Rock, Paper and Scissors Image Classification

Alessandro Bottoni

February 11, 2026

1 Introduction

The following project aims at developing a Convolutional Neural Network capable of recognizing rock, paper and scissors hand gestures.

2 Dataset

The dataset is composed of 2189 pictures in a .png format and divided in three classes: 1. Rock, with 726 pictures. 2. Paper, with 712 pictures. 3. Scissors, with 750 pictures.

3 Preprocessing and Data Augmentation

To prepare the dataset for training, a series of preprocessing steps and data augmentation techniques were applied. First, all images were resized to a uniform size of 256×256 pixels to ensure consistent input dimensions for the CNN. Next, pixel values were normalized to the range by dividing by 255, which supports faster convergence during training. To increase training-data diversity and improve model generalization, several augmentation techniques were used. These included random horizontal flips to reflect natural variability in hand gestures and random rotations within $\pm 15°$ to account for different hand orientations. In addition, random color jittering was applied to vary brightness, contrast, and saturation, improving robustness to changing lighting conditions.

4 Train/Validation/Test Split

Altough it is a best practice to use cross validation to validate models, in order to avoid overfitting on the validation set, for the first model a simple train/validation/test split was used. This choice was made to speed up the training process, which is already quite long with the current hardware. The dataset was split into 70The split was performed separately for each class to ensure that the distribution of classes is consistent across all sets. As for the other two models a cross validation approach has been used, to better evaluate the performance of the models and avoid overfitting on the validation set.

5 Simple CNN Model and Training

Model Architecture: Simple CNN: The first model designed is a very simple Convolutional Neural Network (CNN) architecture. It consists of a single convolutional layer followed by a ReLU activation function, a pooling layer, a flatten layer and a fully connected one. The convolutional layer takes as input 3 channels (RGB color images), the only fixed parameter, while the number of output channels and the kernel size are hyperparameters that can be tuned in the following way: - Number of output channels: 16, 32, 64 - Kernel size: 3, 4, 7 The padding is automatically calculated as the kernel size divided by 2, to preserve the spatial dimensions of the input. The purpose of this layer is to extract spatial features from the input images, such as edges and textures. The ReLU activation function introduces non-linearity to the model, allowing it to learn more complex patterns, while the pooling layer reduces the spatial dimensions of the feature maps, which helps to reduce the number of parameters and computational cost. The kernel size of the pooling layer is tunable as well as we will see further ahead, and can be set to either 2 or 4, where 2 preserves more details, and 4 is faster but loses information Finally, the flatten layer converts the 2D feature maps into a 1D vector, which is then passed to the fully connected layer for classification.

Hyperparameter Tuning: The project uses Ray Tune for hyperparameter tuning, which is a powerful library for distributed hyperparameter optimization; this allows us to efficiently search through the hyperparameter space and find the best combination of parameters for our model, splitting the workload across multiple compute resources if available. The hyperparameters that are being tuned include the number of output channels in the convolutional layer, the kernel size of the convolutional layer and the kernel size of the pooling layer for the architectur of the model, as well as the learning rate and the batch size for the training process. The parameters are sampled in the following way: - Number of output channels and kernel size of the convolutional layer are being sample through a grid search, respectively from the sets 16, 32, 64 and 3, 4, 7. This means that all possible combinations of these parameters will be evaluated, hence a total of 9 combinations (computationally expensive but exhaustive). - Kernal size of the pooling layer and batch size are being sampled through a random search, respectively from the sets 2, 4 and 16, 32, 64. This means that a random combination of these parameters will be evaluated at each trial (less computationally expensive altough not exhaustive, to save computing power and time). - Learning rate is being sampled through a log-uniform distribution between 1e-4 and 1e-2, which allows us to explore a wide range of learning rates on a logarithmic scale, which is often more effective for finding good learning rates in deep learning models.

5.1 Model Training Methodology

5.1.1 Hyperparameter Search Space and Strategy

The project employs Ray Tune, a distributed hyperparameter optimization framework, to systematically explore the hyperparameter space. Ray Tune provides efficient resource management, parallel trial execution, and sophisticated scheduling algorithms for early stopping.

5.1.2 Architecture Hyperparameters (Grid Search)

Architecture-related hyperparameters are explored using exhaustive grid search to ensure comprehensive evaluation of model capacity:

- **Output Channels** $\in \{8, 16, 32\}$: Controls the number of filters in the convolutional layer, directly affecting model capacity. Lower values reduce parameters and computational cost, while higher values increase representational power.
- **Convolutional Kernel Size** $\in \{3, 4, 7\}$: Determines the receptive field of each filter. Smaller kernels capture fine-grained local features, while larger kernels capture broader spatial context. The sizes were chosen to represent small (3×3), medium (4×4), and large (7×7) receptive fields.

Grid search generates $3 \times 3 = 9$ unique architecture combinations.

5.1.3 Training Hyperparameters (Random Search)

Training-related hyperparameters are sampled randomly to balance exploration efficiency with computational cost:

- **Pooling Kernel Size** $\in \{2, 4\}$: Sampled using `tune.choice`. Controls the downsampling factor. A kernel size of 2 is standard practice, preserving more spatial detail, while a kernel size of 4 performs aggressive downsampling, trading spatial resolution for faster computation.
- **Batch Size** $\in \{16, 32, 64\}$: Sampled uniformly with equal probability. Smaller batch sizes provide noisier gradient estimates but often generalize better, while larger batch sizes offer more stable training and faster computation per epoch.

5.1.4 Training Hyperparameters (Log-Uniform)

- **Learning Rate**: Sampled from a log-uniform distribution. Log-uniform sampling ensures equal representation across orders of magnitude, as learning rates often require logarithmic exploration. Mathematically:

With `num_samples=10`, each of the 9 architecture combinations is evaluated with 10 different random samples of training hyperparameters, yielding a maximum of $9 \times 10 = 90$ potential trials.

5.1.5 Ray Tune Configuration and Training Loop

Before describing the tuning workflow, it is useful to summarize the training setup at a high level. The model is trained using stochastic gradient descent with momentum, optimized against a cross-entropy loss for a three-class classification task. Each trial runs for a limited number of epochs (up to 10), and the batch size is treated as a tunable parameter, so the amount of data processed per update can change across trials. Data are fed through separate training and validation loaders: the training loader shuffles samples to improve generalization, while the validation loader keeps a fixed order to make evaluation stable; a small number of worker processes is used to overlap data loading with computation.

In this project, hyperparameter tuning is organized as a collection of independent *trials*. Each trial corresponds to one concrete set of hyperparameters and runs a short training job, producing comparable validation metrics that Ray Tune can track across experiments. The overall goal is to explore many configurations efficiently while keeping the execution stable and reproducible.

To keep the machine responsive, the tuning process is run under a controlled resource budget. Instead of letting every trial consume as many resources as it wants, each trial is assigned a fixed amount of compute (a certain number of CPU cores). This explicit resource assignment determines how many trials can run at the same time: if each trial needs more CPU, fewer trials can run in parallel; if each trial needs less, more trials can run concurrently. Ray Tune supports this kind of per-trial resource allocation so that parallelism is predictable and does not overwhelm the system. [6]

To avoid wasting time on bad configurations, the search uses the ASHA scheduler (Asynchronous Successive Halving). ASHA implements an early-stopping policy that continuously compares trials as they progress: at predefined milestones in training, a trial's performance is evaluated against other trials that have reached the same point. Trials that are clearly underperforming are stopped early, and the freed resources are immediately reused to start new trials or continue more promising ones. Because decisions are made asynchronously, trials do not need to wait for each other, which improves resource utilization and speeds up the overall search. [6]

Checkpointing is used to preserve useful model states during tuning. Rather than saving many snapshots, the configuration can be set up to retain only the most relevant checkpoint per trial (typically the best one according to the chosen validation metric). This makes it easier to recover the best-performing model after tuning and reduces disk usage, which is important when running many trials. [6]

Inside each trial, the training loop follows a standard pattern. Each epoch is split into a training phase (where model parameters are updated) and a validation phase (where performance is measured without updating the model). After each epoch, the trial reports key metrics (such as validation loss, validation accuracy, and training loss) back to Ray Tune so that the scheduler can make early-stopping decisions and so that results are logged in a consistent format. [6]

Finally, the entire experiment is orchestrated by the Tune *Tuner*. The Tuner combines the search space definition with the optimization objective (which metric to minimize or maximize) and the scheduler policy. It generates trial configurations, queues them, runs them when resources are available, and collects the results into a single structured output that you can inspect to select the best configuration and its corresponding checkpoint. [6]

5.2 Results

6 Medium CNN Model and Training

6.1 Medium CNN (Model 2) and training procedure

In the second experiment we moved from the baseline (Model 1) to a moderately deeper CNN architecture, while keeping the same overall input pipeline and classification goal (3 classes: rock, paper, scissors).Differently from Model 1, here the model assessment and model selection were carried out using **K-fold cross-validation**, in order to obtain a more reliable estimate of the generalization performance and to reduce dependence on a single train/-validation split.The preprocessing and data augmentation pipeline is the same as in the model 1, with the exception of the train/validation split, which is now performed within each fold of the cross-validation procedure.

6.1.1 Architecture (MediumCNN)

Model 2 increases capacity by stacking two convolutional blocks and introducing explicit regularization (Batch Normalization and Dropout).The network is composed of a `features` module (convolutions, normalization, ReLU and pooling) and a `classifier` module (fully-connected layers).

Feature extractor.

- **Block 1:** `Conv2d(3` $\to$ `16, kernel=7, stride=1, padding=3)` $\to$ `BatchNorm2d(16)` $\to$ `ReLU` $\to$ `MaxPool2d(2)`.
- **Block 2:** `Conv2d(16` $\to$ `32, kernel=3, padding=1)` $\to$ `BatchNorm2d(32)` $\to$ `ReLU` $\to$ `MaxPool2d(2)`.

Classifier. After the convolutional blocks, the feature maps are flattened and passed to:

- `LazyLinear(128)` $\to$ `ReLU` $\to$ `Dropout(0.5)` $\to$ `Linear(128, 3)`.

The final layer outputs logits for the three-class classification task.

6.1.2 Training setup

The model is trained with cross-entropy loss (`CrossEntropyLoss`) and optimized using stochastic gradient descent with momentum (`SGD`, momentum = 0.9).We fix the main hyperparameters to the best configuration previously found while experimenting with Model 1 (learning rate $\approx 5.12 \times 10^{-4}$, batch size = 16), and we do not run a new tuning procedure for Model 2.Each training run lasts 20 epochs.

To improve reproducibility, we set a global seed (`SEED`=42) for Python, NumPy, and PyTorch, and we enable deterministic behavior where applicable.Training is executed on Apple Silicon `MPS` when available, otherwise on CPU.

6.1.3 K-fold cross-validation protocol

To perform cross-validation, we combine the available training and validation folders into a single dataset and then split it using `KFold` with $K = 10$, shuffling enabled and `random_state`=42.For each fold, the model is trained *from scratch* on $K - 1$ folds and validated on the remaining fold, repeating the process until each fold has served as validation exactly once.Within each fold, we build a shuffled training dataloader and a non-shuffled validation dataloader (batch size = 16, `num_workers`=2).

During training we log training and validation loss/accuracy at each epoch, and we track the best validation loss and best validation accuracy observed within each fold.After completing the 10 folds, we retrain a final instance of the same architecture on the full (train+validation) dataset using the selected training configuration, and we save the resulting weights for the final test evaluation.

6.2 Results

7 Residual CNN Model and Training

7.1 Residual CNN (Model 3) and training procedure

In the third experiment we further increased model capacity by adopting a residual CNN architecture, i.e., a network built from residual blocks with skip connections, while keeping the same three-class classification objective (rock, paper, scissors). As for Model 2, model assessment is performed using **K-fold cross-validation**, which reduces dependence on a single train/validation split and provides a more stable estimate of generalization performance. The preprocessing and data augmentation pipeline is the same as in the model 1, with the exception of the train/validation split, which is now performed within each fold of the cross-validation procedure.

7.1.1 Residual blocks (BasicBlock)

Model 3 is based on residual learning, where the output of a transformation $F(\cdot)$ is added to the original input through a shortcut connection. In its simplest form, a residual block can be expressed as

$$y = F(x) + x,$$

which helps the optimization of deeper networks by allowing gradients to flow through identity paths.

In our implementation, each residual block (`BasicBlock`) contains two 3×3 convolutional layers, each followed by batch normalization, with a ReLU non-linearity after the first convolution and again after the residual addition. To introduce regularization, the block can optionally apply a spatial dropout (`Dropout2d`) after the first activation (with a drop probability set per stage). When the number of channels changes or spatial downsampling is required (i.e., `stride` $\neq 1$), the shortcut branch uses a projection consisting of a 1×1 convolution with matching stride followed by batch normalization, so that tensor dimensions align before the summation.

7.1.2 Architecture (ResCNN)

The network is organized into: (i) an initial convolutional stem, (ii) three residual stages, and (iii) a classification head.

Stem. The stem is composed of `Conv2d(3` $\to$ `16, kernel=7, stride=1, padding=3)` $\to$ `BatchNorm2d(16)` $\to$ `ReLU` $\to$ `MaxPool2d(2)`.

Residual stages.

- **Stage 1 (16 channels, no downsampling):** `BasicBlock(16` $\rightarrow$ `16, stride=1, drop=0.0)` $\times 2$.
- **Stage 2 (32 channels, downsampling):** `BasicBlock(16` $\rightarrow$ `32, stride=2, drop=0.1)` followed by `BasicBlock(32` $\rightarrow$ `32, stride=1, drop=0.1)`.
- **Stage 3 (64 channels, downsampling):** `BasicBlock(32` $\rightarrow$ `64, stride=2, drop=0.2)` followed by `BasicBlock(64` $\rightarrow$ `64, stride=1, drop=0.2)`.

Classifier. After the residual stages, features are flattened and passed to `LazyLinear(128)` $\rightarrow$ `ReLU` $\rightarrow$ `Dropout(0.5)` $\rightarrow$ `Linear(128, 3)`. The output layer produces logits for the three target classes.

7.1.3 Training setup

Training uses a multi-class cross-entropy loss (`CrossEntropyLoss`) and stochastic gradient descent with momentum (`SGD`, momentum = 0.9). We keep the learning rate fixed to 5.115859×10^{-4} and use a batch size of 16, training for 20 epochs per fold. To improve reproducibility, we set a global seed (`SEED`=42) for Python, NumPy, and PyTorch, and enable deterministic behavior where applicable. Training runs on Apple Silicon `MPS` when available; otherwise it falls back to CPU.

7.1.4 K-fold cross-validation protocol

Cross-validation is implemented by merging the existing training and validation folders into a single dataset (`ConcatDataset`) and then applying `KFold` with $K = 5$, shuffling enabled, and `random_state`=42. For each fold, the model is trained *from scratch* on $K - 1$ folds and validated on the remaining fold, so that each sample serves as validation exactly once. Within each fold, we build a shuffled training dataloader and a non-shuffled validation dataloader (batch size = 16, `num_workers`=0). During training we log train/validation loss and accuracy at each epoch and track the best validation metrics within each fold for later aggregation (reported in the Results section).

After completing cross-validation, we train a final instance of the same residual architecture on the full (train+validation) dataset using the selected configuration and save the weights to disk for final test evaluation (reported separately).

7.2 Results

8 Discussion and Conclusion

9 Bibliography

References

[1] Understand "stride". https://medium.com/@bragadeeshs/stride-in-cnns-stepping-towards-efficient-image-processing-e58a34b02ff0

[2] Basic tutorials for inspiration: what is torch.nn?. https://docs.pytorch.org/tutorials/

[3] Basic tutorials for inspiration: training a Neural Network. https://docs.pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

[4] Basic tutorials for inspiration: training an image classifier. https://docs.pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[5] Understand conv2d parameters. https://www.codegenes.net/blog/conv2d-parameter-object-input-pytorch/

[6] Hyperparameter fine-tuning with Ray. https://docs.pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html

[7] Stochastic Gradient Descent and Momentum. https://www.lunartech.ai/blog/mastering-stochastic-gradient-descent-the-backbone-of-deep-learning-optimization

[8] SkLearn for Cross Validation. https://discuss.pytorch.org/t/how-can-i-use-sklearn-kfold-with-imagefolder/36577

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.