CNNs for Rock-Paper-Scissors Classification

Nucera Chiara

July 2025

Abstract

In this project, the goal was to build and evaluate CNN models capable of recognizing hand gestures belonging to three different classes. The project explores three CNN architectures of increasing complexity to understand how network depth and design choices impact performance and generalization and how the complexity of a model has to be proportionate to the task and to the dataset. To improve the model's ability to generalize to new images, data augmentation techniques were applied. Furthermore, automated hyperparameter optimization using Optuna was integrated to find the most effective training configuration. Achieving a reasonable training time has been prioritized on maximizing accuracy.

1 Introduction

In this project, a Convolutional Neural Network (CNN) was developed to classify hand gestures representing Rock, Paper, and Scissors. Three different CNN architectures were implemented, progressively increasing in complexity, and evaluated on a balanced dataset of 2188 labeled images. The goal was to identify an architecture that offers the best trade-off between accuracy, generalization, and computational efficiency, but prioritizing achieving a reasonable training time has been prioritized on maximizing accuracy. The dataset was preprocessed using standard transformations, and data augmentation was applied to improve model robustness. After the training phase and the choice of the best-performing (for this dataset) model, hyperparameter tuning was performed using Optuna with early stopping and pruning to efficiently optimize learning rate, batch size, dropout, and optimizer choice. Among the three models, SecondCNN achieved the best overall performance with a test accuracy of 98.5%, outperforming both the simpler and more complex models. The final model was also evaluated on a custom test set of images to assess generalization. Results show that a moderately complex CNN, combined with systematic hyperparameter tuning, can deliver excellent classification performance with minimal overfitting and efficient training time.
The model was built using Python 3 with the PyTorch library.

2 Theory: CNNs

The Convolutional Neural Networks (CNNs) are deep learning models designed to process data with a grid-like topology, such as images. They work by detecting patterns in the images, recognizing object, classes and categories and thus classifying the input data.

2.0.1 Main components of the CNNs

- Convolutional layers: application of filters (kernels) that slide over the image to detect the patterns, such as edges, textures, or shapes. The result of each filter is a feature map which highlights the locations where the pattern has been found in the input image. When using PyTorch, they are implemented with the self.conv = nn.Conv2d function(input channels, number of feature maps to produce, kernel size). The padding option is also applied to avoid losing information at the borders and preserve the spatial dimensions of the original image.
- Pooling layers: to reduce the spatial dimensions of the feature maps. In this case, for the implementation of the three CNNs, MaxPooling was used to select a maximum value from a group of neighboring pixels and extract the most important information from them.
- Activation functions: to introduce non-linearity into the neural network, enabling it to learn complex patterns. In this work, the ReLu (Rectified Linear Unit) activation function was applied.
- Fully connected layers: used as the last layer to make the prediction basing on the information extracted from the previous ones. The number of classes over which the prediction is made has to be specified. In this case, three classes: Rock, Paper, Scissors.

3 Dataset Description

The dataset (Rock-Paper-Scissors dataset available on Kaggle with author Julien de la Bruère-Terreault at the following link:
https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors) that was used for training and evaluating the three CNNs (Convolutional Neural Networks) is an image dataset containing 2188 images divided in three classes:

1. Rock (726 images)
2. Paper (712 images)
3. Scissors (750 images)

The class were mapped as 'paper': 0, 'rock': 1, 'scissors': 2, therefore the Counter function returned the values Counter(2: 750, 1: 726, 0: 712). The number of data between the classes is balanced, with a difference of only 38 images between the largest and the smallest class.

3.1 Exploratory Data Analysis

To verify the quality of the images in the dataset, it was inspected both manually and with the code. Regarding the coding part two operations were performed: printing a batch of images and checking the presence of black images, that is the equivalent of verifying to have the sum of pixels different from zero. The quality of the dataset was adequate, all the images presented a uniform green background with the hands generally visible even if not always centered (but not significantly decentralized) and coherent to their belonging class. Have not been found corrupted, black or with insufficient quality images.

3.1.1 Training, validation and test set:

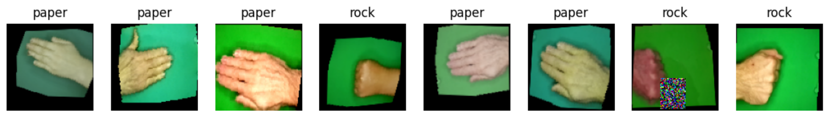
The dataset was divided in three distinct subsets: training, validation and test. The training set was the 70% of the dataset used to train the model; during the training part the CNNs iteratively adjusted their internal parameters to minimize the loss function on this dataset. As dimension for the validation set, 15% of the dataset was chosen and, of course, the same value was also selected for the test set. The validation set was used in the hyperparameter tuning part as unbiased metric to compare different hyperparameter configurations, but also to monitor the model's performance during training calculating loss and accuracy on the validation set at each epoch to understand if the model was continuously improving or (sign of overfitting) if it was the case to stop the training in a specific epoch (early stopping). Lastly, the test set was used to evaluate the trained model's performance after all training and hyperparameter tuning are complete.

Choice of loss function and optimizer for training the CNNs: All CNNs were trained using the CrossEntropyLoss function, which is the standard and appropriate loss function for multi-class classification tasks such as Rock-Paper-Scissors gesture recognition. As the optimizer, Adam optimizer (Adaptive Moment Estimation) was chosen because it is an appropriate optimizer for deep learning tasks, especially for image classification with CNNs, due to its efficiency and effectiveness. During the hyperparameter tuning phase, different alternative of optimizers were explored: Adam, SGD and RMSprop.

3.1.2 Transformations of the data:

For the training and validation process, different transformations were applied on each image: starting from the resizing operation to be sure to have all the data (training, validation and test set), of 100x100 pixels, then applying different **data augmentation techniques** but only on the training set (**ColorJitter**, **RandomHorizontalFlip**, **RandomRotation(20)**, **RandomAffine**) to increase the variety of data and build a more robust and more able to generalize model and, lastly, converting the images into Pytorch tensor and normalizing them using standard mean and standard deviation values. The data augumentation techniques consist in applying random transformations on the data to increase the variety of the data and build a more robust model which learns how to recognize a not fully centered or a rotate picture, avoiding the model to memorize the images. Add addition, a resize with 128x128 pixels was tried but the trade-off between computational time and accuracy didn't exhibit a real advantage for this modification: having 100x100 pixels guaranteed more that a two-times saving in time without any loss in accuracy or in the other metrics; the model was, furthermore, more manageable in terms of RAM and CPU. All this elements supported the choice of 100x100 pixels to achieve a reasonable training time.

After the visualization, a batch of the images in the training set was displayed with the function def **show_images**.



4 Explanation of the code

4.1 Functions (def)

The following functions (def) are implemented in the code: `show_images`, `train_model`, `evaluate_model`, `plot_curves`, `show_misclassified` and, for the hyperparameter tuning part, `objective`, `plot_best_trials` and `plot_confusion_matrix`.

The **train_model** function is required to train the neural networks over multiple (10) epochs while monitoring its performance on the validation set. With this function, the model was trained on the training set and validated on a separate validation set after each epoch. Train losses, validation losses, train accuracy and validation accuracy are calculated and printed.

The **evaluate_model** function performs an evaluation of the trained neural network on a test dataset. It first sets the model to evaluation mode to ensure layers like dropout and batch normalization behave correctly during inference. Then, it disables gradient tracking to improve efficiency (because no learning occurs during evaluation). The function iterates over the test data batches, moves the images to the device, and obtains predictions by passing the images through the model. Then the predicted and true labels are collected to compute the following classification metrics: accuracy, precision, recall, and F1 score.

- Accuracy = to measure the overall correctness of the predictions, it is in fact calculated with the number of corrected predictions divided by the total number of predictions. It is useful in this case because the classes are balanced.

- Precision = to evaluate the correctness of the model predictions over a specific class. It is calculated with the number of true positives divided but the sum of true positives and false positives.
- Recall = to assess how many actual positives were correctly identified. The recal measure is calculated with the number of true positives divided by the sum of the number of true positives and the number of false negatives.
- F1 score = to balance the importance of precision and recall, since precision does not consider false negatives and recall does not count false positives.

Using these metrics, it is possible to understand how well the model is generalizing to unseen data.

The `plot_curves` function plots the curves for the previously (during the training and validation loop) calculated losses and accuracies for training and validation sets.

The `show_misclassified` function contains the code for asking the model to predict the labels for the test or validation images and showing a selection of the misclassified images with the incorrect and the true labels. The purpose of the implementation of `show_misclassified` is to identify patterns in the model's mistakes.

The `plot_confusion_matrix` function to plot the confusion matrix to evaluate the performances of the best model on the test set.

4.2 CNN architecture

Three CNNs have been implemented, they have increasing complexity:

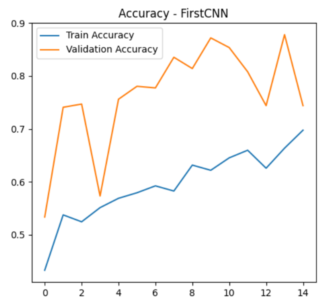
1. FirstCNN: the simplest model, is made by one convolutional layer (conv1 with 3 input channels, 16 output channels(filters), 3x3 kernel, and padding = 1), max pooling layer (2x2 kernel to reduce the spatial dimension from 100x100 to 50x50) and a single fully connected layer (fc1) mapping the flattened output of the pooling layer (16 * 50 * 50) and outputting three classes. It is the simplest and therefore the baseline model.
2. SecondCNN: a more complex model with three convolutional layers with increasing output channels (32, 64, 128) to learn more complex features, three batch normalization layers to help stabilize and accelerate training, max pooling layers reduce spatial dimensions further (100x100 - 50x50 - 25x25 - 12x12), a dropout layer with a rate of 0.3 to reduce overfitting, and two fully connected layers.
3. ThirdCNN: the more complex model of the three. It has four convolutional layers (256 filters), batch normalization after each convolution, a dropout after the convolutional layers (that is a dropout specifically on 2D feature maps) and one after each one of the three fully connected layers which increase the model's ability to learn the relationships from the extracted

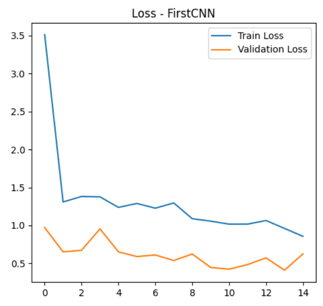
features. The spatial dimension after the poolings will be of 6x6 (100x100 - 50x50 - 25x25 - 12x12 - 6x6).

5 Training Process, 15 epoch

Choice of number of epochs: To choose the best number of epochs for this model, 10, 15 and 20 were tried. Just observing the results with 20 epochs, it was noticeable that the validation accuracy started improving rapidly up to epoch 10-13, peaked around epoch 13-17 and then started to fluctuate but without improving much. In regarding of computational time, 10 epochs showed a significantly faster training period than 20 epochs but with a difference in accuracy of about 1.8%. To capture only the best part of the learning curve avoiding extra training time, the option of 15 epochs was tried and deemed as the best one. Accordingly with the idea of prioritizing a reasonable training time over maximizing accuracy, the early stopping option was also implemented.

1. FirstCNN:
   (a) Training accuracy: increased steadily from 43.3% to 69.8%. Progress slightly fluctuated toward the end, according with the limited learning capacity of the model.
   (b) Training loss: decreased overall from 3.5113 to 0.8566, showing that the model was learning. However, the learning progresses was not consistent: in several epochs the value fluctuates (e.g. 3,4).
   (c) Validation accuracy: fluctuated a lot during the epochs, reaching a peak of 87.8% at epoch 14, but then immediately dropping to 74.4% at epoch 15. This changeableness suggests that the model struggled to generalize.
   (d) Validation loss: dropped from 0.9721 to 0.4106 by epoch 14, then jumped to 0.6251 in epoch 15, indicating mild overfitting in later epochs.
   (e) Test evaluation metrics: **accuracy** 0.7538, **precision** 0.8124, **recall** 0.7763, **F1** 0.7505. The test metrics have decent values according with the fact FirstCNN is a very simple model. The precision value suggests that the predictions were most of the time correct.





2. SecondCNN:

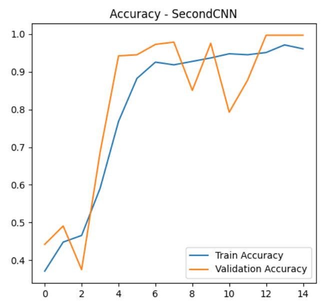
   (a) Training accuracy: increased continuously from 37.1% to 96.1%. The largest improvement occurred between epochs 3 and 7, indicating efficient feature learning after the initial phase.

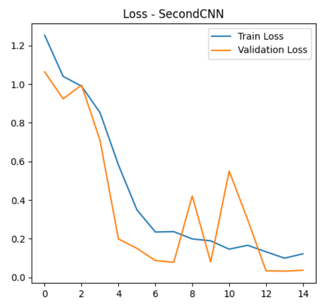
   (b) Training loss: decreased from 1.2528 to 0.1222. The drop was smooth and steady with little to no fluctuations.

   (c) Validation accuracy: continuously increased, had a sudden decrease at epoch 3 but increased significantly in epoch 4 and again in epoch 5. The overall increase was sharp from from 44.2% to 99.7% by epoch 14, then plateaued. The model generalizes extremely well, with minor overfitting (discussed the next section) only visible between epochs 9 and 12.

   (d) Validation loss: dropped sharply to 0.0326 (extreme low value), with small bumps (e.g., at epoch 9 and 11), indicating excellent generalization and minimal overfitting..

   (e) Test evaluation metrics: **accuracy** 0.9848, **precision** 0.9843, **recall** 0.9862, **F1** 0.9951. These metrics demonstrate a significant improvement over FirstCNN, especially across all the four values. SecondCNN is highly accurate and with a good capability to generalize. The recal and F1 scores indicates balanced performances across classes.





3. ThirdCNN: despite being deeper and more complex, performs slightly worse than SecondCNN, suggesting that the added complexity was not beneficial for this dataset (overparameterized).

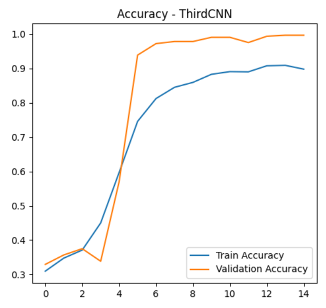
   (a) Training accuracy: started low at 30.9%, increasing faster in the first epochs and then continuing to increase until 89/90%. The gain was slower than in SecondCNN.

   (b) Training loss: reduced from 1.2211 to 0.2826. The curve is smoother after epoch 6, showing gradual improvement in learning.

   (c) Validation accuracy: increased steadily. Started very low and then increased rapidly between epoch 5 (0.5701) and epoch 6 (0.9390), reaching 99.7% in the final epochs. This large jump indicates that the model learned well after sufficient training.

   (d) Validation loss: high value at first (1.0987), then fell dramatically between epoch 4 (1.0923) and 6 (0.3194).

   (e) Test evaluation metrics: **accuracy** 0.9878, **precision** 0.9877, **recall** 0.9888, **F1** 0.9882. These results are really good, with all metrics close to 99%.



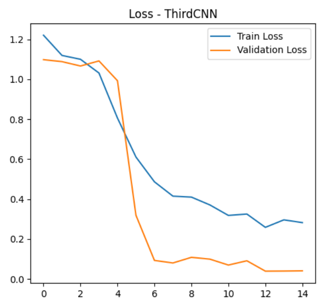


Table: Training and validation values

Values	FirstCNN	SecondCNN	ThirdCNN
T. loss	0.8566	0.1222	0.2826
T. accuracy	0.6976	0.9608	0.8981
V. Loss	0.6251	0.0373	0.0413
V. accuracy	0.7439	0.9970	0.9970

Table: Evaluation on the test set

Values	FirstCNN	SecondCNN	ThirdCNN
Accuracy	0.7538	0.9848	0.9878
Precision	0.8124	0.9843	0.9877
Recall	0.7763	0.9862	0.9888
F1	0.7505	0.9851	0.9882

SecondCNN outperformed both FirstCNN and ThirdCNN, showing strong generalization capabilities. The increased depth, additional convolutional layers, and careful application of batch normalization and dropout layers of ThirdCNN were highly effective for this particular image classification task, however, being the most complex model, ThirdCNN could learn very intricate patterns, becoming more sensitive to overfitting and training instability, especially on a relatively small dataset for which it has too many layers and therefore end up memorizing the training data too well (too many detail, also including noise or background features) but not necessarily generalize better to the new unseen images. While it achieved excellent accuracy, the improvement in accuracy with respect to SecondCNN was really small (0.9878 vs 0.9848), making SecondCNN a better fit: simpler, with an excellent performance and faster, in respect of the computational time constraint.

5.0.1 Overfitting in SecondCNN:

As it is observable from the graphs, the training accuracy and loss for SecondCNN improved steadily, indicating that the model was learning well. However, the validation loss and accuracy showed some fluctuations after epoch 9. This suggests mild overfitting, where the model begins to adapt too closely to the training data, temporarily reducing its ability to generalize to unseen data. Despite these small instabilities, the model recovered and still achieved strong final performance, indicating good overall generalization. Since the overfitting was mild and limited to just some epochs and considering the fact that the final accuracy value was an excellent result, SecondCNN has been considered the best model and the hyperparameter tuning phase has been performed on this model.

6 Hyperparameter tuning

For the hyperparameter tuning part, the Bayesian Optimization was implemented using the Optuna library. This method has been considered advantageous because it is able to automatically find the best training settings for the neural networks with less interactions than other simpler but valid methods, like grid search or random search. In addition, still using Optuna, the early stopping and pruning were implemented to interrupt in advance the trial that were not performing good, saving computational time and also preventing overfitting by halting the training process before the model begins to memorize the training data.

The implementation was made with the `objective` function which defined the training process for one trial where Optuna suggested values for the following hyperparameters: learning rate, batch size, dropout and optimizer type. In the subsequent trials, for each of them the model was trained on the training set and evaluated on the validation set for a pre-chosen (maximum) number of epochs equal to 20. This value was chosen to be sufficient to allow the model to learn without risking to stop to early (underfitting) but also to avoid the model to memorize the data loosing the ability to generalize on new data (overfitting). Early stopping could eventually intervene before the maximum limit of 20 epochs and stop (prune) unpromising trials if the validation accuracy didn't improve for 4 consecutive epoch with the aim of saving time and therefore improve the efficiency of model performance automatically. Training and validation accuracy were, in fact, calculated at each epoch and the best validation accuracy was returned as the trial's score. **Choice of the number of trials:** Both 20 and 30 were tried as values for the trails in study.optimize(objective_func, n_trials=20, timeout=3600) and it has been noticed that choosing 20 didn't lead to a loss of accuracy or other metrics; therefore, also to respect the aim of preferring achieving a reasonable training time over maximizing accuracy, this amount of trials was selected.

6.1 Choice of the hyperparameters:

1. **Learning rate** (lr = trial.suggest_loguniform('learning_rate', 1e-5, 1e-2): controls the speed at which the model updated its weights during the training. If it is too high, the model might overshoot the best solution and fail to converge, but if it is too low, training becomes very slow and risks to get stuck. Tuning the learning rate is crucial to have a good performance of the model because the effect on model's accuracy is direct. In particular, the assigned minimum and maximum values are 0.00001 and 0.01 and the model is asked to choose a random value between them using a logarithmic scale (loguniform), so that all the values have the same probability to be chosen and none of the possible best values is excluded a priori. 0.00001 is a really small value, used when a lot of precision is required to have a slow, and therefore more time-consuming, but complete training, while 0.01 guarantees a faster but less precise training. Still related to the learning rate, the **learning rate scheduler** was applied, is a technique that automatically adjusted the learning rate during the training process. The learning rate was therefore gradually reduced to fine-tune the weights and allow the model to converge more smoothly, reaching better performance in fewer epochs and allowing to avoid wasting time during the training part using a too high or too low learning rate, according with the idea of achieving a reasonable training time. Three learning rate scheduler were tried:

    (a) **StepLR**: gave the worst results, with a test accuracy equal to 31% and a precision of 10%. The evaluation metrics show that the model failed to learn.
    Why was this Scheduler tried? Commontly used because it is simple and effective for many problems.
    textbfResults:

    (b) **CosineAnnealingLR**: determined a significant improvement from StepLR, the model trained using this learning rate scheduler was good and stable, but not excellent.

Why was this Scheduler tried? It smoothly reduces the learning rate, often leading to faster convergence and it is also adapt and often used for image classification.
textbfResults:

(c) **ReduceLROnPlateau**: gave the best overall performance achieving the highest test accuracy also on the custom images (80%)
Why was this Scheduler tried? Automatically reduces the LR when validation loss stops improving preventing wasting time with a too-high learning rate when the model is stuck. It can speed up convergence.
textbfResults:

Table: Evaluation metrics of the learning rate schedulers

Values	StepLR	CosineAnnealingLR	ReduceLROnPlateau
Accuracy	0.3191	0.9301	0.9787
Precision	0.1064	0.9343	0.9780
Recall	0.3333	0.9380	0.9808
F1	0.1613	0.9314	0.9791

2. **Batch size** (batch_size = trial.suggest_categorical('batch_size', [16, 24, 32, 48])): number of sample processed together before the model updates its weights. The four arbitrary values (16, 24, 32, 48) are chosen considering the fact that small batch sizes (16) indicate more frequent updates, a slower process but can lead to a better generalization, and larger batch sizes (48) speed up training but require more memory and may cause the model to generalize worse. It is an important parameter to tune because it affects the speed of training, stability of updates, and quality of the learned model.

3. **Dropout** (dropout = trial.suggest_uniform('dropout', 0.2, 0.5)): regularization technique that randomly "turns off" a fraction of neurons during training. The chosen dropout rates are 20% and 50%, suggesting the model to choose a dropout rate anywhere between 0.2 and 0.5, not just exactly 0.2 or 0.5. With 0.2 (dropping 20% of neurons randomly), the model memorize a lot of the training set (overfitting), and with 0.5 the model learns in a more robust way but can risk to not acquire information well. It is an important parameter to avoid the model to learn too well only on training data, it, in fact, prevents overfitting.

4. **Optimizer**: algorithm that updates the model's weights based on the loss to reduce the error. It is necessary to try different optimizers (Adam, SGD, RMSprop) because they have different behaviors and work better in different situations. Choosing the best optimizer for the specific case, can improve how fast and how well the model learns.

Other hyperparameters could have been tuned to maximize accuracy, like convolutional filters, number of fully connected units or kernel size, however the training time would have increased, in contrast with the aim of this project.

The `plot_best_trials` functions was implemented immediately after the objective function to visualize accuracy and loss for training and validation in top-performing trials.

6.2 Results:

The hyperparameters were optimized only on **SecondCNN**, in agreement with the plan to minimize training time instead of maximizing accuracy. SecondCNN, in fact, showed the best accuracy in less time (as explained in the Training process section). The learning rate scheduler used was ReduceLROnPlateau. The hyperparameter tuning further refined SecondCNN performance reducing the overfitting present in the training phase. This model converged quickly within 10 epochs. The validation accuracy, around 98%, was smooth and stable, with a validation loss that decreased rapidly.

The best hyperparameters (parameters of the best trial) identified were:

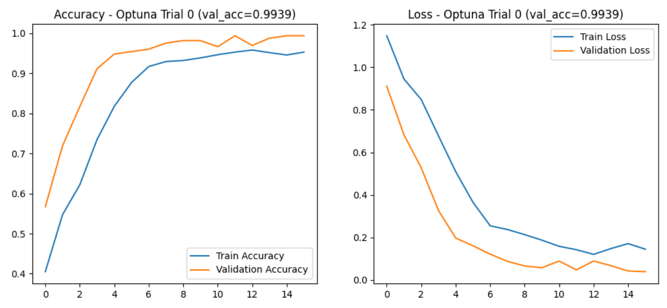
- Learning rate: 0.00030004859971087056 (0.0003)
  The model updated its weights slowly and gradually. Since this was the best trial, gradual updates were more effective than aggressive changes. This contributed to stable convergence and reduced oscillation in validation loss.

- Batch size: 32
  The model learnt effectively from moderate-sized batches. Considering that the suggested values are 18, 24, 32 and 48, the model balanced learning stability and generalization without sacrificing convergence speed.

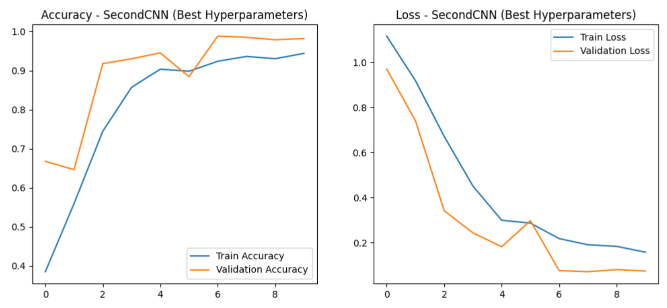
- Dropout: 0.22718746415732172 (0.22)
  Moderate value meaning that the model needed some regularization to prevent overfitting but not an extreme dropout.

- Optimizer: Adam (adaptive optimizer that adjusts learning rates individually for each parameter)
  This optimizer converged faster and more effectively than traditional optimizers.

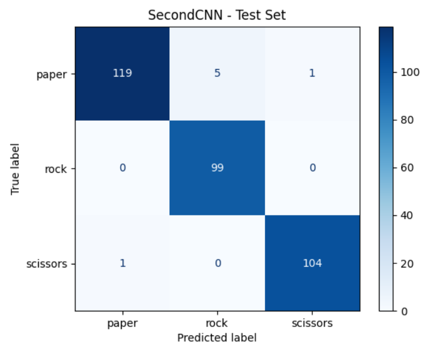
Trial with the best hyperparameters:



SecondCNN trained with the best hyperparameters:



Confusion matrix



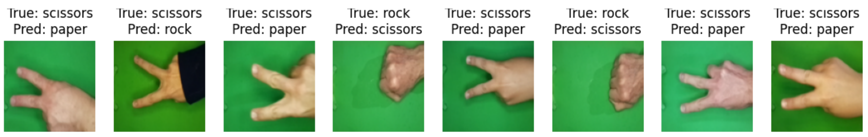
True label \ Predicted label	paper	rock	scissors
paper	119	5	1
rock	0	99	0
scissors	1	0	104

SecondCNN on the test set performed well, making only 7 mistakes on 328 images, confirming that the choice of the model and hyperparameters was correct for this image classification task. Therefore it learnt effectively, also gaining the ability to generalize on unseen data.

7 Misclassified images

The function show_misclassified was implemented to observe and understand the patterns in the wrong classification of the images. The main misclassified classes were Rock and Scissors, Usually, this could have been due to the imbalance between the classes but, not only in this case the three classes were balanced, Scissors and Rock were the most represented classes between the three. Therefore, the assumption on the reason of the misclassification can be found in:

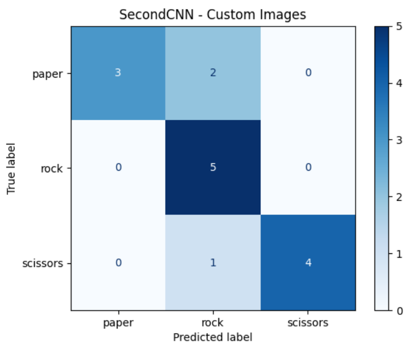
- Gestures ambiguity, also due to the position of the hand after the transformations.
- Lighting condition: as it is possible to observe in the fourth and sixth image, both predicted as Scissors when they were Rock, the shadow may have lead to the misclassification.



8 Generalization test: custom images

A small dataset of custom images was created, including 15 images balanced between the classes (5 for Rock, 5 for Paper and 5 for Scissors), all of them with a green background but with slightly different positions of the hands and different lighting conditions to better test the model. The dataset was able to correctly interpret and classify most of the pictures, as it is possible to observe from the confusion matrix. The misclassified ones presented a deceptive position of the hand (one of them) which, in fact, had the fingers open, or difficult lighting conditions (two of them), for which either the reflection of the light or the shadow of the hand was clearly visible.

Confusion matrix



True \ Predicted	paper	rock	scissors
paper	3	2	0
rock	0	5	0
scissors	0	1	4

9 Conclusions:

In this project, three CNN architectures with increasing complexity were implemented and evaluated on the Rock-Paper-Scissors dataset resized to 100x100 pixels. The simplest model, FirstCNN, had its simplicity reflected in the accuracy and the other evaluation metrics. The more complex SecondCNN significantly improved performance, achieving almost 99% accuracy and demonstrating stable and efficient training. The most complex model, ThirdCNN, was deeper and more parameter-rich, had excellent results too but was too complex for this dataset. Hyperparameter tuning using Optuna, applied only on SecondCNN to respect the computational time constraint, further refined its performance, confirming it as the best model choice balancing accuracy and computational cost. It generalized well to custom hand gestures, proving its robustness. Common errors were associated with class similarity and lighting conditions. Confusion matrices confirmed balanced performance across classes. These results highlight the importance of matching model complexity to dataset size and task requirements, and applying hyperparameter optimization for optimal performance.