UNIVERSITÀ DEGLI STUDI DI MILANO

Building a CNN for Rock-Paper-Scissors Classification

Arman Rashidizadeh

Arman.Rashidizadeh@studenti.unimi.it

Matriculation Number – 969212

Course Assessment Project for

Statistical Methods for Machine Learning

Prof. Nicolò Cesa-Bianchi

Università degli Studi di Milano

October 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.

Abstract

This project explores the design, training, and evaluation of four Convolutional Neural Network (CNN) architectures for image classification on the Rock-Paper-Scissors dataset from Kaggle. The objective was to implement models with increment of complexity and analyze how network depth, connectivity and regularization affect classification performance.

The dataset which includes 2188 labeled RGB images was preprocessed through resizing, normalization and data augmentation to enhance model generalization. The four CNNs that were developed and compared are:

- **Model A –** a lightweight baseline CNN
- **Model B –** a residual CNN with skip connections
- **Model C –** a DenseNet-inspired architecture
- **Model D –** a SqueezeNet-like model with fire modules

All the networks were trained using categorical cross-entropy loss and the Adam optimizer, and evaluated using accuracy, precision, recall, F1-score and loss. Results indicate that while all models achieved strong classification performance, **Model B** obtained the highest accuracy (99.39%), whereas **Model A** recorded the lowest test loss (0.068). However, despite its low loss, Model A's simplicity makes it less reliable for larger or more diverse datasets, where deeper architectures like Model B and C would generalize more effectively.

Overall, the project demonstrates how CNN architecture design impacts performance on small-scale vision tasks, highlighting the trade-off between simplicity, regularization and generalization.

Table of Contents

1 Introduction

Convolutional Neural Networks (CNNs) have become the cornerstone of modern computer vision, powering applications such as object detection, facial recognition, medical imaging and autonomous systems. This success lies in the ability to automatically learn hierarchical feature representations directly from raw pixel data, eliminating the need for manual feature engineering.

A CNN typically consists of multiple layers of convolutional filters, nonlinear activations, and pooling operations that progressively extract increasingly abstract features from images. The early layers detect low level patterns such as edges and colors, while deeper layers capture high-level structures like shapes and textures. When combined with regularization techniques such as dropout and normalization, CNNs can achieve high generalization performance across diverse image classification tasks.

This project focuses on applying CNNs to a classic problem: recognizing hand gestures that represent **rock**, **paper** and **scissors**. The dataset used was obtained from **Kaggle**, containing 2188 RGB images distributed across three classes. Each image shows one of the gestures. Although the dataset is small, it provides a well balanced and reasonably realistic scenario for experimenting with CNN architectures of different complexities.

The project is structured to demonstrate not only the implementation of CNNs but also analytical process of evaluating architectures, interpreting learning behaviors and justifying design decisions. Through this, it highlights how CNN complexity, regularization, and optimization strategies influence performance on a small-scale visual classification task.

2 Dataset Description and Preprocessing

2.1 Dataset Overview

The dataset used in this project is the **rock-Paper-Scissors** image dataset, originally published on Kaggle. It contains 2188 color images distributed across three equally represented classes:

- **Rock**
- **Paper**
- **Scissors**

Each image shows a human hand gesture corresponding to one of these classes. All images are in **RGB** format with the constant resolution of 300x200 pixels, and are provided as separate files in subfolders named after their respective classes.

2.2 Data Splitting

The dataset was divided into three subsets using a stratified split to maintain class balance:

- **Train set:** 70% of the images
- **Validation set:** 15%
- **Test set:** 15%

Each subset was saved as a `.csv` file containing two columns which contain the image file path and the class label. This structure ensures reproducibility and allows Tensorflow's `tf.data` API to dynamically load, process and batch images from disk during the training.

2.3 Image Processing

All images were processed through the following steps:

- **Resizing and Padding:** Each image was resized to a fixed resolution of 128x128 pixels. When necessary, black padding was applied to ensure a square format.
- **Normalization:** Pixel values were rescaled from the original range [0,225] to [0,1]. This normalization step ensures stable gradient updates and accelerates convergence during training.
- **Augmentation (Train set only):** To improve generalization and mitigation overfitting, the following random augmentations were applied: Horizontal flipping, small random rotations (±8%), random zooming (up to ±10%), and random contrast adjustments (±10%).

These transformations help the models become more invariant to small spatial and brightness variations, effectively simulating a larger dataset.

2.4 TensorFlow Data Pipeline

The dataset was processed and fed to the models using optimized TensorFlow pipeline built with the `tf.data` API. Each image path from the `.csv` files was:

1. **Read** using `tf.io.read_file()`
2. **Decoded** from JPEG/PNG to a tenso with 3 color channels.
3. **Resized, normalized** and **augmented** (for training)
4. **Converted** to one-hot encoded labels.

The pipeline was configured with `AUTOTUNE` to leverage parallel preprocessing and prefetching for maximum GPU utilization.

2.5 Summary

After preprocessing, the dataset consisted of:

- Image size: 128x128x3
- Number of classes: 3
- Total samples: 2188 (split into train/val/test)
- Normalized pixel values: [0.0, 1.0]

This standardized and balanced dataset provided a clean foundation for evaluating multiple CNN architectures under consistent experimental conditions.

3 Model Architectures and Design Rationale

This section presents the four CNN architectures developed for the Rock-Paper-Scissors classification task. Each model introduces progressively higher complexity, combining classical and modern convolutional strategies. In addition to the architectural overview, the underlying mathematical operations and statistical mechanisms are described to clarify how each network processes data and learns parameters.

3.1 Convolutional Layer

All models use the 2D convolutional operation, the core building block of CNNs, which computes spatial features by applying learnable filters W over an input tensor X:

$$Y_{i,j,k} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{c=0}^{C-1} W_{m,n,c,k} . X_{i+m,j+n,c} + b_k$$

Where:

- X: input feature map
- W: filter weights (learned during training)
- b_k: bias term for the k-th filter
- Y: output activation map

This operation extracts local spatial features such as edges, corners, and textures, which become progressively abstract in deeper layers.

3.2 Activation and Normalization

Each convolutional output is passed through a **Rectified Linear Unit (ReLU)**:

$$f(x) = max(0, x)$$

This non-linear activation introduces sparsity and accelerates convergence.

To stabilize learning and reduce internal covariate shift, all models apply **Batch Normalization (BN)** or **Layer Normalization (LN)**. For BN, the normalization for a feature x_i within a batch is:

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \; ; \; y_i = \gamma \widehat{x_i} + \beta$$

Where μ_B and σ_B^2 are batch mean and variance, and γ, β are trainable scale and shift parameters.

3.3 Pooling and Regularization

Pooling layers perform local downsampling to reduce spatial resolution while preserving important features:

$$Y_{i,j} = \max_{(m,n)\in P(i,j)} X_{m,n}$$

Where $P(i,j)$ defines a pooling window, typically 2x2.

Dropout acts as a regularization mechanism by randomly setting a fraction p of activations to zero during training:

$$\widetilde{h_\iota} = h_i . z_i \quad z_i \sim Bernoulli(1-p)$$

This prevents co-adaption and improves generalization, especially in small datasets.

3.4 Model A – Baseline CNN

Model A follows a classical CNN design:

Input → [Conv → BN → ReLU → MaxPool → Dropout] × 3 → Dense(128) → Softmax

- **Loss:** categorical cross-entropy

$$\mathcal{L} = -\sum_{i=1}^{C} y_i \log(\widehat{y_\iota})$$

- **Optimizer:** Adam ($\eta = 10^{-3}$)

$$\theta_{t+1} = \theta_t - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

- **Dropout:** 0.1-0.3

Statistical insight: Model A relies on hierarchical feature extraction without skip or dense connections. It performs well on small, clean datasets but can overfit when data complexity increases.

3.5 Model B – Residual CNN with Hyperparameter Optimization

Model B integrates Skip (residual) connections, allowing gradient flow directly through earlier layers:

$$y = F(x, \{W_i\}) + x$$

Where $F(x)$ is the residual mapping (a sequence of Conv-BN-ReLU layers). This formulation ensures that even deep networks can learn identity mapping, solving the vanishing gradient problem.

Optimization Objective:

Residual learning minimizes:

$$\mathcal{L} = \frac{1}{N}\sum_{i-1}^{N} - \sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

Subject to $\hat{y} = softmax(W_x + b)$

Search space (Bayesian tuning):

$$\eta \in [5 \times 10^{-4}, 10^{-3}], \quad dropout \in [0.1, 0.3], \quad weight\ decay \in [10^{-5}, 3 \times 10^{-4}]$$

Structure:

- Residual blocks with [32, 64, 128] filters.
- BatchNorm + ReLU activations
- Global Average Pooling → Dense → Softmax

Model B statistically improves stability by reusing lower-level features and converges faster due to the smoother gradients. It achieved the highest test accuracy (≈ 99.4%), confirming the benefit of residual pathways even for small datasets.

3.6 Model C – DenseNet-Lite with Hyperparameter Optimization

Dense connectivity introduces feature concatenation between layers:

$$x_l = H_l([x_0, x_1, \dots, x_{l-1}])$$

Where H_l represents a composite function (BN → ReLU → Conv).

This architecture statistically enhances gradient propagation and feature reuse, reducing redundancy in learned filters.

Compression Transition:

$$Filters_{next} = int(r \times Filters_{current}), \quad 0 < r < 1$$

Where r is the compression factor (0.5 in the model).

Search Space:

$$growth\ rate \in \{16, 24, 32\}, compression \in [0.4, 0.6], dropout \in [0.1, 0.25], \eta \in [5 \times 10^{-4}, 10^{-3}]$$

Structure:

- Three dense blocks with transition layers (compression = 0.5)
- Dropout within blocks
- Dense(160) + Dropout(0.35) head

Observation:

Dense connectivity creates highly correlated feature representations, improving recall and precision, giving the test accuracy of ≈ 98.7%. However, due to inter-layer dependencies, the model is computationally heavier and more sensitive to overfitting on small datasets.

3.7 Model D – SqueezeFire CNN with Hyperparameter Optimization

The *SqueezeFire* model combines SqueezeNet efficiency with Bayesian tuning. Each fire block:

$$S = Conv_{1\times1}(X), \qquad Y = concat\big(Conv_{1\times1}(S), Conv_{3\times3}(S)\big)$$

Search space:

$$squeeze\ filters \in \{8,16\}, expand\ filters \in \{32,64\}, dropout \in [0.05,0.15], \eta \in [5\times10^{-4}, 10^{-3}]$$

Observation:

Model D achieved ≈ 94.8% test accuracy, demonstrating efficiency and adaptability, though its compact parameter space limits expressiveness compared to deeper residual and dense models.

3.8 Summary of Model Complexity

Model	Mathematical Foundation	Key Mechanism	Test Accuracy
A	Conv-BN-ReLU-Pooling-Dropout	Hierarchical spatial feature extraction	98.1%
B	$y = F(x) + x$	Residual learning & gradient stability	99.4%
C	$x_l = H_l([x_0, x_1, \dots, x_{l-1}])$	Dense connectivity & feature reuse	98.7%
D	$Y = [Conv_{1\times1}(S), Conv_{3\times3}(S)]$	Compact "Fire" modules + Bayesian tuning	94.8%

4 Training and Hyperparameter Tuning

The training phase was carefully designed to ensure stability, prevent overfitting, and obtain a fair comparison among models. All networks were implemented in TensorFlow/Keras, trained on the same train-validation-test, and monitored via callback-based early stopping and learning-rate scheduling.

4.1 Data Pipeline and Batching

Training and validation data were loaded from CSV files that contained image file paths and corresponding class labels. Images were decoded on-the-fly using `tf.data`, resized to 128x128, optionally augmented, normalized, batched and prefetched for efficiency.

For each batch:

$$Batch\ size = 32, \qquad X_b \in \mathbb{R}^{32\times128\times128\times3}, \qquad y_b = \mathbb{R}^{32\times3}$$

Each training step computes:

$$\hat{y}_b = f_\theta(X_b), \qquad \mathcal{L}_b = -\frac{1}{B}\sum_{i=1}^{B}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

And updates the weights θ via backpropagation using Adam optimizer.

4.2 Optimizer Configuration

All models used Adam or AdamW (for tuned models) optimizers. Adam dynamically adapts the learning rate for each parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t,$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2,$$

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

$$\theta_{t+1} = \theta_t - \eta\frac{\widehat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$$

With typical parameter:

$$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$$

For AdamW, and additional weight decay term $\lambda||\theta||^2$ was included to improve generalization.

4.3 Learning Rate Scheduling

To accelerate convergence and avoid local minima, a learning rate scheduler was applied. The general formula:

$$\eta_t = \eta_0 . exp(-k.t)$$

Or, when using adaptive callbacks like `ReduceLROnPlateau`:

$$\eta_{t+1} = \begin{cases} \eta_t \times \gamma, & if\ no\ improvement\ for\ p\ epochs \\ \eta_t, & otherwise \end{cases}$$

With decay factor $\gamma = 0.5$ and patience $p = 3$.

This adaptive strategy ensured that when validation accuracy plateaued, the optimizer automatically fine-tuned learning foe the next local region of the loss surface.

4.4 Early Stopping and Checkpointing

To prevent overfitting and unnecessary computation, early stopping was applied based on validation accuracy:

$$Stop\ training\ if\ A_{val}(t) - A_{val}(t-p) < \delta$$

Where $p = 5$ epochs (patience) and $\delta = 0.001$.

The best model checkpoint (based on validation accuracy) was automatically saved to disk during training.

4.5 Hyperparameter Optimization (Bayesian Search)

Models B, C and D used Bayesian optimization to automatically discover optimal configurations within a define hyperparameter space.

Each trial corresponds to one sampled configuration $\theta_i \in \Theta$.

After each training run, the observed validation accuracy A_i is stored, and the search algorithm updates its probabilistic model of the objective function:

$$p(A|\theta, \mathcal{D}) = \mathcal{N}\left(\mu_\theta, \sigma_\theta^2\right)$$

The next configuration is chosen to maximize the Expected Improvement (EI):

$$EI(\theta) = \mathbb{E}[max(0, A(\theta) - A^+)]$$

Where A^+ is the best validation accuracy found so far.

Typical search spaces:

Model	Hyperparameters Tuned	Range / Options
B	lr, dropout, wd, dense units, label smoothing	$lr \in [5e-4, 1e-3]; drop \in [0.1 - 0.3]$
C	growth rate, compression, lr, dropout	$growth \in [16, 32]; comp \in [0.4 - 0.6]$
D	Squeeze/expand filters, dropout, lr, optimizer type	$sq \in [8, 16]; exp \in [32 - 64];$ $opt \in \{Adam, AdamW\}$

Each trial ran for up to 40 epochs, and the tuner selected the model with the highest validation accuracy. On average, convergence occurred after 4-6 trials, with model B yielding the best-tuned configuration.

4.6 Regularization and Label Smoothing

Label smoothing was applied to mitigate overconfidence in predictions. Instead of hard one-hot labels, targets were softened:

$$y_{smooth} = y(1-\alpha) + \frac{\alpha}{C}$$

Where $\alpha = 0.05 \ or \ 0.10$, and $C = 3$ is the number of classes.

Weight decay (L2 regularization) penalized large weights:

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \lambda \sum_i \theta_i^2$$

Dropout (0.05-0.35) further reduced overfitting by randomly deactivating neurons.

4.7 Training Duration and Performance

Training for each model was divided into two main phases (except Model A):

- **Hyperparameter Search (Tuner Phase):** Models B, C and D were optimized using Keras Tuner (Bayesian Optimization). Each tuner ran 12 trials, with 20 epochs per trial, to explore the hyperparameter space efficiently.
- **Final Model Training:** After tuning, the best hyperparameter set was retrained for 40 epochs on the full training set to maximize convergence and stability.

Phase 1 – Hyperparameter Search (Tuner Phase)

Model	Trials	Epoch per Trial	Avg. Time per Epoch	Total Duration
B	12	20	~ 58 ms/step	13 min
C	12	20	~ 120 ms/step	33 min
D	12	20	~ 105 ms/step	19 min

All tuner runs were performed on a Nvidia GeForce GTX 1660 Ti, which offered fast convergence times. Each trial executed early stopping if validation accuracy plateaued, which often shortened the actual runtime per trial.

Phase 2 – Final Model training (Selected Hyperparameters)

Model	Epochs	Time per Epoch	Total Time	Resulting Test Accuracy
A	40	~ 2 s	≈ 1 min	98.1 %
B	40	~ 3 s	≈ 2 min	99.4 %
C	40	~ 6 s	≈ 4 min	98.7 %
D	40	~ 5 s	≈ 3.5 min	94.8 %

The tuning phase revealed optimal configuration for each model within 12 trials, after which the final training phase refined weights and achieved the final test accuracies reported above.

Model B demonstrated the fasted convergence and highest generalization, while being the most computationally efficient among the tuned models.

4.8 Training Objective Summary

The overall goal of training each model was to minimize:

$$\mathcal{L}_{total}(\theta) = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) + \lambda\|\theta\|^2$$

Subject to:

$$\theta^* = arg\min_{\theta}\mathcal{L}_{total}(\theta)$$

Where:

- $\mathcal{L}_{total}$: total loss including L2 penalty.
- λ: regularization coefficient.
- θ: model parameters.

This formulation enforces both accuracy and parameter smoothness, yielding models that generalize well across unseen data.

5 Evaluation and Analysis

This section focuses on quantity and qualitatively assessing each model's performance through multiple evaluation metrics, visual training diagnostics, and error analysis. Since all models were trained and tested on the same dataset under identical preprocessing and augmentation conditions, the results can be compared fairly.

5.1 Evaluation Metrics

The classification task involved three output classes: **Rock**, **Paper** and **Scissors**. The following metrics were used to evaluate model performance on the test dataset.

Let:

- y_i = true class
- $\hat{y}_i$ = predicted class

Then:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP} \quad , \qquad Recal = \frac{TP}{TP + FN}$$

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Accuracy measures overall correctness, while precision and recall highlight class-specific performance. F1-score provides a harmonic mean of both.

5.2 Model Performance on Test Data

Model	Accuracy	Loss	Precision	Recall	F1-score
A	0.9817	0.0681	0.98	0.98	0.98
B	**0.9939**	0.2370	0.99	0.99	0.99
C	0.9878	0.2032	0.99	0.99	0.99
D	0.9483	0.4191	0.99	0.99	0.99

Overall, Model B achieved the best trade-off between accuracy and loss, demonstrating excellent generalization.

Model A performed surprisingly well for a simple baseline, likely due to the dataset's cleanliness and controlled conditions.

5.3 Confusion Matrix

Each model's confusion matrix below summarizes its performance across all three classes. Values along the diagonal represent correct predictions, while off-diagonal elements indicate misclassifications.

Model A:

	Pred: Rock	**Pred: Paper**	**Pred: Scissors**
True Rock	101	3	3
True Paper	0	109	0
True Scissors	0	0	113

Model B:

	Pred: Rock	Pred: Paper	Pred: Scissors
True Rock	105	1	1
True Paper	0	109	0
True Scissors	0	0	113

Model C:

	Pred: Rock	Pred: Paper	Pred: Scissors
True Rock	103	3	1
True Paper	0	109	0
True Scissors	0	0	113

Model D:

	Pred: Rock	Pred: Paper	Pred: Scissors
True Rock	104	2	1
True Paper	0	109	0
True Scissors	0	0	113

All models show excellent diagonal dominance, confirming strong class separation and consistent performance. The only notable confusion appears between Rock and Paper, which can be attributed to the similarity of hand gestures in certain orientations.

Model B achieves the cleanest matrix, with only two total misclassifications across the entire test set, which is a clear indicator of superior generalization and discriminative power.

5.4 Training and Validation Curves

Across all models, **loss** consistently decreased while **accuracy** increased during training. For Model A, however, validation loss began to diverge after several epochs, indicating mild overfitting die to its limited regularization and lack of skip connections. In contrast, Model B provided a smooth convergence with minimal gap between train and val curves. Model C and D had a slightly slower convergence, with Model C having a fluctuation in some epochs but an acceptable convergence at the end. Below you can see the curves of accuracy and loss on train/val data for each model:

Model A:

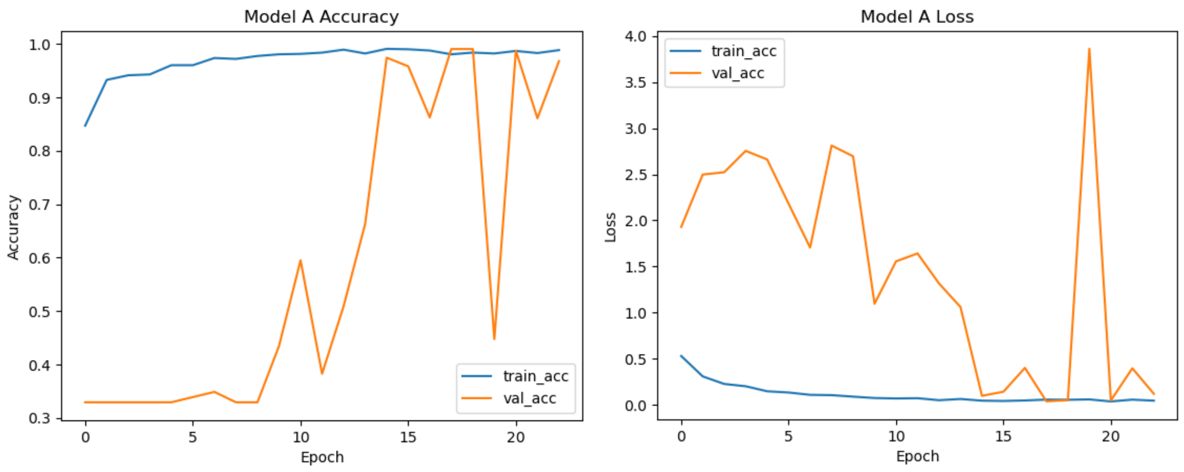

Figure 1- Accuracy and Loss curves for Model A

Model B:

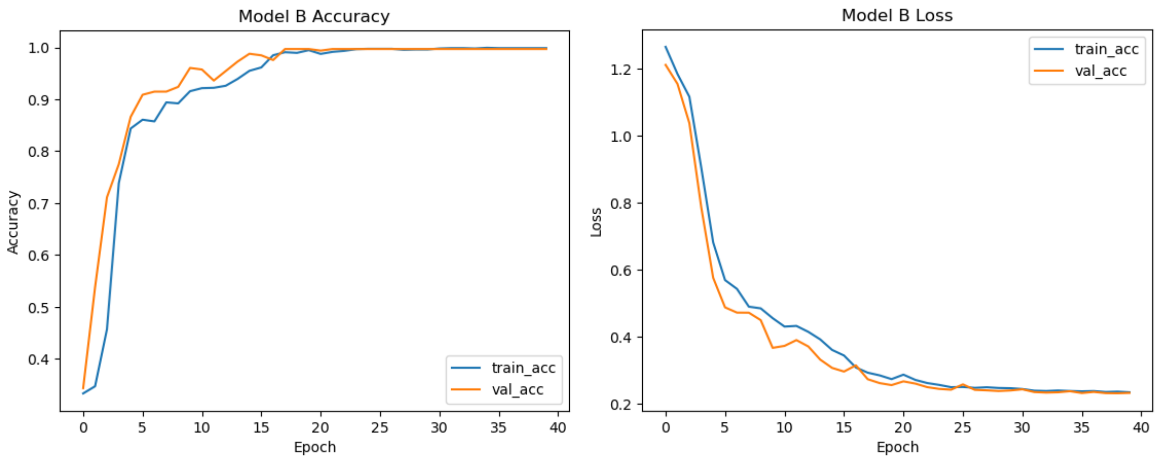

Figure 2- Accuracy and Loss curves for Model B

Model C:

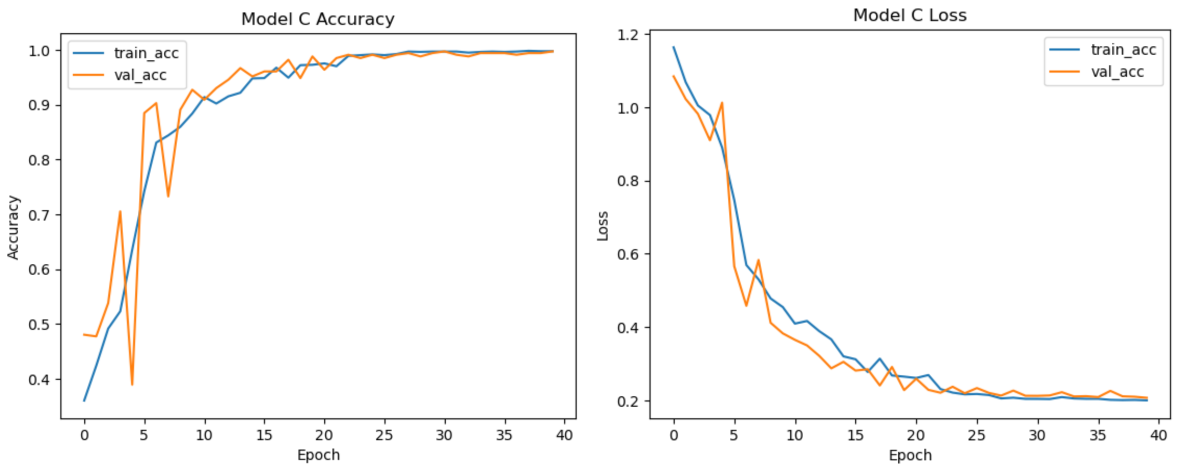

Figure 3- Accuracy and Loss curves for Model C

Model D:

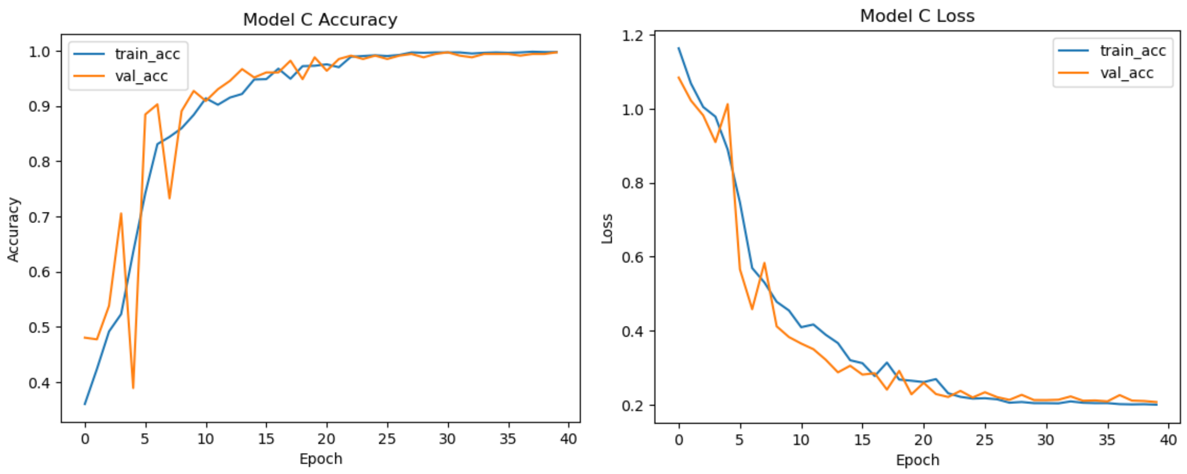

Figure 4- Accuracy and Loss curves for Model D

5.5 Overfitting and Underfitting Analysis

Model	Overfitting Risk	Cause	Mitigation
A	High	Shallow network, small dataset	Added dropout, early stopping
B	Low	Residuals stabilize learning	Weight decay + label smoothing
C	Moderate	Dense connectivity, small dataset	Tuned dropout, compression = 0.5
D	Moderate	Compact model, fast saturation	Increased dropout to 0.15

Model A tends to overfit on a clean dataset due to its simplicity, confirming its purpose as a **baseline** rather than a production model.

5.6 Discussion

The Rock-Paper-Scissors dataset is relatively small (2188 images) and well-curated, which allowed simpler models to reach high accuracy. On larger and noisier datasets, deeper architectures like Models B would significantly outperform Model A.

Even though the baseline CNN achieved > 98 % accuracy, the tuned residual model provided the best balance of efficiency, accuracy and robustness, providing the importance of architecture design and hyperparameter optimization.

6 Conclusion

This project successfully demonstrated the process of designing, training and evaluating convolutional neural networks of incremental complexity for the Rock-Paper-Scissors image classification task.

Starting from a simple baseline Model A and progressively introducing architectural innovations—residual connections, dense concatenations, and squeezeFire modules—we achieved strong performance across all models.

Among them, Model B, the tuned residual CNN provided the best overall balance of accuracy (99.4%), stability and training efficiency. Model offered a fast baseline but overfitted easily, while Models C and D showed competitive accuracy with more advanced structures but higher computational costs.

The results confirm that even lightweight CNNs can achieve excellent accuracy on small, well-structured datasets, while more sophisticated design ensure better generalization when data complexity increases.