Number of Kernels	**Accuracy**	**Feature Dim.**	**Training Time (s)**
1,000	0.6683	5,000	9.3
2,000	0.7011	10,000	20.9
3,024	0.7189	15,120	37.2
5,000	0.7342	25,000	89.2
7,000	**0.7372**	35,000	164.3

Table 7.11: Kernel count ablation study on CIFAR-10 using the mixed-moderate configuration. Training time denotes classifier fit time for the best α observed in the sweep (notebook logs).

7.2.6 Final Configuration (2D-ROCKET)

Setup The final 2D-ROCKET pipeline used the mixed-moderate kernel configuration selected in Phase 1 (kernel sizes $K = [3, 5, 7, 9]$, dilations $D = [1, 2]$) with a total kernel budget of $N = 5000$. Feature types extracted were PPV, Max, MPV, and MIPV. Padding was set to valid.

Feature extraction and storage Features were extracted once and cached prior to classifier training. The final train feature matrices had dimensionalities:

MNIST:	$F = 25{,}000$ features per image (train shape: $(60000, 25000)$), estimated storage $\approx 5.59\,\text{GiB}$ for single-precision arrays.
CIFAR-10:	$F = 25{,}000$ features per image (train shape: $(50000, 25000)$), estimated storage $\approx 4.66\,\text{GiB}$ for single-precision arrays.

The kernels generated for the final runs are reproducible via the recorded seed. The notebook printed the kernel hashes and counts for traceability.

Classifier training and reported metrics For each dataset the predefined α grid is swept and report below the best α. The corresponding test accuracy, and the measured classifier fit time (measured after feature extraction):

MNIST:	best $\alpha = 1200$, test accuracy $= 0.9941$, classifier fit time is approximately $103.9\,\text{s}$.
CIFAR-10:	best $\alpha = 2400$, test accuracy $= 0.7343$, classifier fit time is approximately $90.2\,\text{s}$.

Per-class accuracies and confusion matrices for these final runs are saved and plotted (see 7.4).

Outputs and provenance All final run artifacts (per-alpha rows, per-class metrics, and caches) were written to the results directory and can be used to reproduce these numbers:
Final results CSV: `/content/rocket_final_cache/final_kernel_results.csv`
Feature caches and extraction artifacts: the cache subfolders under `/content/` used by the notebook (see Experiment cards in Chapter 6 for exact paths).

7.3 Theoretical Computational Characteristics

Minimal 1D-ROCKET Adaption The overall runtime of the minimal 1D-ROCKET pipeline is dominated by feature extraction and grows linearly with the number of samples N and the number of kernels K with

$$O(N \cdot K \cdot L)$$

where L is the input sequence length. Training with RidgeClassifier adds a smaller term. The exact cost depends on the training algorithm used.

SVM After random projection, the main computational costs are:

Random projection: $O(N \cdot d \cdot \tilde{d})$, where d is the original feature dimension and $\tilde{d}$ the projected dimension.

Linear SVM training: $O(N \cdot \tilde{d} \cdot \text{iter})$, where iter is the number of optimization iterations.

In total, runtime scales approximately as $O(N \cdot \tilde{d} \cdot \text{iter})$ for large datasets and moderate projected dimensions.

Lightweight CNN The training runtime per epoch is dominated by convolutional layers and is proportional to the number of samples N and image size $H \times W$ with

$$O(\text{epochs} \cdot N \cdot H \cdot W \cdot C)$$

where C is the number of channels. Per-batch complexity depends on the network architecture, but overall scaling with data is linear in N.

Final Pipeline The increased number of kernels leads to a higher feature dimensionality and training time. Training the ridge classifier required approximately 103.9 seconds for the MNIST final run and around 90.2 seconds for the CIFAR-10 final run. For large feature spaces, numerical conditioning warnings may occur during optimization; these were observed in some runs but did not invalidate the reported predictive results.

The final pipeline has two main components: 2D convolution for feature extraction and Ridge classifier training. Let:

N	number of training samples
K	total number of convolution kernels
C	number of image channels
k	(maximal) kernel size
H, W	height and width of the input images
H', W'	height and width of the output feature map after convolution
F	total number of features per image ($F = K \cdot T$)

where T is the number of feature types extracted per kernel.

Feature Extraction For each input image, every kernel is applied via a 2D convolution. The computational cost depends on the number of output positions ($H' \times W'$) and the operations per position ($C \cdot k^2$) resulting in

$$\mathcal{O}\left(N \cdot K \cdot C \cdot k^2 \cdot H' \cdot W'\right).$$

Larger kernels increase operations per output location, determining the worst-case complexity.

Ridge Classifier Training For N samples with F features each, training a linear Ridge classifier using direct solvers may cost:

$$\mathcal{O}\left(N \cdot F^2 + F^3\right),$$

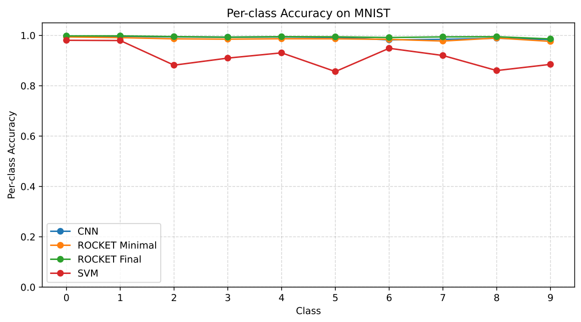
while iterative solvers can have different scaling depending on iteration counts and stopping criteria.

Total Time Complexity This leads to an overall time complexity of

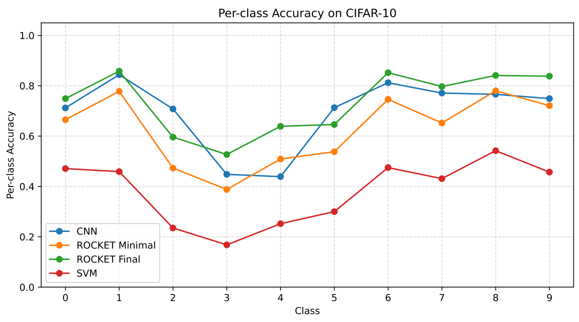
$$\mathcal{O}\left(N \cdot K \cdot C \cdot k^2 \cdot H' \cdot W'\right) + \mathcal{O}\left(N \cdot F^2\right).$$

Batch size affects implementation efficiency but not asymptotic complexity.

7.4 Per-class Performance



(a) MNIST per-class accuracy



(b) CIFAR-10 per-class accuracy

Figure 7.1: Per-class accuracy comparison for different models on MNIST and CIFAR-10.

Analysis The per-class accuracy highlights the strengths and weaknesses of the evaluated models on MNIST and CIFAR-10.

For MNIST, all models achieve overall high accuracies. The ROCKET final pipeline model attains the highest results, with an average per-class accuracy above 0.99. CNN performs slightly lower (around 0.9906), while classical SVM shows notably weaker performance on classes 2, 5, and 7. The Minimal-ROCKET model also performs very well and is almost on par with the final ROCKET pipeline, with only minimal differences in a few classes.

On CIFAR-10, the differences between models are more pronounced. The ROCKET final pipeline achieves the highest per-class accuracies in the logged runs. The Minimal-ROCKET adaption performs worse than the final pipeline, while SVM performs significantly worse on several classes. This highlights that CIFAR-10 has higher visual complexity and more color variation.

7.5 Main Results

The final results show that ROCKET, even in its minimal 1D version for MNIST, already achieves high accuracy. Compared to the minimal 1D-ROCKET baseline on MNIST (0.9858), the final pipeline increases accuracy by 0.83 percentage points (to 0.9941).

For CIFAR-10, comparing the best 1D ablation result (0.6568) to the final 2D pipeline (0.7343) gives an increase of 7.75 percentage points. These comparisons use the reported runs shown in the tables above and clarify which baseline is used for each dataset.

Method	Dataset	Test accuracy	Training Time (s)
Minimal-ROCKET	MNIST	0.9858	13.28
ROCKET-final	MNIST	0.9941	103.9
Small CNN	MNIST	0.9906	77.83
Linear SVM	MNIST	0.9166	7816.6
Minimal-ROCKET	CIFAR-10	0.6258	10.47
ROCKET-final	CIFAR-10	0.7343	90.2
Small CNN	CIFAR-10	0.6962	79.28
Linear SVM	CIFAR-10	0.3790	6890.1

Table 7.12: Main results summary. "Training Time" denotes classifier fit time unless stated otherwise; where a table reports total runtime (extraction + fit) this is explicitly noted in the caption or surrounding text.

While the minimal 1D-ROCKET pipeline is relatively fast at 10-13 seconds, especially compared to the other benchmarks, the actual computing time is increased several times by the adjustments.

Compared to the CNN, which is only about 3.8 percentage points below the final ROCKET pipeline on CIFAR-10, ROCKET requires slightly more computation but achieves higher accuracy. The SVM stands out due to its long computation time and its relatively poor accuracy.

7.6 Answers to Ablation Research Questions

Below are experiment-level answers to the research questions stated in the Methodology for each ablation run. Each answer is tied to the corresponding result table or section.

1D Feature Ablation "Do additional summary statistics (MPV, MIPV, LSPV) improve performance over the original PPV+Max features on flattened inputs?"
Yes, adding local statistics increases accuracy but with diminishing returns and higher computation cost. On MNIST, the full set (F8) produces the top accuracy (0.9902) while F5 already attains near-top performance (0.9901) with substantially fewer features. On CIFAR-10 the full set F8 again gives the best accuracy (0.6592) but at a large increase in dimensionality (see Table 7.4).
Limitation: improvements beyond the core PPV+Max are mostly incremental and incur a clear feature/time penalty.

1D vs. 2D Kernels "Does preserving spatial structure with 2D convolutions outperform flattened 1D kernels at comparable budgets?"
Yes, for the ablation-best feature set, 2D kernels consistently yield higher accuracy than 1D at comparable kernel budgets (MNIST: 0.9901 to 0.9914; CIFAR-10: 0.6568 to 0.6680), see Table 7.5.
Limitation: the gain is small on near-saturated datasets (MNIST) and comes with higher extraction and fit times.

2D Padding Ablation "How do padding strategies (same, valid, random) affect accuracy and feature dimensionality?"
Padding mainly controls feature dimensionality with only small effects on accuracy for the tested datasets. For CIFAR-10, valid produced the highest accuracy (0.6715) at the same feature dimension as same/random. For MNIST valid reduced the feature count while preserving near-top accuracy (see Table 7.6).
Limitation: padding mainly filters large receptive-field activations. On other datasets boundary features might matter more.

Preprocessing Banks "Do kernels on preprocessed channels outperform kernels on dataset-normalized raw inputs, and is concatenation additive?"
The outcome is dataset-dependent. For MNIST the tested preprocessing gives marginal differences
(gray_only 0.9919 vs preproc_only 0.9917). For CIFAR-10 the tested preprocessing bank worsened accuracy relative to RGB-only (0.6531 vs 0.6715). Concatenating banks produced mixed results and often increased feature cost without consistent gains (see Table 7.7).
Limitation: the evaluated preprocessing family is limited, since they refer to per-image/per-channel normalizations (mean/std) and not to filter/kernel operations. Alternative preproc designs may behave differently.

Kernel Configuration "Which kernel size/dilation combinations perform best under a fixed kernel count?"
Mixed multi-scale configurations with moderate dilation performed best in Phase 1. The mixed_moderate design ([3,5,7,9] with dilations [1,2]) yielded the highest MNIST and CIFAR-10 Phase-1 accuracies and was therefore selected for Phase 2 (7.9).
Limitation: the optimality of a configuration depends on image size and dataset structure. The mixed strategy is a robust practical choice but not universally optimal.

Kernel Count Scaling "How does performance scale with the number of kernels?"
Performance improves with kernel count but exhibits diminishing returns. The accuracy increases substantially moving from 1k to 3k kernels and flattens as K grows further, while feature size and training time increase roughly linearly (see Tables 7.10 and 7.11).
Practical choice: $N = 5000$ was selected as a pragmatic compromise between near-peak accuracy and resource cost primarely optimized for CIFAR-10.
Limitation: very large K can cause numerical conditioning issues and heavy memory use.

Final Evaluation "What accuracy does the final 2D ROCKET configuration achieve on each dataset and how does it compare to baselines?"
The final 2D-ROCKET pipeline (mixed_moderate, $N = 5000$, valid padding, dataset appropriate input bank) achieved 0.9941 on MNIST and 0.7343 on CIFAR-10. These results are reported in Table 7.12. Compared to the simple CNN and linear SVM baselines used here, 2D-ROCKET is competitive on MNIST and narrows the gap to the CNN on CIFAR-10 while remaining computationally feasible.
Limitation: these comparisons reflect the chosen baseline complexity and single-seed runs. More optimized baselines or multi-seed stats could affect relative rankings.

Each answer above is intentionally concise and directly tied to the ablation results. Where effect sizes are small (especially on MNIST) multi-seed verification is recommended before claiming robust superiority.

8 Discussion

The purpose of this discussion is not to revisit individual design choices made throughout the experimental pipeline. These are already motivated and justified quantitatively in 7. Instead, this chapter aims to contextualize the observed effects. It provides broader insights into how architectural and representational design choices influence the behavior of ROCKET-style models in two-dimensional image settings. Therefore, focus lies on underlying mechanisms, design implications, and methodological trade-offs rather than on specific parameter values.

8.1 Feature Statistics and Random Representations

Across all experiments, the effectiveness of the 2D-ROCKET pipeline is influenced by the choice of summary statistics applied to random convolutional responses. The pair PPV + Max (the original ROCKET configuration) already captures much of the discriminative signal. For example, in the ablation runs, this baseline reaches 0.9858 on MNIST (feature dimension 6,048) and 0.6258 on CIFAR-10 (feature dimension 6,048). Supplementing the core set with additional local statistics leads to smaller incremental accuracy gains, but substantially increases feature dimensionality and computation.

These findings indicate that representational power in the random convolutional pipeline can be captured to a large extent by a small, targeted set of local features. The core statistics appear to capture most of the informative variation observed in these experiments. Additional descriptors appear to exploit residual information that the base set misses. However, each extension increases the feature budget and runtime. Therefore, marginal accuracy gains diminish while feature budget and runtime increase. It remains an open question where the accuracy plateau occurs. In addition, it remains unclear whether alternative, possibly lower-dimensional, summary statistics could achieve similar gains.

Importantly, the reported results were obtained without domain-specific feature engineering. Combining a handful of simple, interpretable local statistics produced strong classification baselines with minimal implementation effort. This underlines the practical strength of the random feature approach. Robust baselines are accessible with straightforward, generic statistics and limited tuning.

A natural limitation is that only a particular collection of summary statistics and standard datasets were tested here. Generalization to other statistics, datasets with different scale or texture properties, or industrial image domains remains an empirical question. Also, note the dataset bias: experimental choices and tuning prioritized CIFAR-10, the more challenging and natural-image benchmark, whenever a single setting had to be chosen for both datasets. As a consequence, the incremental accuracy gains recorded for MNIST when adding further statistics are very small. For MNIST a cheaper feature combination (e.g. PPV+Max) would often have sufficed. Thus, the accuracy–cost conclusions drawn above primarily concern complex datasets such as CIFAR-10 and should be interpreted with caution for MNIST.

8.1.1 Reproducibility and Experimental Variance

All reported results in this work were obtained with fixed seeds to ensure traceability of kernels and artifacts. Assessing stability across multiple independent seeds (reporting mean $\pm$ standard deviation for key runs) is left for future work and would clarify how robust the observed effects are to kernel sampling and random initialization. In addition, reproducible GPU execution required deterministic settings and appropriate environment variables on the used hardware. Fully deterministic behaviour across all CUDA operations may require additional platform-specific configuration.

8.2 Kernels, Scale, and Spatial Structure

Moving from 1D kernels (applied to flattened images) to full 2D kernels is a conceptual shift: 1D kernels can capture intensity patterns but discard spatial locality, whereas 2D kernels preserve local spatial relations. The experimental results show that 2D kernels systematically improve performance, most notably on datasets with higher intra-class variability and richer spatial content (CIFAR-10). In such cases, random 2D kernels respond to localized patterns and thereby approximate aspects of spatial feature extraction typical of trained convolutional models without learned filters. Overall, the observed results indicate that feature selection in random pipelines is primarily a budgeting problem rather than a representational one.

This improvement is not achieved by mere kernel count alone. Kernel configuration (the choice of kernel sizes and dilation patterns) noticeably affects test accuracy. Configurations that provide balanced coverage of spatial scales consistently outperform other strategies. These strategies include using only large undilated kernels or kernels with excessive dilation. Excessive dilation or a narrow size range reduces effective spatial coverage and harms accuracy. As the kernel count increases, accuracy quickly approaches a plateau. Further increases mainly inflate feature dimensionality and training time while providing only marginal accuracy improvements.

From a methodological viewpoint, these results suggest that scale-diverse kernels with moderate dilation should be prioritised scale-diverse, moderate dilation patterns over blindly increasing kernel counts. In practice, that means allocating kernel budget to ensure multi-scale coverage rather than maximizing K without structure.

8.3 Padding Strategy

Padding controls the set of spatial positions at which kernels produce responses, hence affecting the number of extracted features especially for large/dilated kernels. Across

the experiments, using more restrictive padding reduced feature dimensionality and computational cost. Accuracy remained effectively unchanged for the tested datasets. This indicates that many features produced by kernels with very large receptive fields are redundant for MNIST and CIFAR-10.
Consequently, restrictive padding (e.g., valid) can be used as a practical lever to reduce computation and memory without measurable loss in predictive quality in the settings tested here. In other words, padding acts as a simple, implementation-level regularizer of the feature bank. It removes many boundary/large-receptive-field activations that contribute little to downstream discrimination for these datasets.

8.4 Preprocessing and Input Representations

By default, the dataset loader applies dataset-wide normalization via `transforms.Normalize(MEAN,STD)`. In addition, optional per-image/per-channel normalization (`PREPROC_IMAGE_NORM`) can be performed—for each image, the image mean is subtracted per channel and divided by the image standard deviation. This is pure normalization (no filter/kernel operation such as a Sobel filter). The implementation includes a heuristic that skips per-image normalization if the batch statistics are already close to 0 or 1.

For MNIST, preprocessing banks deliver at most marginal improvements compared to the grayscale input. In the logged runs the per-image preprocessing variant achieves 0.9917 while the gray-only run reaches approximately 0.9919 (differences are negligible). Feature dimensionality is effectively unchanged for the two single-representation runs (13,385 features), and the classifier fit time is slightly higher with per-image normalization approximately 37.6 s vs. approximately 36.9 s), so any accuracy gain comes at negligible or slightly increased runtime cost.

For CIFAR-10, the preprocessing banks tested worsen performance compared to the plain RGB input (approximately 0.6531 vs. approximately 0.6715). Feature dimensionality is the same across these variants (15,120), and fit times are similar (approximately 39.1 s vs. approximately 38.7 s). This suggests that the particular per-image normalization and preprocessing choices used here remove inter-image color/intensity cues that are informative for CIFAR-10, while they are of limited harm or slight benefit on MNIST.

This difference can be attributed to the nature of per-image versus dataset-wise normalization. Per-image normalization removes global intensity and color statistics. On one hand, this can be beneficial for simple, monochrome datasets. On the other hand, it can be slightly detrimental for complex, colored datasets, where inter-image color distribution carries informative patterns. This indicates that preprocessing choices should be adapted to the dataset. Also, uniform application may not be optimal.

This emphasizes the role of simple input transformations, like normalization, in non-learned random architectures. More advanced preprocessing or combining multiple input representations were not explored and may be considered in future work.

8.5 Generalization Across Dataset Complexity

The final pipeline results confirm that ROCKET-style models can be applied successfully to image classification tasks of varying difficulty, as long as spatial structure is incorporated. On MNIST, accuracy quickly plateaued near 99.4%, even with modest random feature sets. This indicates that simple visual structures can be captured efficiently. For CIFAR-10, it was observed that increasing the number and diversity of kernels was required to approach the best accuracy (up to 73.1%).

Note that the final pipeline choices were optimised with CIFAR-10 in mind. Consequently, the MNIST runs could have been obtained more cheaply (lower K or fewer feature types) had MNIST been the sole optimisation target. Practical choices were therefore biased toward configurations that scale to CIFAR-10, rather than to the minimal cost for MNIST.

Per-class analysis shows substantial variation in classification difficulty. Some classes are harder to separate using random features alone, suggesting that class-specific features or adaptive weighting could provide further gains. Importantly, the implementation remained stable and practical, even when the feature dimensionality reached 35,000 and beyond. This demonstrates that 2D-ROCKET handled tested representational settings without excessive computational overhead.

Nevertheless, when initial accuracy is already high, as in MNIST (approximately 98%), it becomes important to carefully evaluate whether additional extensions justify the increased computation time. Using the more complex final pipeline on CIFAR-10 led to a significant increase in accuracy, which puts the increase in computation time into perspective. For MNIST, this assessment is not trivial, as the accuracy improvement is minimal compared to a disproportionate increase in computation time. This also suggests that, despite ROCKET's potential for generalization, there is still room for the individual design of a data-specific pipeline.

Statistical significance and effect size Small absolute accuracy gains (sub-percent) observed on near-saturated datasets should be interpreted cautiously. In the case of minor differences (e.g., MNIST), replication runs with several independent seeds and a statistical evaluation (mean $\pm$ standard deviation across repetitions) are recommended before the effect can be considered robust.

Failure modes and per-class insights Per-class confusion patterns reveal concrete failure modes (for example, classes with large intra-class variability on CIFAR-10). These patterns suggest concrete next steps such as class-aware kernels, targeted augmentations, or light supervised adaptation to address class-specific deficiencies.

8.6 Comparison with Baseline Models

The comparison with classical baselines provides insight into the relative strengths and limitations of the 2D-ROCKET approach. Linear SVMs rely heavily on the quality of handcrafted or precomputed features and lack inherent mechanisms for capturing spatial hierarchies. By contrast, SCNNs benefit from learned convolutional filters and hierarchical feature composition, particularly on complex datasets.

When evaluating the results, it is important to consider that the baseline models were deliberately kept simple. For example, the SVM was applied only to raw features, without HOG or other advanced preprocessing, and the CNN was kept flat. This maintains a complexity comparable to the 2D-ROCKET pipeline but means that higher accuracy would be possible with more highly optimized baselines. At the same time, this experimental framework allows for a fair assessment of the efficiency and predictive power of 2D-ROCKET relative to models of similar complexity.

2D-ROCKET occupies an intermediate position. It does not learn task-specific filters. However, its diverse random convolutions can approximate certain spatial feature extraction properties of CNNs, while requiring much lower training complexity. The competitive performance observed on MNIST and the narrowing performance gap on CIFAR-10 illustrate that carefully designed random feature pipelines can serve as strong baselines, especially in scenarios where training efficiency or limited data are primary concerns.

A ridge classifier was chosen for the experiments because of its simplicity, closed-form regularization behaviour and numerical stability. The regularization strength α was selected via the grid sweep reported in the Results. Alternative linear solvers, iterative methods, or dimensionality reduction prior to classification can change runtime and conditioning trade-offs. These avenues are worth exploring in future work.

Theoretical time complexity and empirical scaling To avoid notation ambiguity, use the following variables: let S denote number of training samples, K the total number of kernels, C the number of input channels, k the (maximal) kernel size, H,W the input height and width, and F the final feature dimension per sample. Feature extraction costs for 2D-ROCKET scale roughly as

$$\mathcal{O}\big(S \cdot K \cdot C \cdot k^2 \cdot H' \cdot W'\big),$$

where $H' \times W'$ is the spatial output size per kernel (depending on padding/dilation). Training a linear ridge model on the extracted features can cost on the order of

$$\mathcal{O}\big(S \cdot F^2\big)$$

for direct or naive solvers (iterative methods scale differently). These expressions match the observed empirical behaviour: doubling kernel count and feature types increases both extraction time and the classifier fit time, and very large feature dimensions (tens of thousands) substantially increase training time.

Empirically, the pipeline remained practical on the tested hardware: feature extraction and training completed in acceptable timeframes even at high K and F, and the method was faster / more stable than the high-dimensional SVM baseline in the logged runs.

8.7 Practical Takeaways and Limitations

From a practical perspective, 2D-ROCKET is well suited for rapid prototyping, small- to medium-scale datasets, and contexts requiring transparent feature mappings. The method

provides a reproducible, deterministic feature bank that is simple to audit and analyze. Feature storage and peak memory are practical on modern hosts. However, for larger experiments, they may push resource limits. Possible countermeasures include streaming extraction, on-disk memory maps, reduced numeric precision, or early dimensionality reduction before classifier training.
Numerical conditioning warnings were observed in some large-F runs during the ridge solves. This motivates the use of stronger regularization, iterative solvers with preconditioning, or dimensionality reduction strategies when pushing to extreme feature budgets.

Reproducibility required deterministic settings on GPU and careful environment configuration. Users replicating these experiments should document environment variables and package versions to ensure consistent runs.

Limitations include the nonadaptive nature of random kernels (no dataset-specific filter learning) and the potentially large memory footprint at very high kernel counts. The method therefore trades some ultimate accuracy for efficiency, transparency, and simplicity. Hybrid designs (semi-adaptive kernels or light fine-tuning) and domain-aware preprocessing are promising extensions to push the approach further.

8.8 Summary

In conclusion, the 2D-ROCKET pipeline demonstrates that lightweight random feature approaches can effectively capture spatial structures in image data, bridging the gap between simple baselines and shallow CNNs. Careful selection of kernel configurations, feature statistics, and preprocessing strategies allows strong classification performance. At the same time, computational efficiency can be retained. The experiments reported here highlight practical trade-offs and offer a baseline configuration that appears to balance predictive power, feature budget, and runtime effectively. They indicate specific, actionable directions for future improvement.

9 Conclusion

In this work, the ROCKET framework was adapted from one-dimensional time series to two-dimensional image data, demonstrating the potential of untrained kernels and simple aggregation features for image classification. The main conclusions can be summarized as follows:

2D adaptation as the core enabler Preserving spatial structure and cross-channel correlations through 2D convolutions proved essential for capturing meaningful image patterns. This design choice alone distinguishes 2D-ROCKET from flattened or sequentialized alternatives, allowing expressive features without iterative training.

Feature statistics determine effectiveness Aggregation statistics emphasizing local extrema, especially Max and PPV, accounted for the majority of the performance gain. Adding further statistics, such as MPV, MIPV, and LSPV resulted in diminishing returns overall. Depending on the selected combination, small but measurable improvements were observed.

Kernel design and coverage matter Classification accuracy depends on kernel size, dilation, and count. Moderate dilation and mixed kernel sizes yield robust representations, whereas extreme or insufficient configurations degrade performance. Saturation effects illustrate the trade-off between accuracy and computational cost.

Interpretability and transparency Unlike CNNs, 2D-ROCKET does not rely on learned hierarchical filters, which makes the individual kernels and aggregation statistics explicitly defined. This structure facilitates a more transparent analysis of feature contributions and can be advantageous in settings where model transparency is important.

Computational efficiency Although theoretical computational cost increases with the number of kernels and extracted features, runtimes in the experiments remained practical. The final pipeline required substantial classifier fit time after extraction, while linear SVMs took considerably longer on high-dimensional feature sets. CNN training benefited from GPU acceleration and achieved short wall-clock times for the small model. These experiments also highlight that practical runtime depends on implementation choices and how it is measured (classifier fit time vs. total runtime).

Dataset-specific budgeting The experiments show that configurations tuned for a more challenging dataset (CIFAR-10) can be over-provisioned for simpler tasks (MNIST). In other words, the final MNIST numbers could have been obtained with a cheaper configuration (fewer kernels / feature types) had MNIST been the sole optimisation target. Settings were deliberately prioritised that scale to CIFAR-10. This trade-off exemplifies the practical need to budget kernels and feature types per dataset.

9.1 Strengths and Limitations

A critical assessment of the proposed 2D-ROCKET approach reveals several key strengths and inherent limitations, which are outlined in Table 9.1. The table below summarizes the main strengths and limitations of the 2D-ROCKET approach, based on the insights from Chapter 8.

Table 9.1: Summary of main strengths and limitations of the 2D-ROCKET approach.

Strengths	**Limitations**
Light to moderate computational and memory footprint, enabling rapid feature extraction and prototyping on common hardware.	High feature dimensionality can increase memory usage at scale and require mitigation (streaming, mmap, lower precision).
Preserves spatial image structure via 2D convolutions, yielding expressive features without iterative learning.	Lacks adaptive, task-specific feature learning. Feature quality depends on random sampling and budgeting.
Contribution of each kernel and statistic can be inspected independently, supporting transparency.	Performance might saturate if the task requires specialized, learned hierarchical features.
Performs well on moderate-scale and low-data regimes, often outperforming simple SVM baselines.	Fixed random kernels may underperform compared to well-tuned deep CNNs on large, highly variable datasets.
Simple, reproducible, and easy to implement. Serves as a robust, well-defined baseline.	Theoretical complexity grows with kernel and feature count. Resource constraints (memory/conditioning) become limiting without additional methods.

9.2 Assessment Against Research Questions

The experiments confirm that 2D-ROCKET addresses the research objectives in the ablation programme:

Performance relative to baselines: On MNIST, near-saturated accuracy comparable to a small CNN was achieved (0.9941), showing efficient capture of dominant visual patterns. On CIFAR-10, the final 2D pipeline reduced the gap to the small CNN and outperformed the linear SVM baseline (0.7343). See Table 7.12 for details.

Influence of design choices: Kernel configuration, padding, and feature selection significantly impacted both accuracy and efficiency. Multi-scale kernel coverage with moderate dilation and a compact core of aggregation statistics provided the most favourable trade-offs (see the Ablation Results).

Practical advantages, limitations, and use cases: 2D-ROCKET offers interpretability, reproducibility and practical efficiency, which make it suitable as a baseline for rapid prototyping and low-data scenarios. Its main limitations are the lack of learned adaptation and potential memory/conditioning issues at extreme feature budgets. The experiments motivate directions such as adaptive kernels, dimensionality reduction, and hybrid trainable adapters to extend applicability.

9.3 Practical Implications and Future Work

The results of this thesis suggest that the 2D-ROCKET framework is particularly well-suited for scenarios with limited training data or constrained computational resources. Its lightweight and efficient design enables rapid prototyping, making it an attractive choice for situations where fast baseline evaluation is important. The pipeline is interpretable due to its transparent feature construction. This makes 2D-ROCKET particularly valuable in domains where understanding model decisions is critical.

Looking ahead, several directions for future research arise from the current findings. First, as the number of generated features grows, storage and computational requirements may become a limiting factor. Storage complexity could be addressed through dimensionality reduction or feature selection techniques. Doing so could further improve the practicality of the approach for large-scale applications.

Another avenue is to explore more structured or domain-aware strategies for kernel sampling. The current approach uses purely random selection. By incorporating prior knowledge about typical image patterns or spatial hierarchies, higher accuracy or greater efficiency may be achievable.

Hybrid pipelines offer another opportunity. Random convolutions could be combined with shallow trainable components to balance adaptivity. Additionally, multi-scale pooling strategies, such as spatial pyramid pooling, could be used to capture information at different spatial resolutions and increase representational power.

It should be noted that, in this work, the preprocessing bank was limited to simple per-channel normalization. Therefore, no explicit edge, gradient, or texture filters were applied before feature extraction. Future research could investigate the impact of incorporating more advanced preprocessing operators. This could enable the random feature pipeline to capture structural information more effectively, potentially improving classification performance, especially on complex or domain-specific datasets.

Further research is also needed to systematically examine the robustness of results when it comes to different random initializations (random seeds). Also, investigation of alternative statistical metrics for feature aggregation should be considered. While this thesis focused on local summary statistics, global or input-wide descriptors may provide complementary or more discriminative information. Moreover, future investigations could systematically evaluate individual statistics in isolation to clarify their unique contribution.

Additionally, future work could focus on optimizing and extending the set of preprocessing operators used as input channels. More diverse or domain-specific preprocessing banks might provide the random feature pipeline with richer and more discriminative input representations. Potentially, there could be advanced edge detectors, texture descriptors, or learned transformations incoorporated. Systematic comparative studies could help clarify which types of preprocessing are most beneficial for different image domains and tasks.

A promising application area for random convolutional kernels lies in non-object-specific datasets, like textures. This could be particularly relevant in industry and materials science, for example, for the automation of visual inspection processes or for robotics.

In addition to the standard data sets for object recognition, there were experiments on the KTH-TIPS dataset conducted to function as a proof of concept. KTH-TIPS is a small texture dataset encompassing ten different material classes. They include examples like aluminum foil, cotton, cork, and linen. The images vary in lighting, viewing angle, and scale, which can make recognition more difficult than with typical standard datasets [34]. This allows exploring the potential of the methods for real-world texture recognition scenarios using a prototype.

For the preliminary tests, a data preparation and pipeline similar to the previous experiments was set up. The 2D-ROCKET adaptation used the same design as in the CIFAR-10 experiments. It was then tested against the previously presented SVM and lightweight CNN implementations. Using the final 2D-ROCKET approach with the same settings as CIFAR-10, KTH-TIPS achieved an overall test accuracy of 94.7%. In comparison, the linear SVM-based approach achieved 43.5% and the small CNN model 52.4%. The total training and testing time for feature extraction and classification is approximately 0.66 seconds for the RidgeClassifier (after feature extraction). The SVM approach took around 15.6 seconds, while the CNN was faster with around 10.8 seconds. This serves as an initial validation of potentially interesting application areas for ROCKET adaptation in two-dimensional space.

Overall, these directions offer multiple pathways for extending and refining the 2D-ROCKET framework, both in terms of performance and applicability to a broader range of image classification tasks.

9.4 Reflection of the Thesis

The work emphasised the importance of careful experimental infrastructure and reproducible runs. In retrospect, early investment into a rigorous, versioned experiment pipeline (including seed sweeps and environment manifests) would have reduced later validation effort. The independent implementation of the full pipeline was challenging but yielded a reproducible baseline and valuable lessons regarding trade-offs between design complexity and experimental clarity.

9.5 Final Statement

Overall, 2D-ROCKET successfully bridges the gap between lightweight random-feature pipelines and image-appropriate spatial processing. It achieves competitive accuracy with minimal training effort and is computationally feasible for the regimes studied here. While it is not intended to replace highly optimized deep networks on the most challenging vision tasks, 2D-ROCKET provides a strong, interpretable, and efficient foundation for further methodological development and practical applications.

A Appendix

A.1 Repository

The complete source code for this work is publicly available at:
https://github.com/Felinchen76/ImageRocket
(Last updated: 16.02.2026, Commit: 2d622e7)

A.2 Dataset-wide normalization statistics

Listing A.1: Compute dataset-wide mean/std for MNIST and CIFAR-10

```
import torch
from torchvision import datasets, transforms

# MNIST (single-channel)
train_mnist = datasets.MNIST('./data', train=True, download=True,
   ↪ transform=transforms.ToTensor())
loader = torch.utils.data.DataLoader(train_mnist, batch_size=len(
   ↪ train_mnist), shuffle=False)
data = next(iter(loader))[0]  # returns images only, shape (N, C, H
   ↪ , W)

mnist_mean = data.mean().item()
mnist_std = data.std().item()

print(f"MNIST␣mean:␣{mnist_mean:.4f},␣std:␣{mnist_std:.4f}")

# CIFAR-10 (3 channels)
train_cifar = datasets.CIFAR10(root='./data', train=True, download=
   ↪ True, transform=transforms.ToTensor())
loader = torch.utils.data.DataLoader(train_cifar, batch_size=len(
   ↪ train_cifar), shuffle=False)
data = next(iter(loader))[0]  # shape (N, 3, H, W)

cifar_mean = data.mean(dim=(0,2,3))
cifar_std = data.std(dim=(0,2,3))

print('CIFAR-10␣mean:', [round(m.item(), 4) for m in cifar_mean])
print('CIFAR-10␣std:',  [round(s.item(), 4) for s in cifar_std])
```

A.3 Observed values (Output)

```
MNIST mean: 0.1307, std: 0.3081

CIFAR-10 mean: [0.4914, 0.4822, 0.4465]
CIFAR-10 std:  [0.2470, 0.2435, 0.2616]
```

A.4 config.py

Listing A.2: config.py — central configuration and loader factory

```
"""Central␣configuration␣source␣(single␣source␣of␣truth).

Only␣here␣ENV␣reads/defaults;␣other␣modules␣import␣from␣config.
"""
import os
from pathlib import Path
import random
import numpy as np
import torch

from typing import Tuple, Optional
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

# sklearn helpers (for build_ridge)
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import RidgeClassifier
from sklearn.pipeline import make_pipeline

#
# Basic experiment defaults
#
DEFAULT_SEED = 42

# calculated normalization constants for mnist and cifar
CIFAR10_MEAN = (0.4914, 0.4822, 0.4465)
CIFAR10_STD  = (0.2470, 0.2435, 0.2610)
MNIST_MEAN   = (0.1307,)
MNIST_STD    = (0.3081,)

#
# Batch / loader defaults (ENV-overridable)
#
BATCH_SIZE_TRAIN = int(os.environ.get("BATCH_SIZE_TRAIN", os.
   ↪ environ.get("BATCH_SIZE", "512")))
BATCH_SIZE_TEST  = int(os.environ.get("BATCH_SIZE_TEST",  os.
   ↪ environ.get("BATCH_SIZE", "512")))
BATCH_SIZE_DEFAULT = int(os.environ.get("BATCH_SIZE", str(
   ↪ BATCH_SIZE_TRAIN)))
```

```
NUM_WORKERS = int(os.environ.get("NUM_WORKERS", "2"))
PIN_MEMORY  = os.environ.get("PIN_MEMORY", "1").lower() not in ("0"
   ↪ , "false", "no")

def get_batch_config():
    return {
        "batch_size_train": BATCH_SIZE_TRAIN,
        "batch_size_test":  BATCH_SIZE_TEST,
        "num_workers":      NUM_WORKERS,
        "pin_memory":       PIN_MEMORY,
    }

#
# Seed & deterministic helpers
# -
def set_seed(seed: int = DEFAULT_SEED):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

def set_deterministic():
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.use_deterministic_algorithms(True, warn_only=True)

#
# Stride config (ENV-overridable)
#
STRIDE_1D = int(os.environ.get("STRIDE_1D", "1"))
STRIDE_2D = int(os.environ.get("STRIDE_2D", "1"))

def get_stride_config():
    return {"stride_1d": STRIDE_1D, "stride_2d": STRIDE_2D}

#
# Alphas helper
#
def _parse_int_list(s: str) -> list:
    return [int(x.strip()) for x in s.split(",") if x.strip() != ""
       ↪ ]

def get_alphas(default=None, env_key: str = "ALPHAS") -> list:
    if default is None:
        default = [1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400]
    s = os.environ.get(env_key, None)
    if s is None or s.strip() == "":
        return default
    try:
        return _parse_int_list(s) if "," in s else [int(s)]
    except Exception:
        return default

ALPHAS = get_alphas()

#
# Dataset loader helper
#
def _make_sel_idx(labels: np.ndarray, per_class: int, seed: int) ->
   ↪  np.ndarray:
    rng = np.random.RandomState(seed)
    sel = []
    for c in np.unique(labels):
        ids = np.where(labels == c)[0]
        pick = rng.choice(ids, size=min(per_class, len(ids)),
           ↪ replace=False)
        sel.extend(pick.tolist())
    return np.array(sel, dtype=np.int64)

def get_loaders(dataset: str,
                batch_size_train: Optional[int] = None,
                batch_size_test: Optional[int]  = None,
                seed: int = DEFAULT_SEED,
                quick_per_class: Optional[int] = None,
                num_workers: Optional[int] = None,
                pin_memory: Optional[bool] = None) -> Tuple[
                   ↪ DataLoader, DataLoader, int, Tuple[int, int],
                   ↪  int]:

    if dataset == "mnist":
        tf = transforms.Compose([transforms.ToTensor(), transforms.
           ↪ Normalize(MNIST_MEAN, MNIST_STD)])
        train_full = datasets.MNIST(root="/content", train=True,
           ↪ download=True, transform=tf)
        test_set   = datasets.MNIST(root="/content", train=False,
           ↪ download=True, transform=tf)
        in_channels, H, W = 1, 28, 28
    elif dataset == "cifar10":
        tf = transforms.Compose([transforms.ToTensor(), transforms.
           ↪ Normalize(CIFAR10_MEAN, CIFAR10_STD)])
        train_full = datasets.CIFAR10(root="/content", train=True,
           ↪ download=True, transform=tf)
        test_set   = datasets.CIFAR10(root="/content", train=False,
           ↪  download=True, transform=tf)
        in_channels, H, W = 3, 32, 32
    else:
        raise ValueError("unsupported␣dataset.␣use␣'mnist'␣or␣'
           ↪ cifar10'.")

    if quick_per_class is not None:
        labels = np.array(train_full.targets)
        sel_idx = _make_sel_idx(labels, quick_per_class, seed)
        train_set = Subset(train_full, sel_idx)
    else:
        train_set = train_full

    if batch_size_train is None: batch_size_train =
       ↪ BATCH_SIZE_TRAIN
    if batch_size_test  is None: batch_size_test  = BATCH_SIZE_TEST
    if num_workers      is None: num_workers      = NUM_WORKERS
    if pin_memory       is None: pin_memory       = PIN_MEMORY

    train_loader = DataLoader(train_set, batch_size=
       ↪ batch_size_train, shuffle=True,
                              num_workers=num_workers, pin_memory=
                                 ↪ pin_memory, generator=torch.
                                 ↪ Generator().manual_seed(seed))
    test_loader  = DataLoader(test_set, batch_size=batch_size_test,
       ↪  shuffle=False,
                              num_workers=num_workers, pin_memory=
                                 ↪ pin_memory)
    seq_len = H * W
    return train_loader, test_loader, in_channels, (H, W), seq_len

#
# Classifier builder
#
def build_ridge(alpha: int, scale: bool = True):
    steps = []
    if scale:
        steps.append(StandardScaler(with_mean=True, with_std=True))
    steps.append(RidgeClassifier(alpha=alpha))
    return make_pipeline(*steps)

#
# Exports / convenience values (single source)
#
SEED = int(os.environ.get("SEED", str(DEFAULT_SEED)))

BATCH_CONFIG = get_batch_config()
BATCH_SIZE_TRAIN = int(BATCH_CONFIG["batch_size_train"])
BATCH_SIZE_TEST  = int(BATCH_CONFIG["batch_size_test"])
NUM_WORKERS      = int(BATCH_CONFIG["num_workers"])
PIN_MEMORY       = bool(BATCH_CONFIG["pin_memory"])

STRIDE_CONFIG = get_stride_config()
STRIDE_1D = int(STRIDE_CONFIG.get("stride_1d", 1))
STRIDE_2D = int(STRIDE_CONFIG.get("stride_2d", 1))

# dataset canonical
_DATASET_ENV = os.environ.get("DATASET", "both").lower()
```

```
DATASET_LIST = ["mnist", "cifar10"] if _DATASET_ENV == "both" else
    ↪ [_DATASET_ENV]

# device
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ↪ ")

# cache / results convenience
CACHE_BASE = Path(os.environ.get("CACHE_DIR", "/content"))
CACHE_DIR = CACHE_BASE.joinpath("cache")
# get_results_file stellt sicher, dass das Zielverzeichnis
    ↪ existiert.

def get_results_file(name: str, ext: str = "csv", subfolder:
    ↪ Optional[str] = None) -> str:
    base = CACHE_DIR if subfolder is None else CACHE_BASE.joinpath(
        ↪ subfolder)
    base.mkdir(parents=True, exist_ok=True)
    return str(base.joinpath(f"{name}.{ext}"))
```

A.5 utils/constants.py

Listing A.3: utils/constants.py — feature and preprocessing constants

```
# central strings

FEATURE_TYPES_1D = ["ppv", "max", "mpv", "mipv", "lspv"]
FEATURE_TYPES_2D = ["ppv", "max", "mpv", "mipv_y", "mipv_x", "lspv"
    ↪ ]

# preprocessing modes
PREPROC_NONE = "none"
PREPROC_IMAGE_NORM = "per_image_norm"
PREPROC_SOBEL = "sobel"

PREPROC_MODES = {PREPROC_NONE, PREPROC_IMAGE_NORM, PREPROC_SOBEL}
```

A.6 utils/kernels.py

Listing A.4: utils/kernels.py — random kernel generation and hashing

```
import math
import numpy as np
from collections import OrderedDict
from typing import Dict, Tuple, List, Optional
import config

```

```
def make_random_kernels_1d(n_kernels: int, in_channels: int,
   ↪ k_choices: List[int],
                            seq_len: int, seed: Optional[int] = None
                               ↪ , dilation_cap: Optional[int] =
                               ↪ None) -> Dict[Tuple[int,int], List
                               ↪ [dict]]:
    if seed is None:
        seed = config.DEFAULT_SEED
    rng = np.random.RandomState(seed)
    kernels_by_size = OrderedDict()
    for _ in range(n_kernels):
        k = int(rng.choice(k_choices))
        if k == 1:
            d_max = 1
        else:
            d_max = max(1, (seq_len - 1) // (k - 1))
        if dilation_cap is not None:
            d_max = min(d_max, dilation_cap)
        d = int(rng.randint(1, d_max + 1))
        key = (k, d)
        fan_in = in_channels * k
        std = math.sqrt(2.0 / max(1, fan_in))
        w = rng.normal(0.0, std, size=(in_channels, k)).astype(np.
           ↪ float32)
        b = float(rng.normal(0, 0.1))
        kernels_by_size.setdefault(key, []).append({"weight": w, "
           ↪ bias": b})
    return kernels_by_size

def make_random_kernels_2d(n_kernels: int, in_channels: int,
   ↪ k_choices: List[int],
                            d_choices: List[int], seed: Optional[int
                               ↪ ] = None) -> Dict[Tuple[int,int],
                               ↪ List[dict]]:
    if seed is None:
        seed = config.DEFAULT_SEED
    rng = np.random.RandomState(seed)
    kernels_by_group = OrderedDict()
    for _ in range(n_kernels):
        k = int(rng.choice(k_choices))
        d = int(rng.choice(d_choices))
        key = (k, d)
        fan_in = in_channels * (k**2)
        std = math.sqrt(2.0 / max(1, fan_in))
        w = rng.normal(0.0, std, size=(in_channels, k, k)).astype(
           ↪ np.float32)
        b = float(rng.normal(0, 0.1))
        kernels_by_group.setdefault(key, []).append({"weight": w, "
           ↪ bias": b})
    return kernels_by_group
```

```
def count_kernels(kernels: Dict[Tuple[int,int], List[dict]]) -> int
   ↪ :
    return sum(len(lst) for lst in kernels.values()) if kernels is
        ↪ not None else 0

def kernel_hash(kernels: Dict[Tuple[int,int], List[dict]]) ->
   ↪ Optional[str]:
    import hashlib, numpy as _np
    if kernels is None:
        return None
    h = hashlib.md5()
    for (k,d), lst in kernels.items():
        for km in lst:
            h.update(km["weight"].tobytes())
            h.update(_np.array([km["bias"]], dtype=_np.float32).
               ↪ tobytes())
    return h.hexdigest()
```

A.7 utils/preproc.py

Listing A.5: utils/preproc.py — apply_preprocessing and preproc_output_channels

```
import torch
import torch.nn.functional as F
from typing import Optional
from utils.constants import PREPROC_NONE, PREPROC_IMAGE_NORM,
   ↪ PREPROC_SOBEL

# Conservative heuristics (tunable)
_MEAN_SKIP_TOL = 1e-3
_STD_SKIP_TOL = 1e-2
_STD_EPS = 1e-6

def apply_preprocessing(imgs: torch.Tensor, mode: Optional[str] =
   ↪ PREPROC_NONE, force: bool = False) -> torch.Tensor:
    if mode is None or mode == PREPROC_NONE:
        return imgs

    device = imgs.device
    dtype = imgs.dtype

    if mode == PREPROC_IMAGE_NORM:
        # Heuristic: skip if already approximately dataset-
           ↪ normalized across batch,
        # unless force=True was requested.
        if not force:
            with torch.no_grad():
                per_chan_mean = imgs.mean(dim=[0, 2, 3])        #
                   ↪ shape (C,)
                per_chan_std  = imgs.std(dim=[0, 2, 3], unbiased=
                   ↪ False)
            if torch.all(per_chan_mean.abs() < _MEAN_SKIP_TOL) and
               ↪ torch.all((per_chan_std - 1.0).abs() <
               ↪ _STD_SKIP_TOL):
                return imgs

        B, C, H, W = imgs.shape
        x = imgs.view(B, C, -1)
        mean = x.mean(dim=2, keepdim=True)
        std = x.std(dim=2, keepdim=True, unbiased=False).clamp(min=
           ↪ _STD_EPS)
        x = (x - mean) / std
        return x.view(B, C, H, W)

    if mode == PREPROC_SOBEL:
        # Vectorized depthwise Sobel for all channels at once
        B, C, H, W = imgs.shape
        # base kernels
        kx = torch.tensor([[1, 0, -1], [2, 0, -2], [1, 0, -1]],
           ↪ dtype=dtype, device=device).view(1, 1, 3, 3)
        ky = torch.tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]],
           ↪ dtype=dtype, device=device).view(1, 1, 3, 3)
        kx_rep = kx.repeat(C, 1, 1, 1)  # (C,1,3,3)
        ky_rep = ky.repeat(C, 1, 1, 1)
        gx = F.conv2d(imgs, kx_rep, padding=1, groups=C)
        gy = F.conv2d(imgs, ky_rep, padding=1, groups=C)
        mag = torch.sqrt(gx * gx + gy * gy + 1e-8)
        mag_mean = mag.mean(dim=1, keepdim=True)  # (B,1,H,W)
        return mag_mean

    raise ValueError(f"Unknown␣preprocessing␣mode:␣{mode}")


def preproc_output_channels(mode: Optional[str], in_channels: int)
   ↪ -> int:
    """
␣␣␣␣Return␣number␣of␣channels␣after␣preprocessing
␣␣␣␣"""
    if mode == PREPROC_SOBEL:
        return 1
    return in_channels
```

A.8 utils/features.py

Listing A.6: utils/features.py — feature extraction (1D and 2D) and statistics

```
# Feature-Extraction helpers (1D & 2D) with defensive checks.
#
import numpy as np
import torch
import torch.nn.functional as F
from typing import List, Tuple, Optional
from tqdm import tqdm
from .preproc import apply_preprocessing
import config
from utils.constants import PREPROC_IMAGE_NORM, PREPROC_SOBEL,
   ↪ PREPROC_NONE

def compute_mpv_1d(out: torch.Tensor) -> torch.Tensor:
    mask = (out > 0.0).float()
    return (out * mask).sum(dim=2) / mask.sum(dim=2).clamp(min=1.0)

def compute_mipv_1d(out: torch.Tensor) -> torch.Tensor:
    B, G, L = out.shape
    mask = (out > 0.0).float()
    pos_count = mask.sum(dim=2).clamp(min=1.0)
    idx = torch.arange(L, device=out.device, dtype=torch.float32).
       ↪ view(1, 1, L)
    mipv = (idx * mask).sum(dim=2) / pos_count
    if L > 1:
        mipv = mipv / (L - 1)
    return mipv

def compute_lspv_1d(out: torch.Tensor) -> torch.Tensor:
    mask = (out > 0.0).int()
    B, G, L = mask.shape
    runs = torch.zeros_like(mask, dtype=torch.int32)
    runs[:, :, 0] = mask[:, :, 0]
    for i in range(1, L):
        runs[:, :, i] = torch.where(mask[:, :, i] == 1, runs[:, :,
           ↪ i - 1] + 1, torch.zeros_like(runs[:, :, i]))
    lspv = runs.max(dim=2).values.float()
    return lspv

def compute_mpv_2d(out: torch.Tensor) -> torch.Tensor:
    mask = (out > 0.0).float()
    return (out * mask).sum(dim=[2, 3]) / mask.sum(dim=[2, 3]).
       ↪ clamp(min=1.0)

def compute_mipv_2d(out: torch.Tensor, normalize: bool = True) ->
   ↪ Tuple[torch.Tensor, torch.Tensor]:
    B, G, H, W = out.shape
    mask = (out > 0.0).float()
    pos_counts = mask.sum(dim=[2, 3]).clamp(min=1.0)
    yy = torch.arange(H, device=out.device, dtype=torch.float32).
       ↪ view(1, 1, H, 1)
    xx = torch.arange(W, device=out.device, dtype=torch.float32).
       ↪ view(1, 1, 1, W)
    my = (yy * mask).sum(dim=[2, 3]) / pos_counts
    mx = (xx * mask).sum(dim=[2, 3]) / pos_counts
    if normalize:
        if H > 1: my = my / (H - 1)
        if W > 1: mx = mx / (W - 1)
    return my, mx

def _make_pad_map(order, seq_or_hw, padding_mode: str, seed: int):
    rng = np.random.RandomState(seed)
    pad_map = {}
    for (k, d) in order:
        same_pad = (d * (k - 1)) // 2
        pad_map[(k, d)] = {"same": same_pad, "valid": 0, "pick":
           ↪ int(rng.randint(0, 2))}
    return pad_map

def extract_features_1d(loader, kernels_by_group, in_channels,
   ↪ seq_len, feature_types: List[str],
                        stride: int = 1, padding_mode: str = "
                          ↪ random", seed: Optional[int] = None,
                          ↪ device: Optional[object] = None):
    """
␣␣␣␣1D␣extractor.␣Uses␣GPU␣concat␣per␣batch␣and␣returns␣(X,␣y)␣
   ↪ numpy␣arrays.
␣␣␣␣Defensive:␣checks␣that␣kernel␣in_channels␣match␣input␣channels.
␣␣␣␣"""
    if seed is None:
        seed = config.DEFAULT_SEED
    device = device or torch.device("cuda" if torch.cuda.
       ↪ is_available() else "cpu")
    # kernels_by_group is expected as dict[(k,d)] -> list of {"
       ↪ weight","bias"}
    grouped_W, grouped_B, order = {}, {}, []
    for (k, d), lst in kernels_by_group.items():
        W = np.stack([km["weight"] for km in lst], axis=0)
        B = np.array([km["bias"] for km in lst], dtype=np.float32)
        grouped_W[(k, d)] = torch.from_numpy(W).to(device)
        grouped_B[(k, d)] = torch.from_numpy(B).to(device)
        order.append((k, d))

    def eff_k(k, d): return (k - 1) * d + 1
    order_filtered = [(k, d) for (k, d) in order if not (
       ↪ padding_mode == "valid" and eff_k(k, d) > seq_len)]
    pad_map = _make_pad_map(order, seq_len, padding_mode, seed) if
       ↪ padding_mode == "random" else None

    # ensure grouped_W in_channels match sequence channels
    if grouped_W:
        sample = next(iter(grouped_W.values()))
        # sample shape: (n_kernels_group, in_channels, k) for 1D
        expected_in_ch = sample.shape[1]
        actual_seq_ch = in_channels
        if expected_in_ch != actual_seq_ch:
            raise ValueError(
                f"Channel␣mismatch␣between␣kernels␣(in_channels={
                   ↪ expected_in_ch})␣"
                f"and␣provided␣sequence␣channels␣(in_channels={
                   ↪ actual_seq_ch}).␣"
            )

    X_parts, y_parts = [], []
    with torch.no_grad():

        for imgs, labels in tqdm(loader, desc=f"Feature␣extraction␣
           ↪ (1D/{padding_mode})", leave=False):
            imgs = imgs.to(device)
            Bsize = imgs.shape[0]
            seq = imgs.view(Bsize, in_channels, seq_len)
            group_feats_gpu = []
            for (k, d) in order_filtered:
                W = grouped_W[(k, d)]; Bv = grouped_B[(k, d)]
                if padding_mode == "same":
                    pad = (d * (k - 1)) // 2
                elif padding_mode == "valid":
                    pad = 0
                else:  # random
                    pad = int(pad_map[(k, d)]["same"]) if pad_map[(
                       ↪ k, d)]["pick"] == 0 else 0
                out = F.conv1d(seq, W, bias=Bv, stride=stride,
                   ↪ padding=pad, dilation=d)
                feats = []
                if "ppv" in feature_types: feats.append((out > 0.0)
                   ↪ .float().mean(dim=2))
                if "max" in feature_types: feats.append(out.amax(
                   ↪ dim=2))
                if "mpv" in feature_types: feats.append(
                   ↪ compute_mpv_1d(out))
                if "mipv" in feature_types: feats.append(
                   ↪ compute_mipv_1d(out))
                if "lspv" in feature_types: feats.append(
                   ↪ compute_lspv_1d(out))
                if feats:
                    group_feats_gpu.append(torch.cat(feats, dim=1))
            if group_feats_gpu:
                batch_concat = torch.cat(group_feats_gpu, dim=1).
                   ↪ cpu().numpy()
            else:
                batch_concat = np.zeros((imgs.shape[0], 0), dtype=
                   ↪ np.float32)
            X_parts.append(batch_concat)
            y_parts.append(labels.cpu().numpy())
    X = np.vstack(X_parts) if len(X_parts) > 0 else np.zeros((0, 0)
       ↪ , dtype=np.float32)
    y = np.concatenate(y_parts) if len(y_parts) > 0 else np.zeros
        ↪ ((0,), dtype=np.int64)
    return X, y

def extract_features_2d(loader, kernels_by_group, feature_types:
    ↪ List[str], image_hw: Tuple[int, int],
                        padding_mode: str = "valid", stride: int =
                            ↪ 1, seed: Optional[int] = None,
                        preproc_mode: str = PREPROC_NONE, device:
                            ↪ Optional[object] = None,
                        preproc_force: bool = False):
    """
␣␣␣␣2D␣extractor.
␣␣␣␣␣␣-␣Default␣preproc_mode␣is␣PREPROC_NONE␣(no␣extra␣per-image␣
    ↪ preprocessing).
␣␣␣␣␣␣-␣To␣force␣per-image␣normalization␣even␣if␣the␣loader␣already
    ↪ ␣applies␣dataset-wide␣Normalize,
␣␣␣␣␣␣␣␣set␣preproc_force=True.␣This␣ensures␣the␣preproc␣bank␣
    ↪ actually␣performs␣per-image␣normalization

␣␣␣␣Returns␣(X,y)␣as␣numpy␣arrays.
␣␣␣␣Defensive:␣checks␣that␣kernel␣in_channels␣match␣preprocessed␣
    ↪ image␣channels.
␣␣␣␣"""
    if seed is None:
        seed = config.DEFAULT_SEED
    device = device or torch.device("cuda" if torch.cuda.
        ↪ is_available() else "cpu")
    grouped_W, grouped_B, order = {}, {}, []
    for (k, d), lst in kernels_by_group.items():
        W_np = np.stack([km["weight"] for km in lst], axis=0)
        B_np = np.array([km["bias"] for km in lst], dtype=np.
            ↪ float32)
        grouped_W[(k, d)] = torch.from_numpy(W_np).to(device)
        grouped_B[(k, d)] = torch.from_numpy(B_np).to(device)
        order.append((k, d))

    H_img, W_img = image_hw
    def eff_k(k, d): return (k - 1) * d + 1
    order_filtered = [(k, d) for (k, d) in order if not (
        ↪ padding_mode == "valid" and (eff_k(k, d) > H_img or eff_k
        ↪ (k, d) > W_img))]
    pad_map = _make_pad_map(order, (H_img, W_img), padding_mode,
        ↪ seed) if padding_mode == "random" else None

    X_parts, y_parts = [], []
    with torch.no_grad():
        for imgs, labels in tqdm(loader, desc=f"Feature␣extraction␣
            ↪ (2D/{padding_mode})", leave=False):
            imgs = imgs.to(device)
            # apply preprocessing with optional force override
        imgs_used = apply_preprocessing(imgs, preproc_mode,
           ↪ force=preproc_force)

        # Defensive channel check: ensure kernel in_channels
           ↪ match imgs_used channels
        if grouped_W:
            sample = next(iter(grouped_W.values()))
            # sample shape: (n_kernels_group, in_channels, k, k
               ↪ ) for 2D
            expected_in_ch = sample.shape[1]
            actual_in_ch = imgs_used.shape[1]
            if expected_in_ch != actual_in_ch:
                raise ValueError(
                    f"Channel␣mismatch␣between␣kernels␣(
                       ↪ in_channels={expected_in_ch})␣"
                    f"and␣preprocessed␣images␣(channels={
                       ↪ actual_in_ch}).␣"
                    "If␣PREPROC_SOBEL␣is␣used,␣create␣kernels␣
                       ↪ with␣in_channels=1␣"
                )

        group_feats_gpu = []
        for (k, d) in order_filtered:
            if padding_mode == "same":
                pad = ((k - 1) * d) // 2
            elif padding_mode == "valid":
                pad = 0
            else:
                effK = eff_k(k, d)
                if effK <= H_img and effK <= W_img:
                    pad = pad_map[(k, d)]["same"] if pad_map[(k
                       ↪ , d)]["pick"] == 0 else 0
                else:
                    pad = pad_map[(k, d)]["same"]
            out = F.conv2d(imgs_used, grouped_W[(k, d)], bias=
               ↪ grouped_B[(k, d)], stride=stride, padding=pad
               ↪ , dilation=d)
            feats = []
            if "ppv" in feature_types: feats.append((out > 0.0)
               ↪ .float().mean(dim=[2, 3]))
            if "max" in feature_types: feats.append(out.amax(
               ↪ dim=[2, 3]))
            if "mpv" in feature_types: feats.append(
               ↪ compute_mpv_2d(out))
            if "mipv_y" in feature_types or "mipv_x" in
               ↪ feature_types:
                my, mx = compute_mipv_2d(out, normalize=True)
                if "mipv_y" in feature_types: feats.append(my)
                if "mipv_x" in feature_types: feats.append(mx)
            if feats:
                group_feats_gpu.append(torch.cat(feats, dim=1))
        if group_feats_gpu:
            batch_concat = torch.cat(group_feats_gpu, dim=1).
               ↪ cpu().numpy()
        else:
            batch_concat = np.zeros((imgs.shape[0], 0), dtype=
               ↪ np.float32)
        X_parts.append(batch_concat)
        y_parts.append(labels.cpu().numpy())
X = np.vstack(X_parts) if len(X_parts) > 0 else np.zeros((0, 0)
   ↪ , dtype=np.float32)
y = np.concatenate(y_parts) if len(y_parts) > 0 else np.zeros
   ↪ ((0,), dtype=np.int64)
return X, y
```

A.9 utils/eval.py

Listing A.7: utils/eval.py — training/sweep helper and reporting utilities

```
import time
from config import build_ridge
from sklearn.metrics import accuracy_score, classification_report,
   ↪ confusion_matrix

def train_and_sweep_alphas(X_train, y_train, X_test, y_test, alphas
   ↪ ):
    best_alpha, best_acc, best_time, best_pred = None, -1.0, None,
       ↪ None
    for alpha in alphas:
        pipe = build_ridge(alpha=alpha, scale=True)
        t0 = time.time()
        pipe.fit(X_train, y_train)
        tt = time.time() - t0
        ypred = pipe.predict(X_test)
        acc = accuracy_score(y_test, ypred)
        if acc > best_acc:
            best_alpha, best_acc, best_time, best_pred = alpha, acc
               ↪ , tt, ypred
    return best_alpha, best_acc, best_time, best_pred

def per_class_accuracy(y_true, y_pred):
    """
␣␣␣␣Compute␣per-class␣accuracy␣as␣a␣dict␣{class_label:␣accuracy}.
␣␣␣␣"""
    classes = sorted(set(y_true))
    return {
        str(c): float((y_pred[y_true == c] == c).mean())
        if (y_true == c).sum() > 0 else None
        for c in classes
    }
```

```
def evaluate_and_report(model_name: str, dataset: str, seed: int,
  ↪ y_true, y_pred, scores=None, train_time: float = None):
    acc = accuracy_score(y_true, y_pred)
    print(f"[{model_name}][{dataset}][seed={seed}]␣acc={acc:.4f}␣|␣
      ↪ train_time={train_time␣if␣train_time␣is␣not␣None␣else␣
      ↪ '-'}s")
    try:
        print(classification_report(y_true, y_pred, digits=4))
    except Exception as e:
        print("classification_report␣failed:", e)
    try:
        cm = confusion_matrix(y_true, y_pred)
        print("Confusion␣matrix:\n", cm)
    except Exception as e:
        print("confusion_matrix␣failed:", e)
    return acc
```

A.10 utils/sanity.py

Listing A.8: utils/sanity.py — lightweight assertions and size estimates

```
from typing import Dict, Tuple
import numpy as _np

def assert_kernel_count(kernels, expected):
    total = sum(len(lst) for lst in kernels.values()) if kernels is
      ↪  not None else 0
    assert total == expected, f"Kernel␣count␣mismatch:␣expected␣{
      ↪ expected},␣got␣{total}"
    return total

def assert_feature_dim(X_train, expected):
    got = 0
    if X_train is None:
        got = 0
    else:
        shape = getattr(X_train, "shape", None)
        if shape is not None and len(shape) >= 2:
            got = int(shape[1])
        else:
            got = 0
    assert got == expected, f"Feature␣dim␣mismatch:␣expected␣{
      ↪ expected},␣got␣{got}"
    return got

def estimate_feature_bytes(n_samples, feature_dim, dtype_bytes=4):
    return n_samples * feature_dim * dtype_bytes
```

A.11 utils/settings.py

Listing A.9: utils/settings.py — cache/result path helpers

```
from pathlib import Path
import config

def get_cache_dir(subfolder: str = None) -> Path:
    """
␣␣␣␣returns␣Path␣to␣cache␣(standard:␣config.cache_dir)
␣␣␣␣optional:␣subfolder␣under␣cache_base␣(config.cache_base)
␣␣␣␣"""
    # config.CACHE_DIR is a path -> return it or the subfolder
       ↪ under CACHE_BASE.
    if subfolder is None:
        return Path(config.CACHE_DIR)
    else:
        p = Path(config.CACHE_BASE).joinpath(subfolder)
        p.mkdir(parents=True, exist_ok=True)
        return p

def get_results_file(name: str, ext: str = "csv", subfolder: str =
   ↪ None) -> str:
    """
␣␣␣␣Implementation␣found␣in␣config␣as␣single␣source␣of␣truth
␣␣␣␣"""
    return config.get_results_file(name, ext=ext, subfolder=
       ↪ subfolder)
```

A.12 Minimal 1D ROCKET run

Listing A.10: Minimal 1D ROCKET run (experiment script)

```
# Main part from the notebook that runs the minimal 1D ROCKET
   ↪ experiments
set_seed(SEED)
set_deterministic()

# experiment-local choices
PADDING_MODE_1D = os.environ.get("PADDING_MODE_1D", "random").lower
   ↪ ()
MODE = "pure_rocket_1d"

N_KERNELS = int(os.environ.get("N_KERNELS", "3024"))
KERNEL_SIZE_CANDIDATES = [7, 9, 11]
STRIDE = STRIDE_1D

# build kernels and extract features
train_loader, test_loader, in_channels, (H, W), seq_len =
   ↪ get_loaders(dataset, seed=SEED)
```

```
kernels = make_random_kernels_1d(N_KERNELS, in_channels,
    ↪ KERNEL_SIZE_CANDIDATES, seq_len, seed=SEED)
X_train, y_train = extract_features_1d(train_loader, kernels,
    ↪ in_channels, seq_len, feature_types=["ppv","max"],
    ↪ padding_mode=PADDING_MODE_1D, seed=SEED)
X_test, y_test   = extract_features_1d(test_loader,  kernels,
    ↪ in_channels, seq_len, feature_types=["ppv","max"],
    ↪ padding_mode=PADDING_MODE_1D, seed=SEED)

# sweep alphas and train ridge classifier (report best)
for alpha in ALPHAS:
    pipe = build_ridge(alpha=alpha, scale=True)
    pipe.fit(X_train, y_train)
    acc = accuracy_score(y_test, pipe.predict(X_test))
    print(f"alpha={alpha}␣acc={acc:.4f}")
```

A.13 SVM baseline (raw images ± random projection)

Listing A.11: SVM baseline script (raw images with optional RP)

```
# SVM baseline excerpt
set_seed(SEED)
set_deterministic()

# load data -> flatten
X_tr, y_tr = loader_to_numpy(train_loader, flatten=True)
X_te, y_te = loader_to_numpy(test_loader,  flatten=True)

# optional projection
if rp_dim is not None:
    rp = GaussianRandomProjection(n_components=rp_dim, random_state
        ↪ =SEED)
    X_tr = rp.fit_transform(X_tr); X_te = rp.transform(X_te)

clf = LinearSVC(C=0.1, max_iter=max_iter, random_state=SEED)
clf.fit(X_tr, y_tr)
acc = float((clf.predict(X_te) == y_te).mean())
print(f"SVM␣acc={acc:.4f}")
```

A.14 Lightweight CNN

Listing A.12: Small CNN training loop (benchmark)

```
# Small CNN excerpt
class SmallCNN(nn.Module):
    def __init__(self, in_channels: int, img_size: int, num_classes
       ↪ : int = 10):
        super().__init__()
        self.features = nn.Sequential(nn.Conv2d(in_channels, 32, 3,
           ↪  padding=1), nn.ReLU(inplace=True),
                                     nn.MaxPool2d(2), nn.Conv2d
                                       ↪ (32, 64, 3, padding=1),
                                       ↪  nn.ReLU(inplace=True),
                                     nn.MaxPool2d(2))
        feat = img_size // 4
        self.classifier = nn.Sequential(nn.Flatten(), nn.Linear(64*
           ↪ feat*feat,128), nn.ReLU(inplace=True), nn.Linear(128,
           ↪ num_classes))

    def forward(self, x): return self.classifier(self.features(x))

# training loop
model = SmallCNN(in_channels, img_sz).to(device)
opt = optim.Adam(model.parameters(), lr=1e-3); crit = nn.
   ↪ CrossEntropyLoss()
for ep in range(1, epochs+1):
    model.train()
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        opt.zero_grad(); loss = crit(model(xb), yb); loss.backward
           ↪ (); opt.step()
    acc_val, _, _, _ = eval_loader(model, test_loader, device)
    print(f"epoch␣{ep}␣|␣test␣acc␣{acc_val:.4f}")
```

Bibliography

[1] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: A review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, Jul. 2019, Accessed: 2026-02-07. DOI: 10.1007/s10618-019-00619-1. [Online]. Available: https://link.springer.com/article/10.1007/s10618-019-00619-1.

[2] C. Solomon and T. Breckon, *Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab*. Chichester, UK: John Wiley & Sons, 2011, Accessed: 2026-02-07.

[3] A. Dempster, F. Petitjean, and G. I. Webb, "ROCKET: Exceptionally fast and accurate time series classification using random convolutional kernels," *Data Mining and Knowledge Discovery*, vol. 34, no. 5, pp. 1454–1495, Sep. 2020, Accessed: 2026-02-11. DOI: 10.1007/s10618-020-00701-z. arXiv: 1910.13051 [cs]. [Online]. Available: https://arxiv.org/pdf/1910.13051.

[4] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. New York: Pearson, 2018, Accessed: 2026-02-11. [Online]. Available: https://www.researchgate.net/publication/333856607_Digital_Image_Processing_Second_Edition/link/5d096caa299bf1f539cefac4/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uIn19.

[5] F. Yu and V. Koltun, *Multi-scale context aggregation by dilated convolutions*, Accessed: 2026-02-07, 2016. arXiv: 1511.07122 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1511.07122.

[6] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, Accessed: 2026-02-07, 2018. arXiv: 1603.07285 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1603.07285.

[7] M. Krichen, "Convolutional neural networks: A survey," *Computers*, vol. 12, p. 151, 2023, Accessed: 2026-02-07. DOI: 10.3390/computers12080151. [Online]. Available: https://doi.org/10.3390/computers12080151.

[8] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Computation*, vol. 29, pp. 2352–2449, Sep. 2017, Accessed: 2026-02-11. DOI: 10.1162/NECO_a_00990.

[9] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995, Accessed: 2026-02-07. DOI: 10.1007/BF00994018. [Online]. Available: https://link.springer.com/article/10.1007/BF00994018.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, Accessed: 2026-02-07.

[11] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, *Johnson-Lindenstrauss Lemma, Linear and Nonlinear Random Projections, Random Fourier Features, and Random Kitchen Sinks: Tutorial and Survey*, Accessed: 2026-02-07, Aug. 2021. DOI: 10.48550/arXiv.2108.04172. arXiv: 2108.04172 [stat]. [Online]. Available: https://arxiv.org/abs/2108.04172.

[12] K. P. Murphy, *Machine Learning: A Probabilistic Perspective* (Adaptive Computation and Machine Learning Series), 4. print. (fixed many typos). Cambridge, Mass.: MIT Press, 2013, Accessed: 2026-02-07. [Online]. Available: https://www.cs.ubc.ca/~murphyk/MLbook/pml-toc-1may12.pdf.

[13] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing* (Always Learning), Third edition, Pearson New international edition. Harlow: Pearson, 2014, Accessed: 2026-02-07. [Online]. Available: https://api.pageplace.de/preview/DT0400.9781292038155_A24581738/preview-9781292038155_A24581738.pdf.

[14] Y. LeCun, C. Cortes, and C. J. C. Burges, *Mnist database of handwritten digits*, https://archive.ics.uci.edu/dataset/683/mnist+database+of+handwritten+digits, Accessed: 2026-01-24, 2024.

[15] A. Krizhevsky, *Cifar-10 object recognition dataset*, https://archive.ics.uci.edu/dataset/691/cifar+10, Accessed: 2026-01-24, 2024.

[16] C. Wöhler, *3D Computer Vision: Efficient Methods and Applications*, 2nd ed. Springer, 2015, Accessed: 2026-02-11. [Online]. Available: https://link.springer.com/book/10.1007/978-3-662-45129-8.

[17] G. K. V. and D. J. V. Gripsy, “Image classification using hog and lbp feature descriptors with svm and cnn,” *International Journal of Engineering Research and Technology (IJERT)*, vol. V8, no. 4, 2020, Accessed: 2026-02-11. DOI: 10.17577/IJERTCONV8IS04021. [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/110347554/image-classification-using-hog-and-lbp-feature-descriptors-with-svm-and-cnn-IJERTCONV8IS04021-libre.pdf.

[18] N. Dalal and B. Triggs, “Histograms of Oriented Gradients for Human Detection,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, Accessed: 2026-02-07, vol. 1, San Diego, CA, USA: IEEE, 2005, pp. 886–893. DOI: 10.1109/CVPR.2005.177. [Online]. Available: https://ieeexplore.ieee.org/document/1467360.

[19] M. Pietikäinen, A. Hadid, G. Zhao, and T. Ahonen, *Computer vision using local binary patterns (overview)*, https://link.springer.com/book/10.1007/978-0-85729-748-8, Accessed: 2026-02-07, 2011.

[20] M. Awad and R. Khanna, “Support Vector Machines for Classification,” in *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*, Accessed: 2026-02-11, Berkeley, CA: Apress, 2015, pp. 39–66. DOI: 10.1007/978-1-4302-5990-9_3. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4302-5990-9_3.

[21] A. Jakulin, M. Možina, J. Demšar, I. Bratko, and B. Zupan, *Nomograms for visualizing support vector machines*, Accessed: 2026-02-11, Aug. 2005. DOI: 10.1145/1081870.1081886. [Online]. Available: https://www.researchgate.net/publication/33550089_Nomograms_for_Visualizing_Support_Vector_Machines.

[22] A. H. Alshammari, G. Bencsik, and A. H. Ali, "A Survey of Six Classical Classifiers, Including Algorithms, Methodological Characteristics, Foundational Variants, and Recent Advances," *Algorithms*, vol. 19, no. 1, p. 37, Jan. 2026, Accessed: 2026-02-07. DOI: 10.3390/a19010037. [Online]. Available: https://www.mdpi.com/1999-4893/19/1/37.

[23] W. Sun, X. Zhang, and X. He, "Lightweight image classifier using dilated and depthwise separable convolutions," *Journal of Cloud Computing*, vol. 9, no. 1, p. 55, Sep. 2020, Accessed: 2026-02-07. DOI: 10.1186/s13677-020-00203-9. [Online]. Available: https://link.springer.com/article/10.1186/s13677-020-00203-9.

[24] F. Chen, S. Li, J. Han, F. Ren, and Z. Yang, "Review of Lightweight Deep Convolutional Neural Networks," *Archives of Computational Methods in Engineering*, vol. 31, no. 4, pp. 1915–1937, May 2024, Accessed: 2026-02-07. DOI: 10.1007/s11831-023-10032-z. [Online]. Available: https://link.springer.com/article/10.1007/s11831-023-10032-z.

[25] F. Lei, X. Liu, Q. Dai, and B. W.-K. Ling, "Shallow convolutional neural network for image classification," *SN Applied Sciences*, vol. 2, no. 1, p. 97, Dec. 2019, Accessed: 2026-02-07. DOI: 10.1007/s42452-019-1903-4. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s42452-019-1903-4.pdf.

[26] H. Song, "Comparison of different depth of convolutional neural network deep and shallow cnn comparison based on fer-2013," *Highlights in Science, Engineering and Technology*, vol. 41, pp. 80–86, Mar. 2023, Accessed: 2026-02-07. DOI: 10.54097/hset.v41i.6746. [Online]. Available: https://www.researchgate.net/publication/369868403_Comparison_of_Different_Depth_of_Convolutional_Neural_Network_Deep_and_shallow_CNN_comparison_based_on_FER-2013.

[27] M. Middlehurst, P. Schäfer, and A. Bagnall, "Bake off redux: A review and experimental evaluation of recent time series classification algorithms," *Data Mining and Knowledge Discovery*, vol. 38, no. 4, pp. 1958–2031, Jul. 2024, Accessed: 2026-02-11. DOI: 10.1007/s10618-024-01022-1. arXiv: 2304.13029 [cs]. [Online]. Available: https://doi.org/10.1007/s10618-024-01022-1.

[28] A. Dempster, D. F. Schmidt, and G. I. Webb, "MINIROCKET: A very fast (almost) deterministic transform for time series classification," *CoRR*, vol. abs/2012.08791, 2020, Accessed: 2026-02-11. arXiv: 2012.08791. [Online]. Available: https://arxiv.org/abs/2012.08791.

[29] C. W. Tan, A. Dempster, C. Bergmeir, and G. I. Webb, *MultiRocket: Multiple pooling operators and transformations for fast and effective time series classification*, Accessed: 2026-02-11, Feb. 2022. DOI: 10.48550/arXiv.2102.00457. arXiv: 2102.00457 [cs].

[30] A. Dempster, D. F. Schmidt, and G. I. Webb, *HYDRA: Competing convolutional kernels for fast and accurate time series classification*, Accessed: 2026-02-11, Mar. 2022. DOI: 10.48550/arXiv.2203.13652. arXiv: 2203.13652 [cs]. [Online]. Available: https://arxiv.org/abs/2203.13652.

[31] A. Rahimi and B. Recht, *Random Features for Large-Scale Kernel Machines*, Accessed: 2026-02-07. [Online]. Available: https://people.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf.

[32] Y. Xue and U. Roshan, “Random depthwise signed convolutional neural networks,” *CoRR*, vol. abs/1806.05789, 2018, Accessed: 2026-02-07. arXiv: `1806.05789`. [Online]. Available: `http://arxiv.org/abs/1806.05789`.

[33] M. Ren, *Facial Expression Classification with Random Filters Feature Extraction*, Accessed: 2026-02-07. [Online]. Available: `https://www.cs.toronto.edu/~mren/projects/csc411_a4_report.pdf`.

[34] K. Grauman and H. Taloen, *Kth-tips image data set*, `https://www.csc.kth.se/cvap/databases/kth-tips/index.html`, Accessed: 2026-01-24, 2004.

Documentation of the use of AI tools

Purpose of Use	**AI Tool Used**
Support with plot creation (syntax, formatting), debugging advice, pipeline checks (e.g. validation of global settings used in every setup), creating structured comments, formatting of code output, and prototyping	GitHub Copilot
Support with LaTeX formatting (tables, layout, structural issues), clarification of comprehension questions, and feedback on linguistic clarity and sentence structure	ChatGPT (GPT)
Translation of individual text passages (German $\leftrightarrow$ English)	DeepL
Initial orientation in scientific publications (scanning abstracts and identifying key aspects)	NotebookLLM
Grammar and spell checker	QuillBot