Rock, Paper, Scissors: CNN for Image Classification

Greta Zehnder

February 12, 2026

Abstract

This project aims to investigate the application of Convolutional Neural Networks (CNNs) to the task of image classification using a Rock-Paper-Scissors (RPS) dataset, with the objective of designing, training, and evaluating multiple deep learning models.

The experimental pipeline is composed of dataset exploration, data preprocessing (which includes train/validation/test splitting, input normalization, and data augmentation), followed by the development of three CNN models (ordered by increasing complexity), and their supervised training and performance evaluation. Finally, a generalization part is carried out to highlight the effectiveness of using CNNs for image classification tasks.

The entire study was carried out in accordance with the official TensorFlow/Keras API documentation.

Contents

1 Introduction

Convolutional Neural Networks are widely adopted in image classification tasks because they provide a flexible framework for learning visual patterns directly from image data while supporting different architectural and training choices.

In this context, the development of a CNN can be seen as an exploratory process, in which progressively more complex modeling decisions are introduced and evaluated, moving from simple baseline architectures to manually tuned configurations and, eventually, to more systematic hyperparameter search strategies. Exploring these choices is therefore not only a means of improving performance, but also a way to better understand the sensitivity of a model to different design decisions and training conditions, and to observe how changes in architecture and optimization affect learning dynamics in practice.

This exploratory process is computationally expensive, as training and evaluating multiple configurations requires significant time and resources, which naturally constrains the scope of experimentation and motivates a controlled and incremental evaluation under realistic computational limitations.

2 Data exploration and preprocessing

2.1 Exploratory Data Analysis

The dataset used in this project consists of RGB images representing hand gestures corresponding to the three classes of the Rock-Paper-Scissors game, namely rock, paper, and scissors, and includes a total of 2188 images organized into class-specific subdirectories. The distribution of images across classes is relatively balanced, with 726 images belonging to the rock class, 712 to paper, and 750 to scissors, a property that is particularly relevant for supervised classification tasks, as it allows model performance across classes to be analyzed without strong bias effects.

All images are stored in PNG format and were acquired under controlled conditions, with a uniform green background and consistent lighting and white balance, which limit environmental variability while still preserving meaningful differences in hand shape, orientation, and gesture configuration. An analysis of both absolute class frequencies and relative class proportions confirms the near-uniform representation of the three categories, while a visual inspection of randomly sampled images from each class highlights the presence of intra-class variability related to hand positioning and finger articulation.

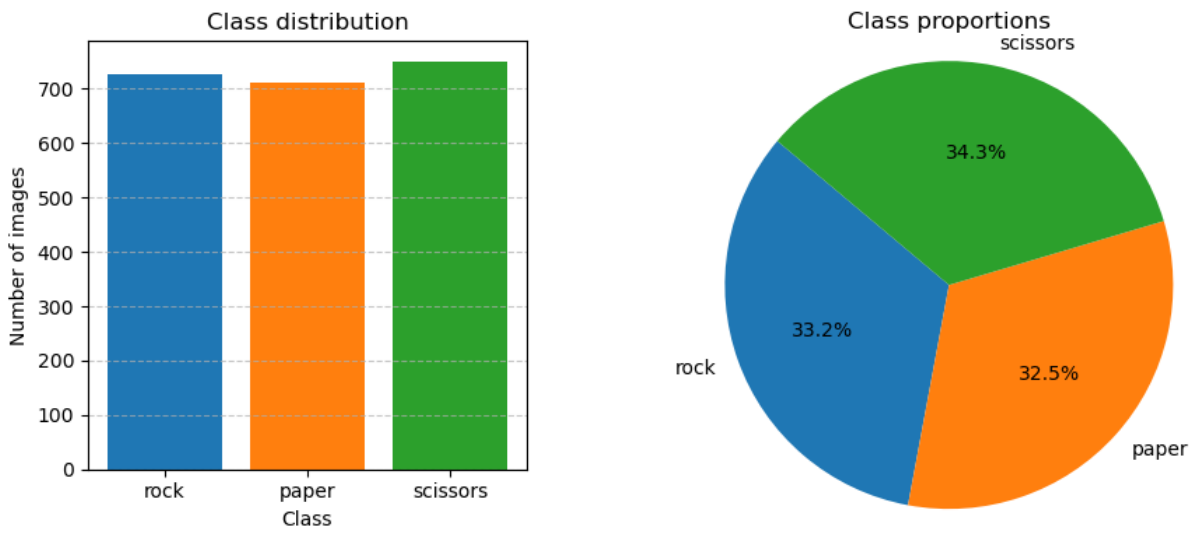

Figure 1: Class distribution (left) and class proportions (right) of the Rock-Paper-Scissors dataset.

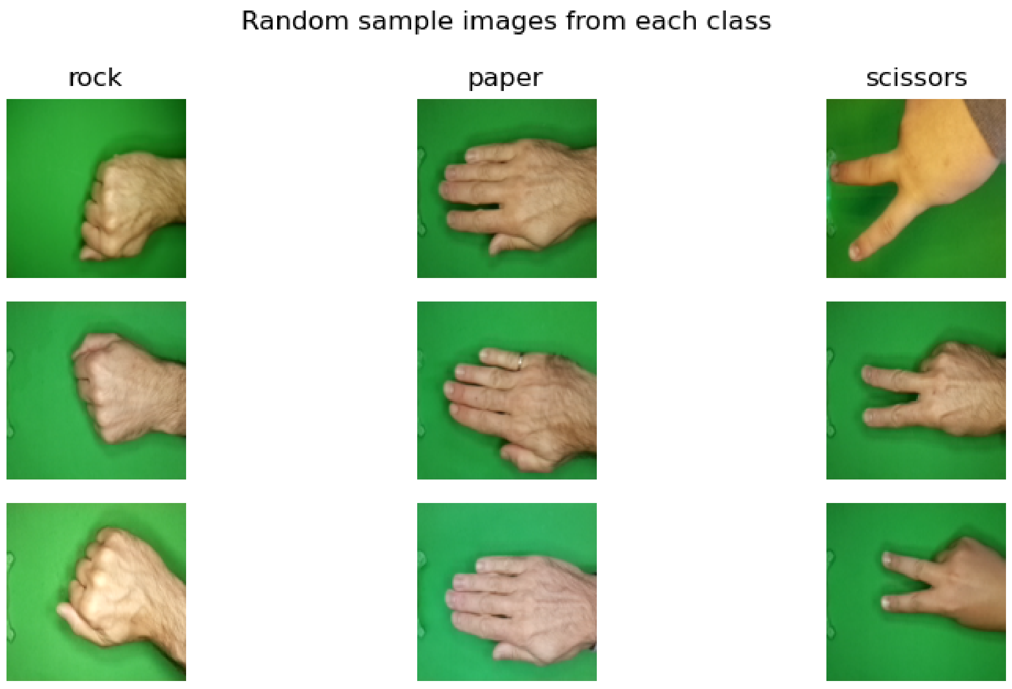

Figure 2: Randomly selected sample images from each class of the Rock-Paper-Scissors dataset.

2.2 Preprocessing

2.2.1 Train, validation and test splitting

The dataset was split into training, validation, and test subsets prior to model training in order to enable both hyperparameter tuning and an unbiased final evaluation. The splitting procedure was applied independently to each class, using a fixed random seed to ensure reproducibility, with 15% of the images assigned to the validation set, 15% to the test set, and the remaining samples used for training.

The resulting subsets were stored in a structured directory hierarchy with separate folders for each split and class, allowing a clear separation between training and evaluation data and ensuring compatibility with standard deep learning data-loading utilities.

2.2.2 Normalization

Image normalization was applied as part of the data input pipeline by rescaling pixel intensities from the original $[0, 255]$ range to $[0, 1]$ using a dedicated `Rescaling(1./255)` layer provided by the TensorFlow/Keras API [33]. This normalization step was applied uniformly to the training, validation, and test datasets in order to ensure consistent input scaling across all phases of model development and evaluation, improving numerical stability during optimization.

The datasets were constructed using the `image_dataset_from_directory` utility [34, 13], and normalization was implemented through a mapping operation within the `tf.data` pipeline, leveraging parallel execution to improve performance [31]. In addition, dataset caching and prefetching were employed to reduce input latency and optimize data throughput during both training and evaluation [30].

2.2.3 Data augmentation

Data augmentation was introduced to improve robustness and reduce overfitting by exposing the model, during training, to realistic variations of the input images that preserve the underlying class label. In this project, augmentation was implemented through Keras image preprocessing layers [11], following the standard workflow recommended in the official TensorFlow documentation [26].

A dedicated augmentation module was defined as a small preprocessing pipeline composed of random horizontal flipping, small random rotations, and random zoom transformations, which are particularly suitable for hand-gesture images because they mimic plausible changes in pose and framing without altering the semantic meaning of the gesture. Crucially, these stochastic transformations were applied only to the training data, while validation and test sets were kept unmodified in order to ensure that evaluation metrics reflect performance on the original data distribution.

Finally, the augmentation layers were designed to be integrated as part of the model architecture, so that they are active during training but automatically disabled at inference time, ensuring consistency between training and deployment behavior [29].

3 CNN architecture and training

3.1 Model A: baseline CNN

Model A is implemented as a baseline convolutional neural network using the Keras high-level API and the `Sequential` model paradigm, which allows layers to be stacked linearly in a clear and interpretable manner [16, 22, 23]. The objective of this model is to establish a simple yet representative reference architecture for the Rock–Paper–Scissors image classification task, against which more advanced models can be systematically compared.

As introduced in the previous section, data augmentation is integrated directly into the model pipeline and applied immediately after the input layer. This design choice ensures that augmentation is performed exclusively during training, while validation and test samples remain unaltered.

3.1.1 Architectural structure

The architecture of Model A follows a standard convolutional design commonly adopted for image classification tasks and is summarized schematically below. All layers are implemented using the Keras Layers API [14].

- **Input layer**: RGB images of fixed spatial resolution are provided to the network through an explicit input layer [12].
- **Data augmentation**: the augmentation pipeline described in the previous section is applied during training to improve robustness to geometric variations.
- **First convolutional block**: a Conv2D layer with 32 filters and 3×3 kernels is used to extract low-level visual features, followed by a MaxPooling2D layer that reduces the spatial resolution [5, 18]. ReLU activations are employed to introduce non-linearity [21].
- **Second convolutional block**: a second Conv2D layer increases the number of filters to 64, enabling the network to learn more abstract feature representations. Spatial down-sampling is again performed via max-pooling [5, 18].
- **Flatten layer**: the resulting feature maps are flattened into a one-dimensional vector to interface with the fully connected layers [9].
- **Fully connected layer**: a dense layer with 128 units and ReLU activation combines the extracted features into a compact representation [6, 21].
- **Output layer**: the final dense layer consists of three neurons with softmax activation, producing a normalized probability distribution over the three target classes [6].

The use of multiple convolutional blocks allows the network to progressively learn hierarchical feature representations, where early layers capture low-level visual patterns while deeper layers encode increasingly abstract and discriminative features [25]. At the same time, limiting the architecture to two convolutional blocks keeps the model complexity low, making it suitable as a baseline reference.

3.1.2 Training configuration

The model is compiled using the Adam optimizer [17] with a learning rate of 0.001 and the Sparse Categorical Cross-Entropy loss function, which is appropriate for multi-class classification problems with integer-encoded labels [15]. Training is performed using the `fit` method provided by Keras [19]. During training, classification accuracy on the validation set is monitored in order to track the learning dynamics and detect potential overfitting.

3.2 Model B: tuned CNN

Model B extends the baseline architecture introduced in Model A with the objective of exploring the impact of key architectural and training choices on model behavior. Rather than introducing a substantially different network design, this model adopts a controlled experimental approach in which architectural depth, feature extraction capacity, and optimization parameters are varied within a predefined search space.

In line with the exploratory perspective outlined in the project introduction, Model B incorporates an explicit grid search procedure over a limited set of hyperparameters. This approach enables a structured investigation of how changes in the number of convolutional blocks, the base number of filters, and the learning rate affect training dynamics, while preserving the overall convolutional structure of the baseline model. Additional regularization mechanisms, including batch normalization and dropout, are introduced to improve training stability and robustness.

3.2.1 Architectural structure

The architectural structure of Model B follows the same overall design principles as the baseline CNN, while allowing for greater flexibility in depth and regularization. All layers are implemented using the Keras Layers API [14].

- **Input layer**: RGB images of fixed spatial resolution are provided through an explicit input layer [12].
- **Data augmentation**: the data augmentation pipeline introduced in the previous section is applied during training.
- **Convolutional blocks**: the feature extraction stage consists of a variable number of convolutional blocks. Each block includes:
  - a Conv2D layer with 3×3 kernels [5],
  - an optional Batch Normalization layer [4],
  - a ReLU activation function [21],
  - a MaxPooling2D layer for spatial downsampling [18],
  - an optional Dropout layer for regularization [7].
- **Flatten layer**: the final feature maps are flattened into a one-dimensional representation [9].
- **Fully connected layer**: a dense layer with 128 units and ReLU activation [6, 21].
- **Output layer**: a dense layer with three neurons and softmax activation produces class probability estimates [6].

3.2.2 Hyperparameter tuning

In line with the exploratory perspective outlined in the project introduction, Model B introduces an explicit hyperparameter tuning phase based on a grid search strategy. The search space is deliberately kept limited in order to enable a controlled investigation of the impact of architectural depth, feature extraction capacity, and optimization dynamics.

The grid search explores combinations of the following hyperparameters:

- number of convolutional blocks: $\{2, 3\}$,
- base number of convolutional filters: $\{32, 64\}$,
- learning rate of the Adam optimizer: $\{10^{-3}, 3 \times 10^{-4}\}$.

This results in a total of eight distinct configurations, all trained using the same data splits and random seed to ensure a fair comparison. For each configuration, the best validation accuracy observed during training is recorded. The complete set of results is stored in a structured CSV file for reproducibility and further analysis.

Among the evaluated configurations, the best-performing setup is:

- number of convolutional blocks: X,
- base number of filters: Y,
- learning rate: Z,

which achieved a validation accuracy of A and a validation loss of B. This configuration is therefore selected to train the final instance of Model B.

3.2.3 Training configuration

The final instance of Model B is trained using the Adam optimizer [3], with the learning rate set according to the best configuration identified during the hyperparameter tuning phase. Training is performed for a maximum of 15 epochs.

Early stopping is employed as a regularization strategy to prevent overfitting by monitoring the validation loss during training [27, 8]. When no improvement is observed for a fixed number of epochs, training is halted and the model weights corresponding to the lowest validation loss are restored.

As in the previous models, training dynamics are monitored using loss and classification accuracy on the validation set.

3.3 Model C: complex CNN

Model C represents a further evolution of the tuned CNN introduced in Model B and is designed to explore a broader architectural and regularization space. While Model B focuses on a controlled grid search over a limited set of hyperparameters, Model C adopts a more flexible and expressive design, both in terms of network architecture and optimization strategy.

In particular, Model C introduces additional regularization mechanisms, deeper convolutional configurations, and a random search strategy for hyperparameter exploration. This model is intended to assess whether increased architectural expressiveness and stochastic hyperparameter sampling can further improve performance beyond the tuned configuration identified in Model B.

3.3.1 Architectural structure

Model C is implemented using the Keras Functional API, which allows for greater flexibility in defining complex architectures compared to the Sequential paradigm used in Models A and B [10]. The overall convolutional structure is preserved, while several architectural enhancements are introduced.

- **Input layer**: RGB images of fixed spatial resolution are provided through an explicit input layer.
- **Data augmentation**: the same augmentation pipeline used in previous models is applied during training.
- **Convolutional blocks**: the feature extraction stage consists of a variable number of convolutional blocks ($n \in \{3, 4\}$). Each block includes:
  - a Conv2D layer with 3×3 kernels and He normal initialization [2],
  - optional L2 kernel regularization to penalize large weights [28],
  - an optional Batch Normalization layer,
  - a ReLU activation function,
  - a MaxPooling2D layer for spatial downsampling,
  - an optional SpatialDropout2D layer to regularize entire feature maps [24].
- **Flatten layer**: feature maps are flattened into a one-dimensional representation.
- **Fully connected layer**: a dense layer with a variable number of units ($\{128, 256, 512\}$), optionally regularized via L2 penalties.
- **Output layer**: a dense layer with three neurons and softmax activation produces class probability estimates.

3.3.2 Hyperparameter search

3.3.3 Hyperparameter search

Unlike Model B, which relies on a grid search strategy, Model C adopts a random search approach to explore a higher-dimensional hyperparameter space [1]. Random search is particularly suitable in this setting, as the number of tunable hyperparameters increases and exhaustive exploration would become computationally prohibitive.

A total of 12 random trials is performed. For each trial, a single configuration is sampled by randomly selecting architectural and regularization parameters, as well as the learning rate of the Adam optimizer. In particular, the learning rate is sampled log-uniformly over a predefined range, allowing the exploration of multiple orders of magnitude.

The following hyperparameters are included in the random search:

- number of convolutional blocks,
- base number of convolutional filters,
- number of units in the dense layer,
- spatial dropout rate in convolutional blocks,
- dropout rate in the classification head,
- L2 regularization strength,
- learning rate of the Adam optimizer.

For each sampled configuration, the best validation accuracy and the corresponding validation loss observed during training are recorded. In case of ties in validation accuracy, the configuration with the lowest validation loss is selected.

The complete set of results for all trials is stored in a CSV file to ensure reproducibility and transparency.

Among the evaluated configurations, the model with four convolutional blocks, 64 base filters, 256 units in the dense layer, and a learning rate of approximately 3.8×10^{-5} achieves the highest validation accuracy (0.982), together with the lowest validation loss. This configuration is therefore selected to train the final instance of Model C, whose performance is analyzed in the evaluation section.

3.3.4 Training configuration

Model C is trained using the Adam optimizer, with the learning rate selected according to the best configuration identified during the random search phase. Training is performed for a maximum of 15 epochs.

Early stopping is employed to prevent overfitting by monitoring the validation loss. In addition, a learning rate scheduling strategy is introduced through the ReduceLROnPlateau callback, which automatically reduces the learning rate when the validation loss stops improving [32, 20].

As in the previous models, training dynamics are monitored using loss and classification accuracy on the validation set.

4 Evaluation and analysis

4.1 Experimental results

4.2 Model comparison

4.3 Discussion

5 Generalization test

6 Conclusions

References

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[3] Keras Team. Adam optimizer. `https://keras.io/api/optimizers/adam/`, 2024.

[4] Keras Team. Batchnormalization layer. `https://keras.io/api/layers/normalization_layers/batch_normalization/`, 2024.

[5] Keras Team. Conv2d layer. `https://keras.io/api/layers/convolution_layers/convolution2d/`, 2024.

[6] Keras Team. Dense layer. `https://keras.io/api/layers/core_layers/dense/`, 2024.

[7] Keras Team. Dropout layer. `https://keras.io/api/layers/regularization_layers/dropout/`, 2024.

[8] Keras Team. Early stopping. `https://keras.io/api/callbacks/early_stopping/`, 2024.

[9] Keras Team. Flatten layer. `https://keras.io/api/layers/reshaping_layers/flatten/`, 2024.

[10] Keras Team. The functional api. `https://keras.io/guides/functional_api/`, 2024.

[11] Keras Team. Image augmentation layers. `https://keras.io/api/layers/preprocessing_layers/image_augmentation/`, 2024.

[12] Keras Team. Input layer. `https://keras.io/api/layers/core_layers/input/`, 2024.

[13] Keras Team. Keras image data loading api. `https://keras.io/api/data_loading/image/`, 2024.

[14] Keras Team. Keras layers api. `https://keras.io/api/layers/`, 2024.

[15] Keras Team. Keras loss functions. `https://keras.io/api/losses/`, 2024.

[16] Keras Team. Keras models api. `https://keras.io/api/models/`, 2024.

[17] Keras Team. Keras optimizers. `https://keras.io/api/optimizers/`, 2024.

[18] Keras Team. Maxpooling2d layer. `https://keras.io/api/layers/pooling_layers/max_pooling2d/`, 2024.

[19] Keras Team. Model training with `fit`. `https://keras.io/api/models/model_training_apis/`, 2024.

[20] Keras Team. Reduce learning rate on plateau. `https://keras.io/api/callbacks/reduce_lr_on_plateau/`, 2024.

[21] Keras Team. Relu activation function. `https://keras.io/api/layers/activations/`, 2024.

[22] Keras Team. Sequential model. `https://keras.io/api/models/sequential/`, 2024.

[23] Keras Team. The sequential model guide. `https://keras.io/guides/sequential_model/`, 2024.

[24] Keras Team. Spatialdropout2d layer. `https://keras.io/api/layers/regularization_layers/spatial_dropout2d/`, 2024.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*, 2015.

[26] TensorFlow Developers. Data augmentation for images. `https://www.tensorflow.org/tutorials/images/data_augmentation`, 2024.

[27] TensorFlow Developers. Earlystopping callback. `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping`, 2024.

[28] TensorFlow Developers. L2 regularization. `https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L2`, 2024.

[29] TensorFlow Developers. Making preprocessing layers part of the model. `https://www.tensorflow.org/tutorials/images/data_augmentation#option_1_make_the_preprocessing_layers_part_of_your_model`, 2024.

[30] TensorFlow Developers. Optimizing data input pipeline: caching and prefetching. `https://www.tensorflow.org/guide/data_performance`, 2024.

[31] TensorFlow Developers. Optimizing data input pipeline: parallel mapping. `https://www.tensorflow.org/guide/data_performance#parallel_mapping`, 2024.

[32] TensorFlow Developers. Reducelronplateau callback. `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau`, 2024.

[33] TensorFlow Developers. Rescaling layer. `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling`, 2024.

[34] TensorFlow Developers. tf.keras.preprocessing.image_dataset_from_directory. `https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory`, 2024. Accessed: 2025.

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.