Rock, Paper, Scissors: CNN for Image Classification

Greta Zehnder

February 16, 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.

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.

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 [24] 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.

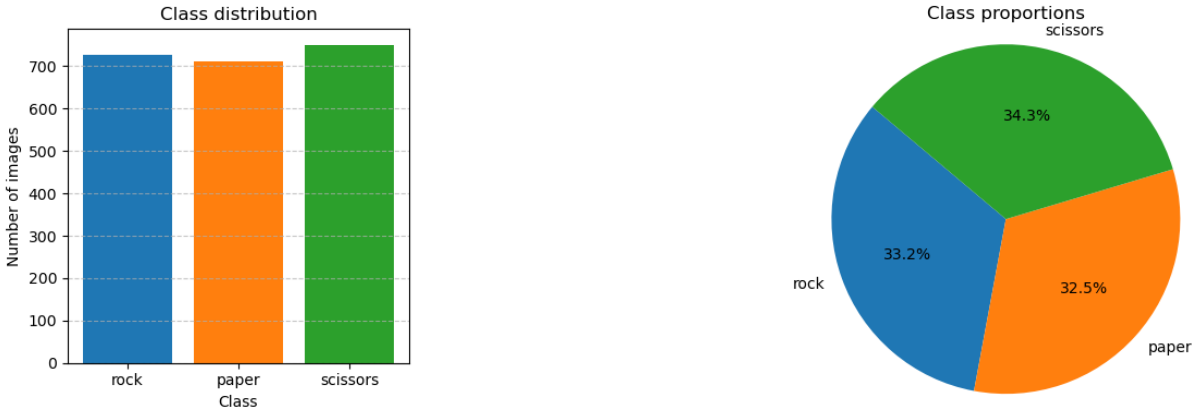


Figure 1: Class distribution (left) and class proportions (right) of the Rock-Paper-Scissors dataset.

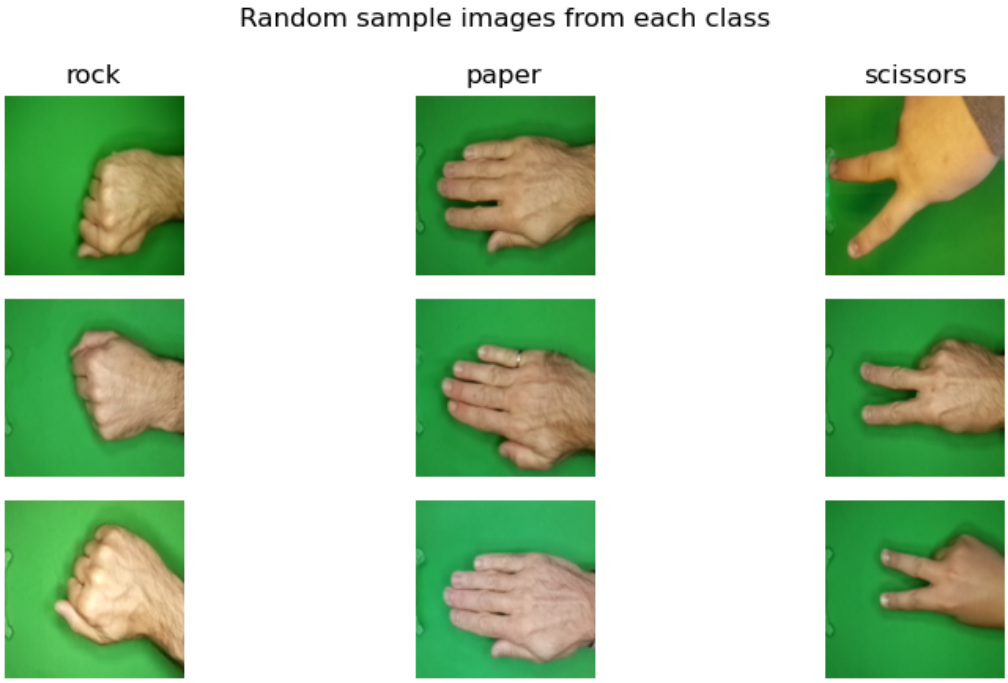


Figure 2: Randomly selected sample images from each class of the Rock-Paper-Scissors dataset.

2.2 Preprocessing

2.2.1 Train, validation and test splitting

The dataset was split into training, validation, and test subsets before model training to enable both hyperparameter tuning and an unbiased final evaluation, with the splitting procedure applied independently to each class using a fixed random seed for reproducibility. In particular, 15% of the images were assigned to the validation set, 15% to the test set, and the remaining samples were used for training, preserving the class distribution across all subsets. The resulting data were then organized into a structured directory hierarchy with separate folders for each split and class, to ensure a clear separation between training and evaluation data.

2.2.2 Normalization

Image normalization was integrated into the data input pipeline by rescaling pixel values from the original $[0, 255]$ range to $[0, 1]$ using the `Rescaling(1./255)` layer provided by the TensorFlow/Keras API [33], so that all images were fed to the network on the same numerical scale. This transformation was applied consistently to the training, validation, and test sets, ensuring that the model received comparable inputs during both training and evaluation and helping maintain stable optimization.

The datasets were created using the `image_dataset_from_directory` utility [34, 13], and normalization was implemented through a mapping operation within the `tf.data` pipeline, which allowed parallel execution and improved data loading efficiency [31]. In addition, caching and prefetching were used to reduce input latency and keep the data flow efficient during both training and testing [30].

2.2.3 Data augmentation

Data augmentation was introduced to improve model robustness and reduce overfitting by exposing the network (during training) to slightly modified versions of the input images while preserving the original class label. In this project, augmentation was implemented using Keras

image preprocessing layers [11], following the workflow recommended in the official TensorFlow documentation [26]: a dedicated augmentation module was defined as a small preprocessing pipeline including random horizontal flipping, small random rotations, and random zoom operations, which are well suited for hand-gesture images as they reflect realistic variations in pose and framing (such as small changes in orientation or distance from the camera) without altering the meaning of the gesture. These transformations were applied only to the training set, whereas the validation and test sets were kept unchanged, so that the evaluation metrics reflect performance on the original data distribution.

The augmentation layers were integrated directly into the model architecture, meaning that they are active during training but automatically disabled during inference, ensuring consistent behavior at deployment while still increasing data variability during learning [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 structure, which allows layers to be stacked linearly in a transparent and interpretable way [15, 21, 22]. The purpose of this model is to define a simple yet meaningful reference architecture for the Rock–Paper–Scissors image classification task, so that more advanced models can later be compared against it in a clear, consistent and structured manner.

3.1.1 Architectural structure

The architecture of Model A follows a standard convolutional structure commonly used in image classification problems and is summarized below.

- **Input layer**: RGB images with fixed spatial dimensions are provided to the network through an explicit input layer [12], which clearly defines the expected shape of the data and ensures compatibility with the subsequent layers.
- **Data augmentation**: the augmentation pipeline described earlier is applied directly after the input layer (and only during training), introducing small geometric variations that increase data diversity without modifying the class label.
- **First convolutional block**: a Conv2D layer with 32 filters and 3×3 kernels is used to extract low-level visual patterns (such as edges and simple textures), followed by a MaxPooling2D layer with a 2×2 window, which reduces the spatial resolution by downsampling the feature maps and retaining the strongest activations within each local region. ReLU activation functions are applied to introduce non-linearity into the model [5, 17, 20].
- **Second convolutional block**: thanks to a second Conv2D layer, the number of filters is increased to 64, allowing the network to learn more complex and abstract representations, and is again followed by a 2×2 max-pooling operation that further reduces the spatial dimensions by a factor of two while preserving the most informative features.
- **Flatten layer**: the resulting feature maps are then transformed into a one-dimensional vector through a flattening operation, enabling the transition from convolutional feature extraction to fully connected processing [9].
- **Fully connected layer**: a dense layer with 128 units and ReLU activation combines the extracted features into a compact internal representation, helping the model integrate information across all spatial locations [6].

- **Output layer**: the final dense layer contains three neurons with softmax activation, to produce a normalized probability distribution over the three target classes (rock, paper, and scissors).

The use of two convolutional blocks allows the network to progressively build hierarchical representations, where earlier layers focus on simple visual structures while deeper layers combine them into more informative and abstract patterns, a behavior that has been widely observed in convolutional architectures and is discussed, for instance, in the VGG network study by Simonyan and Zisserman [25]. At the same time, keeping the architecture relatively shallow (i.e., limiting it to two convolutional stages) helps to control model complexity and makes it suitable as a baseline reference for comparison.

3.1.2 Training configuration

The model is compiled using the Adam optimizer [16] with a learning rate of 0.001 and the Sparse Categorical Cross-Entropy loss function, which is appropriate for multi-class classification tasks with integer-encoded labels [14]. Training is performed for 10 epochs using the `fit` method provided by Keras [18], while classification accuracy on the validation set is monitored throughout the process in order to observe the learning dynamics and identify possible signs of overfitting.

3.2 Model B: tuned CNN

Model B extends the baseline architecture introduced in Model A with the objective of systematically evaluating how a restricted set of architectural and optimization choices affects model performance. Rather than proposing a substantially different network design, Model B preserves the overall convolutional structure of the baseline and introduces a controlled tuning phase over depth, capacity, and learning rate.

3.2.1 Architectural structure

Model B follows the same general design principles as Model A, while allowing for variable depth and feature extraction capacity.

- **Input layer**: once again, RGB images of fixed spatial resolution are provided through an explicit input layer.
- **Data augmentation**: the same data augmentation pipeline introduced for Model A is applied during training.
- **Convolutional blocks**: the feature extraction stage consists of $n_$blocks convolutional blocks. Each block includes:
  - a Conv2D layer with 3×3 kernels and `same` padding [5],
  - a ReLU activation function [20],
  - a MaxPooling2D layer for spatial downsampling [17].

  The number of filters is controlled through a *base* value, `base_filters`, and is doubled after each block (i.e., $f, 2f, 4f, \dots$). Therefore, `base_filters` determines the initial channel width, while deeper blocks progressively increase representational capacity.
- **Flatten layer**: identically to Model A, the final feature maps are flattened into a one-dimensional representation.

- **Fully connected layer**: the classifier head, as before, includes a dense layer, this time with 256 units and ReLU activation, increasing head capacity compared to Model A.
- **Output layer**: again, a dense layer with three neurons and softmax activation, to produce class probability estimates.

3.2.2 Hyperparameter tuning

Model B also introduces an explicit hyperparameter tuning phase based on a grid search strategy, a selection manually conducted and intentionally limited to enable a controlled analysis of the effects of depth, capacity, and optimization dynamics.

The grid search explores combinations of the following hyperparameters:

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

The choice of $\{2, 3\}$ convolutional blocks directly builds on Model A, which uses two convolutional blocks: including 2 blocks enables a direct comparison against the baseline depth, while 3 blocks represent a moderate increase in depth to assess whether additional hierarchical feature extraction improves performance.

The learning rate values include the baseline setting used in Model A (10^{-3}) and a smaller alternative (3×10^{-4}) to evaluate whether a reduced step size improves optimization stability and generalization [3].

In total, eight configurations are evaluated under the same data splits and fixed random seed, and, for each configuration, the best validation accuracy observed during training is recorded. In case of ties in validation accuracy, the configuration with the lowest validation loss is selected. All results are stored in structured JSON and CSV files for reproducibility.

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

- number of convolutional blocks: 3,
- base number of filters: 32,
- learning rate: 10^{-3}.

This configuration achieved a validation accuracy of 0.9816 and a validation loss of 0.0442, and is therefore selected to train the final instance of Model B.

3.2.3 Training configuration

All grid search runs are trained with a fixed dense layer size (256 units) and a maximum of 20 epochs, while early stopping is employed as a regularization strategy by monitoring the validation loss [27, 8]. Training is halted if no improvement is observed for three consecutive epochs, and the model weights corresponding to the lowest validation loss are restored.

The final instance of Model B is then retrained using the best hyperparameter configuration identified during tuning, using the Adam optimizer with the selected learning rate, and training dynamics are monitored through validation loss and classification accuracy.

Model B extends the baseline architecture introduced in Model A with the objective of understanding and investigating the impact of selected architectural and optimization hyperparameters on model performance: rather than introducing a fundamentally different network design, this model adopts a controlled experimental approach in which model depth, feature extraction capacity, and learning rate are varied within a predefined search space. This approach enables a transparent comparison between configurations while preserving the overall convolutional structure of the baseline model.

3.3 Model C: random-search CNN

Model C represents the most flexible and regularized architecture within the experimental progression: in fact, while Models A and B relied on structured grid exploration, this last model adopts a stochastic hyperparameter optimization strategy based on random search. Random search is often more efficient than grid search when only a subset of hyperparameters significantly affects performance, especially in mixed discrete–continuous search spaces.

Compared to Model B, Model C introduces:

- an additional convolutional block (up to four),
- stronger regularization mechanisms (including L2 regularization [28] and SpatialDropout2D [23]),
- optional Batch Normalization [4] layers to improve training stability,
- an improved weight initialization strategy (He normal initialization [2]),
- log-uniform sampling of the learning rate,
- learning-rate scheduling during training (via ReduceLROnPlateau [19, 32]).

3.3.1 Architectural structure

Differently from the previous models, Model C is implemented using the Keras Functional API [10], to enable a more explicit and flexible definition of the computation graph.

The network consists of n_{blocks} convolutional blocks, where $n_{\text{blocks}} \in \{3, 4\}$, and, as in Model B, the number of filters is controlled by a base value, `base_filters`, and doubled after each block (i.e., $f, 2f, 4f, 8f$): this progressive widening increases representational capacity as spatial resolution decreases.

Each convolutional block includes:

- **Conv2D layer**: kernels are initialized using He normal initialization, which is particularly suitable when ReLU activations are employed, as it helps to maintain a stable variance of activations across layers and supports efficient gradient propagation during training. An optional L2 kernel regularizer is also applied, depending on the hyperparameter `l2_reg`, introducing a penalty on large weight values in order to reduce overfitting and encourage smoother and more stable solutions.
- **Batch Normalization (optional)**: when enabled, Batch Normalization is applied immediately after convolution. In this case, the convolution is configured with `use_bias=False`, choice motivated by the fact that Batch Normalization introduces a learnable shift parameter, rendering the explicit bias term redundant.
- **ReLU activation** followed by **MaxPooling2D** for spatial downsampling, as seen in Model A and Model B.
- **SpatialDropout2D (optional)**: applied after pooling to randomly drop entire feature maps, and encouraging robustness in convolutional representations, this mechanism is introduced to strengthen regularization within the feature extraction stage, reducing the risk of overfitting by preventing the network from relying excessively on specific convolutional channels.

After the convolutional backbone, feature maps are flattened and passed to a fully connected layer with tunable size (`dense_units` $\in \{256, 512\}$). An optional Dropout layer is applied in the classification head (`dropout_head`) [7], and the final layer is a softmax classifier over the three output classes.

3.3.2 Hyperparameter tuning via random search

Hyperparameters are sampled randomly over the following search space:

- number of convolutional blocks: $\{3, 4\}$,
- base number of filters: $\{32, 64\}$,
- dense units: $\{256, 512\}$,
- SpatialDropout2D rate: $\{0.0, 0.10\}$,
- dropout rate in the classification head: $\{0.3, 0.4, 0.5\}$,
- L2 regularization coefficient: $\{0.0, 10^{-4}\}$,
- learning rate: sampled log-uniformly in $[10^{-5}, 10^{-3}]$.

Sampling the learning rate from a log-uniform distribution allows exploration across multiple orders of magnitude while maintaining equal relative probability [1]. Here, the upper bound matches previous models, while the lower bound extends exploration toward smaller step sizes suitable for deeper architectures.

A total of 12 trials are performed, and, for each trial, the best validation accuracy achieved during training is recorded (again, in case of ties in validation accuracy, the configuration with the lowest validation loss is selected). As in the previous model, results are saved as JSON and CSV files.

3.3.3 Training configuration

Each trial is trained for a maximum of 25 epochs (five more than Model B) using, once again, the Adam optimizer, and early stopping is applied by monitoring validation loss with a patience of 3 epochs. When triggered, the model weights corresponding to the lowest validation loss are restored.

Additionally (as mentioned) Model C incorporates a learning-rate scheduling mechanism via the ReduceLROnPlateau callback, meaning that when validation loss stagnates, the learning rate is reduced multiplicatively, enabling finer convergence during later training stages.

3.3.4 Best configuration and analysis

Among the evaluated configurations, the best-performing setup (that corresponds to trial 4) is defined by:

- number of convolutional blocks: 4,
- base number of filters: 64,
- dense units: 256,
- SpatialDropout2D rate: 0.1,
- dropout rate in the classification head: 0.4,
- L2 regularization coefficient: 0.0,
- learning rate: 2.85×10^{-5},
- Batch Normalization: enabled.

This configuration achieved a validation accuracy of 0.9847 and a validation loss of 0.0512.

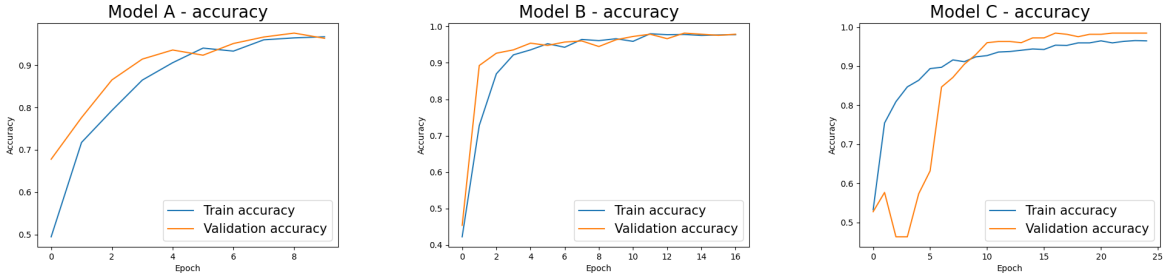
The selected architecture suggests that increased depth (four convolutional blocks) combined with stronger feature capacity (base filters of 64 with progressive doubling) enhances hierarchical feature extraction. Interestingly, L2 regularization was not selected, while both SpatialDropout2D and head dropout were retained, indicating that stochastic regularization mechanisms were more effective than weight penalization in this setting.

Moreover, the optimal learning rate (2.85×10^{-5}) is significantly lower than those explored in previous models. This is consistent with the increased architectural complexity: deeper networks with Batch Normalization and larger representational capacity may benefit from smaller optimization steps to ensure stable convergence and improved generalization.

The final instance of Model C is retrained using this configuration and saved together with the training history and random-search results.

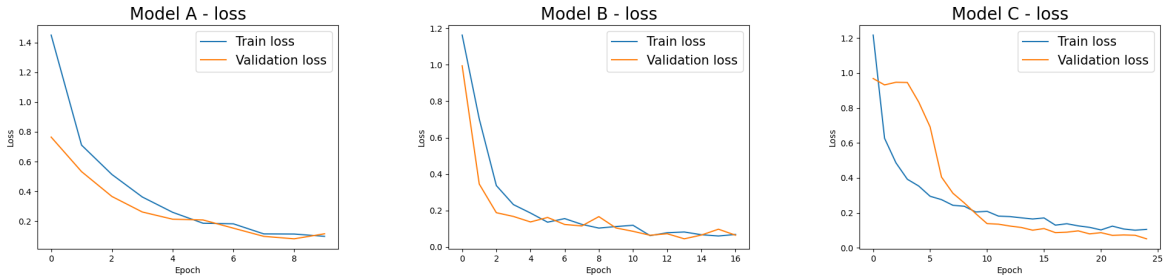
4 Evaluation and comparative analysis

4.1 Training and validation curves



(a) Model A *(b) Model B* *(c) Model C*

Figure 3: Training and validation accuracy curves for the three models.



(a) Model A *(b) Model B* *(c) Model C*

Figure 4: Training and validation loss curves for the three models.

Figures 3 and 4 illustrate the evolution of accuracy and loss during training for all models, highlighting the progressive impact of architectural refinement and optimization strategies.

Model A exhibits smooth and stable convergence: training accuracy increases steadily across epochs, while validation accuracy follows a similar trend with only a small gap between the two curves. The corresponding loss curves decrease consistently, and only a slight rise in validation loss appears in the final epoch, suggesting a mild onset of overfitting. Overall, the baseline architecture demonstrates balanced bias-variance behavior and reliable generalization.

Model B converges faster than the baseline model, with both training and validation accuracy increasing rapidly during the first epochs and reaching high values much earlier than in Model

A. At the same time, validation loss decreases quickly and stabilizes at a lower level compared to the baseline. The training and validation curves remain close to each other throughout the process, which suggests that the model does not suffer from significant overfitting.

Early stopping is activated before reaching the maximum number of epochs, indicating that the model had already reached a stable performance level and that further training would not have led to meaningful improvements. Overall, Model B shows a smoother and more efficient learning process, with improved validation performance and no clear signs of underfitting or overfitting.

Model C shows a more irregular behavior in the first training epochs, where validation accuracy and loss fluctuate more than in the previous models. This is likely due to the increased depth of the network and the stronger regularization mechanisms, which make the optimization slightly more unstable at the beginning. After this initial phase, both training and validation curves become smoother and improve steadily over time, with a small and consistent gap between them.

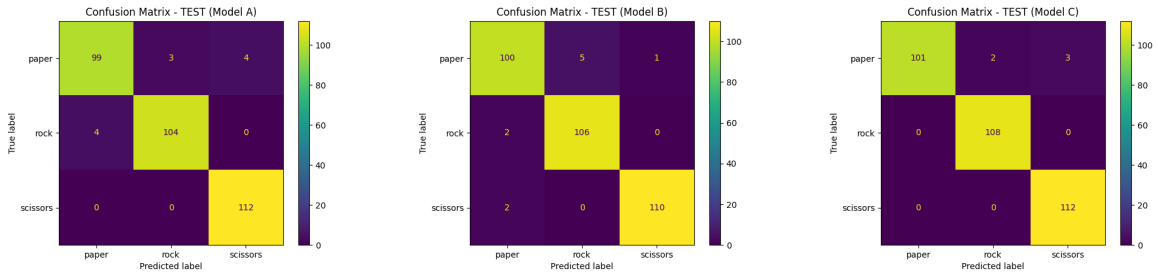
The learning-rate scheduling mechanism helps the model converge more gradually in the later epochs, as visible in the continuous reduction of validation loss. Unlike Model B, early stopping is not activated, since validation loss keeps improving within the fixed number of epochs. Importantly, there are no clear signs of overfitting, as validation performance remains aligned with training performance throughout training. Overall, Model C achieves the highest validation accuracy while maintaining stable generalization behavior.

Taken together, the curves reveal a clear progression: the baseline model provides stable performance, structured hyperparameter tuning improves convergence efficiency and reduces validation loss, and the introduction of random search, deeper architecture, and adaptive learning-rate scheduling yields further, though incrementally smaller, gains in validation accuracy.

4.2 Test set performance

Table 1: Test set performance comparison across models.

Model	Test accuracy	Test loss	Macro F1	Weighted F1
Model A	0.9663	0.1267	0.9658	0.9661
Model B	0.9693	0.1334	0.9690	0.9693
Model C	0.9847	0.0744	0.9845	0.9846



(a) Model A *(b) Model B* *(c) Model C*

Figure 5: Confusion matrices on the test set for all models.

Table 1 summarizes the final performance of all models on the held-out test set. Model A already achieves strong generalization, with accuracy above 96%. Model B provides a slight improvement in classification performance, although with a marginally higher test loss.

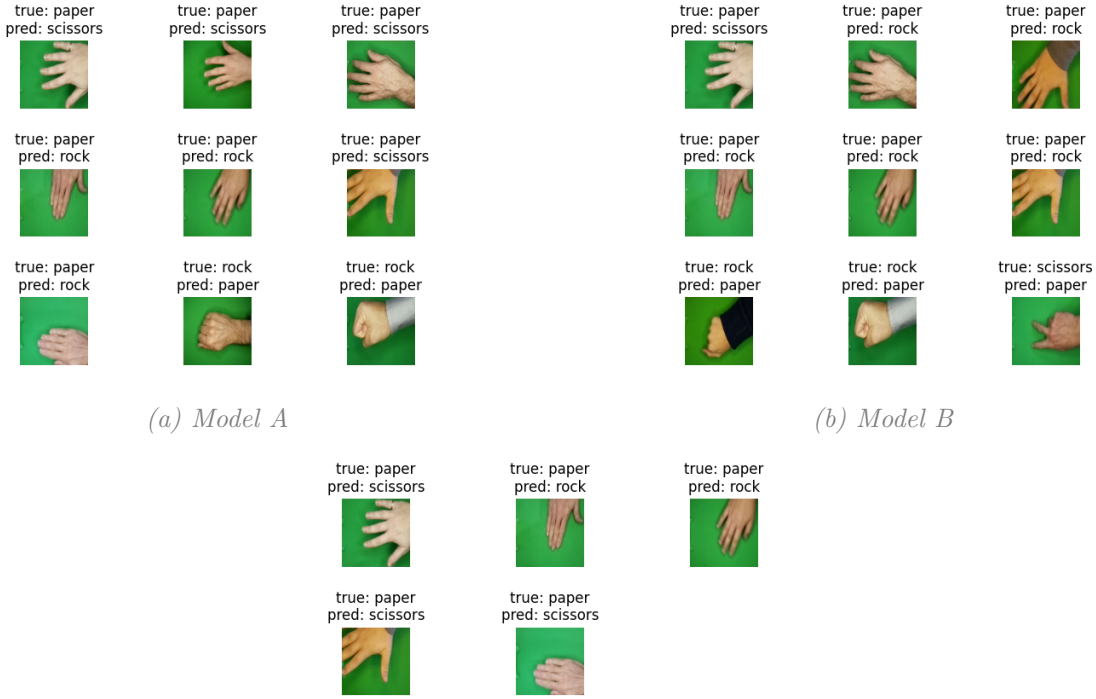
Model C achieves the highest test accuracy (0.9847) and the lowest test loss (0.0744), confirming the effectiveness of deeper architecture, stochastic hyperparameter search, and adaptive learning-rate scheduling.

Figure 5 further highlights this progression: in fact, while Models A and B exhibit minor confusion primarily between paper and rock, Model C shows near-perfect class separation. In particular, the classes rock and scissors are classified without error, and only a few paper samples are misclassified.

These results demonstrate that architectural refinement and advanced optimization strategies yield consistent improvements in generalization performance, with Model C providing the most robust, accurate and reliable predictions on unseen data.

4.3 Error analysis and misclassified examples

Error analysis is conducted exclusively on the held-out test set to ensure an unbiased assessment of generalization performance.



(a) Model A

(b) Model B

(c) Model C

Figure 6: Examples of misclassified test samples for each model. True labels are shown above each image and predicted labels below.

Model B. Figure 6 (Model A) displays up to nine representative misclassified test samples. As we have observed in the confusion matrix, Model A misclassifies seven instances of *paper* and four instances of *rock*, while all *scissors* samples are correctly classified. The examples plotted show how errors involve the class *paper*, which is sometimes predicted as *scissors* when the fingers are partially separated, or as *rock* when the hand is slightly rotated or partially closed. The misclassified *rock* samples are instead predicted as *paper*, typically in cases where lighting or hand orientation reduces the visual prominence of the closed fist.

Model B. Figure 6 (Model B) shows nine (of the ten total errors) representative misclassified test samples. Although the overall number of errors is comparable to Model A, their distribution changes slightly.

Several misclassifications still involve the class *paper*, which is often predicted as *rock* and occasionally as *scissors*, confirming that flat-hand configurations remain visually ambiguous under certain orientations. In addition, two *rock* samples are predicted as *paper*, and one *scissors* sample is also classified as *paper*.

Compared to Model A, the confusion appears more evenly distributed across classes rather than concentrated primarily on *paper*.

Model C. Figure 6 (Model C) shows a smaller and more concentrated set of errors compared to the previous models: all misclassified samples belong to the class *paper*, which is occasionally predicted as either *rock* or *scissors*, while no instances of *rock* or *scissors* are incorrectly classified.

This suggests that the model has learned to clearly distinguish between closed-fist and two-finger configurations, and that the remaining ambiguity is limited to certain flat-hand gestures where finger spread, orientation, or lighting conditions make the visual pattern less distinct.

These observations further confirm that, as the qualitative inspection of the errors aligns with the test metrics, Model C achieves the most stable and reliable behavior among the three architectures.

4.4 Discussion

While Model C achieves the highest generalization performance, these improvements come at the cost of increased computational complexity, as deeper architectures, random hyperparameter search, and adaptive learning-rate scheduling significantly raise training time and resource consumption compared to the baseline model. In particular, the introduction of additional convolutional blocks, Batch Normalization, and multiple regularization mechanisms increases both the number of trainable parameters and the overall optimization overhead, making the model less suitable for constrained environments despite its superior accuracy.

Further performance gains could potentially be obtained through more extensive hyperparameter optimization, larger search spaces, data augmentation strategies of greater diversity, or transfer learning from pre-trained convolutional backbones; however, such approaches were not explored in this study in order to maintain a controlled experimental setup and reasonable computational cost. Therefore, the results highlight a practical trade-off between architectural sophistication and efficiency, suggesting that Model B may represent a balanced compromise when computational resources are limited, while Model C is preferable when maximum predictive performance is required.

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 loss functions. https://keras.io/api/losses/, 2024.

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

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

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

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

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

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

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

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

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

[24] Laurence Moroney. Rock paper scissors dataset. https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors, 2019.

[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.