Università di Catania

UNIVERSITY OF CATANIA

Department of Mathematics and Computer Science

Machine learning course

Alfio Spoto

RPS Classifier

Final Project Report

Prof. Giovanni Maria Farinella

Academic Year 2023 - 2024

Contents

Chapter 1

Problem

1.1 Introduction to Image Classification Task

In the context of machine learning, image classification represents a fundamental task, the objective of which is to automatically assign an image to a predefined category based on its visual characteristics. The accurate classification of images is of significant importance in numerous application areas, including object detection, surveillance, and many others.

Convolutional neural networks (CNNs) have become the standard technology for tackling image classification tasks due to their ability to capture spatial hierarchies in images through layers of convolution, pooling, and non-linear operations. These networks differ from simple machine learning models mainly due to their architectural complexity and their remarkable ability to automatically learn and extract relevant features from raw pixel data [1].

The specific task of this project is to develop a classifier for the game 'Rock', 'Paper' and 'Scissors' . This involves creating a model capable of distinguishing and classifying images representing the three categories: 'Rock', 'Paper' and 'Scissors' . The use of CNNs for this classification task allows for efficient feature extraction and robust classification performance, leveraging the deep learning capabilities that have proven effective in complex image recognition problems.

1.1.1 Motivation and Application Fields

The 'Rock', 'Paper' and 'Scissors' image classification task was proposed by lecturer *Giovanni Maria Farinella* as a project for the Machine Learning academic course, with the objective of completing the exam. This problem represents an interesting case of the practical application of machine learning techniques.

The potential applications of the classification system span various fields. For instance, a similar system could enhance gesture-based interactive games by swiftly and accurately identifying gestures of 'Rock', 'Paper', or 'Scissors', which is crucial for engaging player interactions. Moreover, in the realm of education, such an application could be utilized to teach children about object recognition and interaction with digital systems in an interactive and enjoyable manner.

1.2 Problem Description

The classification of images representing the game of Rock, Paper, Scissors is a challenging task due to the variability in the way these gestures can be visually represented.

- **Variability within Class:** Within each category (rock, paper, scissors), there may be significant variations in appearance due to differences in hand shapes, hand positions, and camera angles. For instance, the 'scissors' gesture may be mistaken for the 'rock' gesture if filmed from an angle that obscures the fingers or from above. Similarly, it could be confused for the 'paper' gesture if three fingers are extended instead of two.
- **Background and Lighting Conditions:** Images may be captured in different environments with varying backgrounds and lighting conditions, which may affect the performance of the classifier.

In order to guarantee the practicality of the system for real-time applications such as interactive games, it is essential that it is capable of classifying images in a timely manner and that its user interface is intuitive and user-friendly. In order to meet these challenges, it is necessary to develop a robust classification system that can generalise well between different conditions.

Figure 1.1: An example of what a user-friendly gui for the RPS classifier might look like.

1.3 Model Formalization

The classification task in Rock-Paper-Scissors is formalized using a convolutional neural network (CNN) model. The CNN model, denoted as $h_\theta(x)$, predicts a probability distribution over the classes 0, 1, and 2 (representing "Rock," "Paper," and "Scissors" respectively) for a given input pattern x. The model's prediction is defined as follows:

- **Model** :

$$h_\theta(x) = P(y = c \mid x) = \frac{e^{\theta(c)^T \cdot x}}{\sum_{i=0}^{K-1} e^{\theta(i)^T \cdot x}}$$

  where $x \in \mathbb{R}^n$ represents the input pattern, $\theta^{(c)} \in \mathbb{R}^n$ are the parameters for class c, and K is the number of classes.

- **Loss function** :

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=0}^{K-1} \left[\mathbb{I}(y^{(i)} = c) \log(h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \sum_{c=0}^{K-1} \theta_j^{(c)2}$$

  where m is the number of input patterns, $y^{(i)}$ is the true class label for pattern i, $x^{(i)}$ is the feature vector of pattern i, λ is the regularization parameter, and $\theta^{(c)}$ are the model parameters for class c.

- **Derivative of Loss** :

$$\frac{\partial J(\theta)}{\partial \theta_j^{(c)}} = -\frac{1}{m} \sum_{i=1}^{m} \left[\left(\mathbb{I}(y^{(i)} = c) - h_\theta(x^{(i)}) \right) x_j \right] + \frac{\lambda}{m} \theta_j^{(c)} \quad \text{for } j = 0, 1, \ldots, n-1$$

- **Learning procedure**:

$$\theta_j^{(c)} := \theta_j^{(c)} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{(c)}}$$

  where α denotes the learning rate, used to update the parameters $\theta^{(c)}$ based on the gradient of the loss function $J(\theta)$.

The CNN model comprises two main sections: a feature extractor and a classifier.

The feature extractor consists of multiple convolutional layers with batch normalization, ReLU activation, max pooling, and dropout.

The classifier section includes fully connected layers that process the extracted features and output the final class probabilities. Cross-entropy loss is used as the loss function to train the model, which measures the difference between the predicted probabilities and the true labels. The loss function is minimized using gradient descent, where the parameters are updated iteratively based on the derivatives of the loss with respect to the parameters.

Chapter 2

Dataset

2.1 Dataset Creation

For the project, we decided to create the dataset from scratch instead of reusing an existing one, despite the availability of many explanatory datasets for the task at hand. This decision was made for several reasons. The first reason is that the creation of a customised dataset ensures that the data exactly matches the specific requirements and nuances of our project, which may not be adequately covered by existing datasets. Secondly, the process of creating and labelling our data provides a deeper understanding and familiarity with the dataset, which is invaluable during the analysis and training phases of the model. Thirdly, one of the main goals of the project is to encourage hands-on experience in building the dataset and model. This approach forces students to become thoroughly familiar with the entire production process, tackling the problem from scratch and proposing a functional solution for the task at hand.

The dataset consists of 224×224 images representing gestures categorized into three different classes: 'Rock', 'Paper' and 'Scissors' . For dataset construction, videos were captured using an iPhone 14 256GB device [2], which enabled the acquisition of 1080p and 60fps scenes depicting the gesture in motion. The camera was fixed while the gesture moved. Subsequently, a bash script [3] was used to convert the videos from the *MOV* format (the iPhone's acquisition format) to *MP4* for greater standardization and a more intuitive video format for processing. The script invokes `ffmpeg`, a multimedia manipulation utility, to convert the MOV files to MP4 [4] [5].

```
#!/bin/bash

mkdir -p data/train/video_mp4
for file in data/train/video_MOV/*.MOV; do
    filename=$(basename -- "$file")
    ffmpeg -i "$file" -c:v libx264 -preset medium -crf 23 -c:a
    aac -strict experimental "./data/train/video_MP4/${filename%.*}.mp4"
done
```

2.1.1 Data Labeling

To label the data, we adopted an approach based on the folder structure within the dataset. The process involved navigating through the `data/train` and `data/test` folders. Each folder represented a distinct class of gestures, identified by the folder's name itself, with the removal of the '_frames' suffix if present. For instance, all frames associated with the 'rock' gesture were located in the 'rock_frames' folder, and similarly for other gestures.

Within each folder, the corresponding images were extracted and matched with their respective labels. Images from the training folders (`data/train`) were incorporated into a training DataFrame, while those from the test folders (`data/test`) were included in a separate test DataFrame. This method ensured that every image received accurate labeling according to its specific gesture category, effectively preparing the data for both the training and evaluation phases of the model.

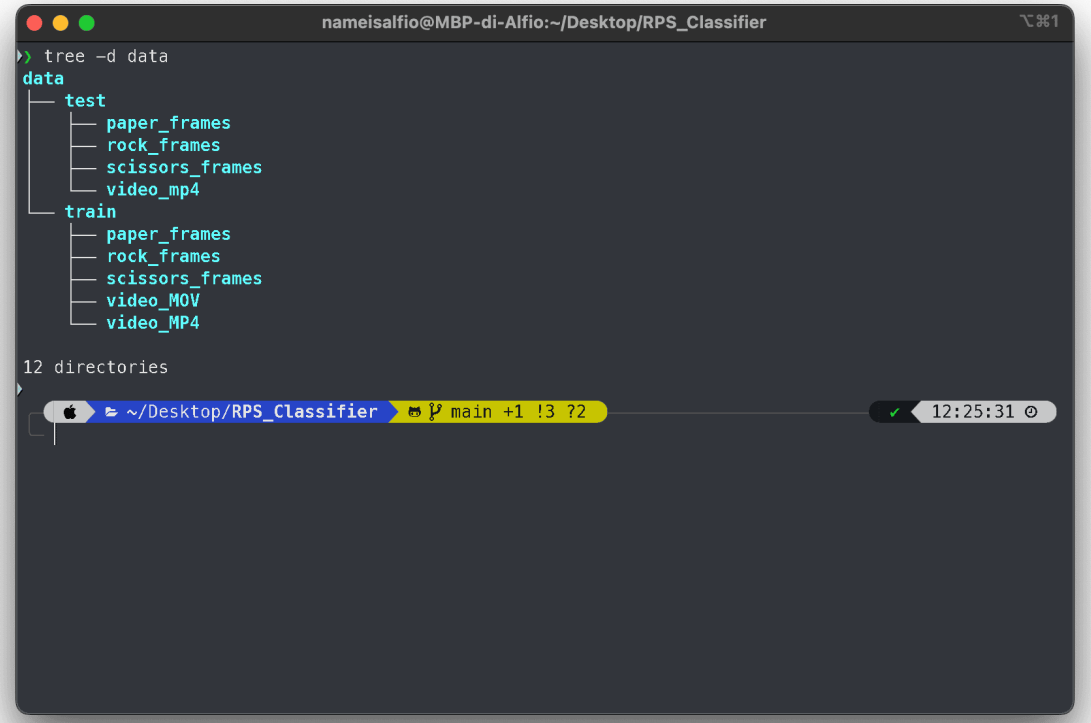


Figure 2.1: Data Organization

Figure 2.1 shows the structure of data folder. Each class contains images corresponding to gestures of that category. This organization allows easy access and management of data during the training and evaluation phases of the model. It was necessary to acquire two different datasets (albeit using the same methodology) because, given the frame acquisition method, the variance between one frame and the next is minimal. In particular, there might even be identical frames. This would represent a clear data leakage problem if the same frame appeared in both the training and test sets. A thorough analysis led to the conclusion that the two datasets should be made independent of each other.

2.1.2 General Information on the Dataset

The entire dataset consists of a total of 16,116 images, evenly distributed among the three gesture classes, and 1,500 testing images. The class distribution is shown in the following images 2.2 for training samples and in 2.3 for testing samples.

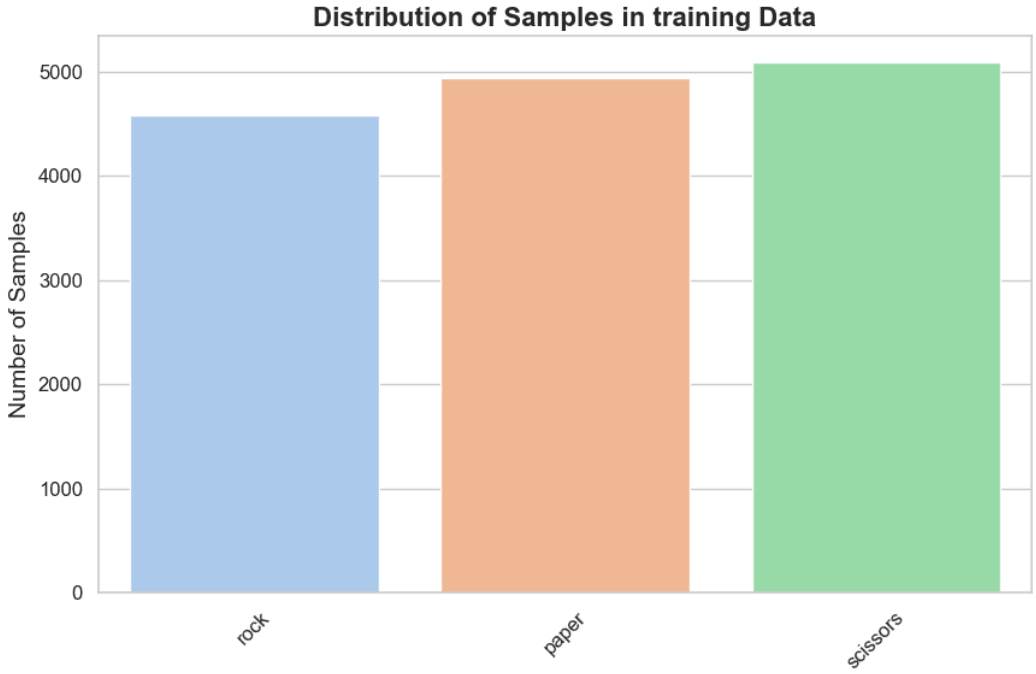


Figure 2.2: Distribution of Training samples

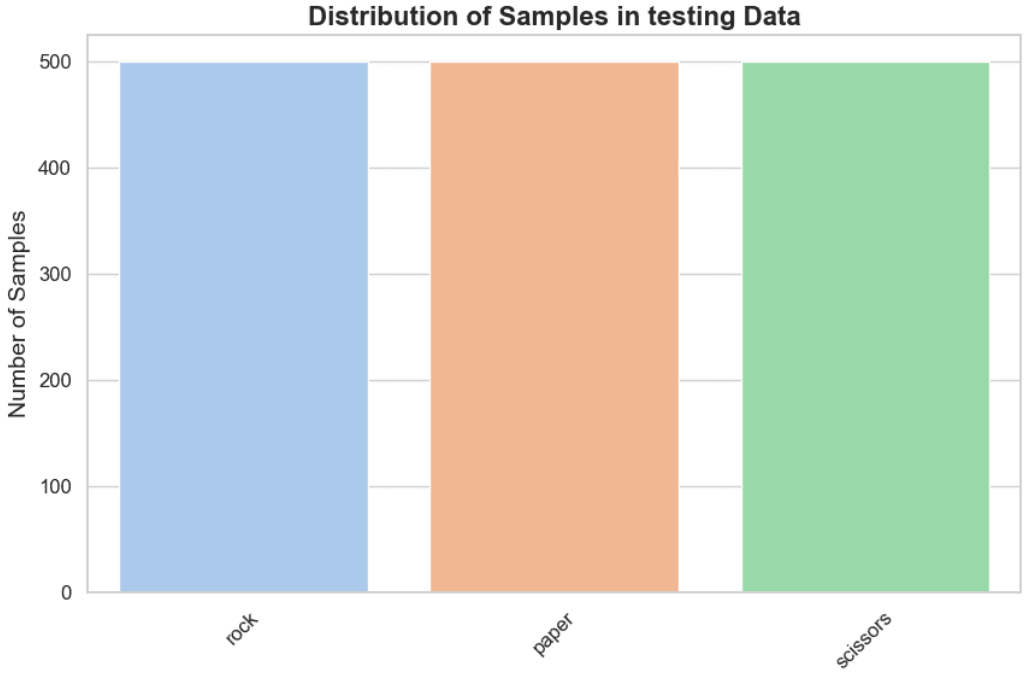


Figure 2.3: Distribution of Testing samples

2.1.3 Visual Examples of the Data

For a better understanding of the dataset, Figure 2.4 shows a visual examples of images belonging to the different classes.

Figure 2.4: Examples of images for each class

These examples highlight the visual diversity of gestures within the dataset, confirming the need for a robust model for their correct classification. A higher variance was intentionally introduced into the training data to enhance the model's robustness and mitigate overfitting. This strategy is designed to enhance performance on the test set by exposing the model to more complex data during training (such as images featuring gloves, watches, and bracelets, which introduce more intricate patterns). Subsequently, during validation and testing phases, the model performs better as it encounters less challenging scenarios.

2.2 Dataset Transformation

To mitigate the risk of overfitting, we applied data augmentation techniques to the input data during model training [6]. The images underwent various transformations to increase data variability:

- Random rotation up to 45 degrees: Images were randomly rotated within a range of up to 45 degrees.
- Random horizontal flip: Images were flipped horizontally with a random probability.
- Random vertical flip: Images were flipped vertically with a random probability.
- Color variation: Brightness, contrast, saturation, and hue variations were applied to the images.
- Random resized crop: Images were randomly cropped and resized to a size of 50x50 pixels.
- Random affine transformation: Random geometric transformations such as rotation, translation, and scaling were applied.
- Final resizing: Images were resized to a fixed size of 50x50 pixels.
- Conversion to tensor: Images were converted into tensors for use as input to the model.
- Normalization: Pixel intensities of the images were normalized using specified mean and standard deviation values.

Additionally, we increased the cardinality of the training set by retaining both the original and augmented images for each example. The validation set was obtained as a partition of the test set and was not obtained from the training data. This decision serves two critical purposes in model development.

Firstly, by extracting the validation set from the testing dataset, we mitigate the risk of data leakage and ensure unbiased model evaluation during training. This approach prevents any potential contamination of the validation set with information from the training data, which could lead to overly optimistic performance estimates.

Secondly, deriving the validation set from the testing dataset enables a more realistic evaluation scenario where the model's performance is assessed on completely unseen data, akin to real-world deployments. This setup provides a reliable measure of the model's generalization capabilities and its ability to handle novel inputs effectively.

The transformations applied to the testing dataset included resizing to 50x50 pixels, conversion to tensors, and normalization using the same mean and standard deviation values used for the training dataset.

- **Training Dataset:** 29,232 samples after applying data augmentation techniques.
- **Validation Dataset:** 300 samples extracted from the testing dataset.
- **Testing Dataset:** 1,200 samples after splitting and excluding the validation set.
- **Total Samples:** 30,732 across training, validation, and testing datasets.

Chapter 3

Methods

The *Rock-Paper-Scissors* (RPS) classifier is designed to recognize and classify images into three categories: rock, paper, and scissors. Figure 3.1 shows a schematic of how the RPS classifier might function.

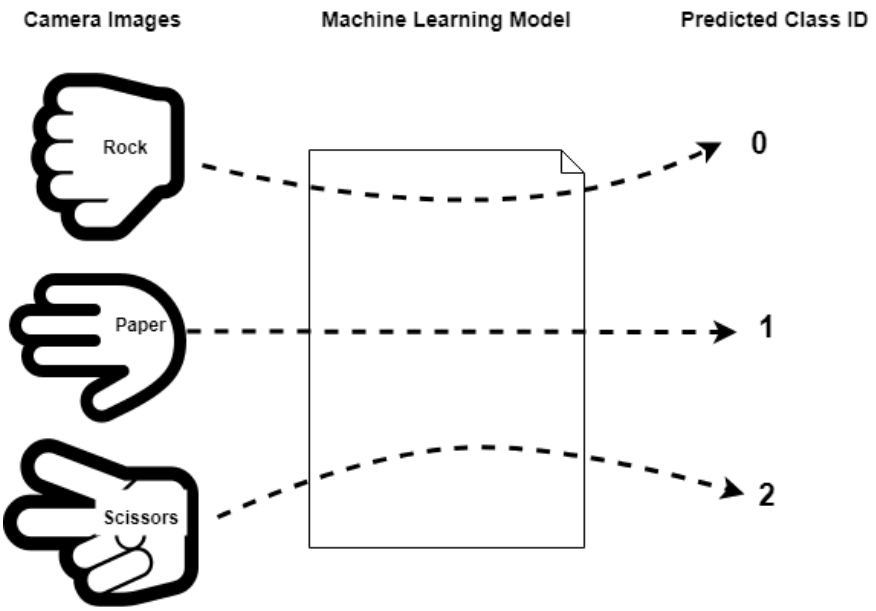


Figure 3.1: Diagram of the RPS classifier functionality.

Once an input pattern is received, the model processes it as follows: the image is passed through a **feature extraction section** comprising multiple convolutional layers, each followed by batch normalization, a ReLU activation function, max pooling, and dropout to prevent overfitting. The extracted features are then flattened and fed into the **classification section**, which consists of fully connected layers. The final output layer produces the predicted probabilities for each class.

The objective of the model is to produce a *one-hot encoding vector* that specifies the class membership for a given input gesture, which identifies the corresponding class [7]. Therefore, the model's predictions can be interpreted as 0, 1, or 2, corresponding to rock, paper, and scissors, respectively.

3.1 Model Architecture

For the design of the RPSClassifier, we conceived a model based on a Convolutional Neural Network (CNN) that utilizes a *softmax* function to finalize the classification [8]. The network architecture consists of four convolutional layers, followed by pooling layers, dropout layers, and fully connected layers. Figure 3.2 shows a three-dimensional diagram of the CNN architecture.

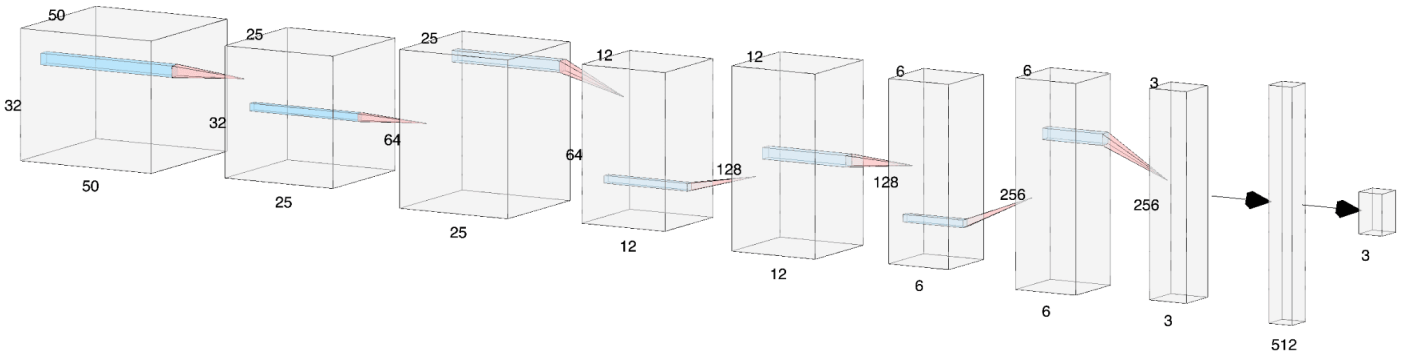


Figure 3.2: 3D diagram of the CNN architecture.

- **Layer 1:** Convolutional layer with 32 filters, 3x3 kernel size, followed by Batch Normalization and ReLU activation. MaxPooling with a 2x2 kernel size and Dropout with a rate of 0.5.
- **Layer 2:** Convolutional layer with 64 filters, 3x3 kernel size, followed by Batch Normalization and ReLU activation. MaxPooling with a 2x2 kernel size and Dropout.
- **Layer 3:** Convolutional layer with 128 filters, 3x3 kernel size, followed by Batch Normalization and ReLU activation. MaxPooling with a 2x2 kernel size and Dropout.
- **Layer 4:** Convolutional layer with 256 filters, 3x3 kernel size, followed by Batch Normalization and ReLU activation. MaxPooling with a 2x2 kernel size and Dropout.
- **Fully Connected Layer:** After flattening, a fully connected layer with 512 neurons, ReLU activation, and Dropout, followed by an output layer with 3 neurons.

3.2 Loss Function

The proposed task is therefore a multiclass classification problem. The loss function used for the task in question is the *Cross Entropy Loss*, which is suitable for multi-class classification problems [9].

More formally, the Cross-Entropy Loss is defined as follows:

- **Cross Entropy Loss (L2-regularization)**:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=0}^{K-1}\left[\mathbb{I}(y^{(i)}=c)\log(h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\sum_{c=0}^{K-1}\theta_j^{(c)2}$$

  where m is the number of input patterns, $y^{(i)}$ is the true class label for pattern i, $x^{(i)}$ is the feature vector of pattern i, λ is the regularization parameter, and $\theta^{(c)}$ are the model parameters for class c.

The regularization term exploits L2 regularization (Ridge Regression), also known as Weight Decay, because it penalizes the model for having large weights. This is achieved by adding the sum of the squared weights to the loss function, scaled by a hyperparameter λ and divided by $2m$ (where m is the number of training samples). This penalty discourages the model from assigning excessive weights to any particular feature, promoting a smoother decision boundary and potentially reducing overfitting [10].

- **Cross Entropy Loss (L1-regularization)**:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum c = 0^{K-1}\left[\mathbb{I}(y^{(i)}=c)\log(h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\sum c = 0^{K-1}|\theta_j^{(c)}|$$

In contrast, L1 regularization (Lasso Regression) introduces a penalty based on the absolute value of the model's parameters [11]. This penalty term, also scaled by a hyperparameter, encourages sparsity in the model by driving some weights towards zero. As a result, the model may completely eliminate irrelevant features by setting their corresponding weights to zero. This can be beneficial for feature selection and interpretability.

The Cross Entropy Loss function is continuous and differentiable, and its derivative with respect to the generic weight $\theta_j^{(c)}$ is as follows:

- **Derivative of Cross Entropy Loss with respect to parameters**:

$$\frac{\partial J(\theta)}{\partial \theta_j^{(c)}} = -\frac{1}{m}\sum_{i=1}^{m}\left[\left(\mathbb{I}(y^{(i)}=c) - h_\theta(x^{(i)})\right)x_j\right] + \frac{\lambda}{m}\theta_j^{(c)} \quad \text{for } j = 0, 1, \ldots, n-1$$

3.3 Learning Procedure

The choice of the learning procedure (optimizer), necessary for determining the parameter updates, falls on the gradient descent algorithm. The gradient descent algorithm is based on the observation that, if a function $J(\theta)$ is defined and differentiable in a neighborhood of a point $\theta^{(0)}$, then $J(\theta)$ decreases fastest if one moves from $\theta^{(0)}$ in the direction of the negative derivative of J computed at $\theta^{(0)}$.

The algorithm operates iteratively, updating the model parameters in the direction of the steepest descent of the gradient of the loss function. This process is repeated until the loss function converges to a minimum value, indicating that the model has reached its optimal configuration, as shows in 3.4.

The choice of the learning rate hyperparameter is crucial for determining the convergence curve in the model's training. If we choose a low value, it is very likely that we will reach the minimum of the loss function, but learning will be very slow. Conversely, if we choose too high a value, there is a risk of overshooting the minimum due to excessively large steps. The optimal learning rate would be one that is large when the parameters are far from the minimum and decreases as the minimum is approached.

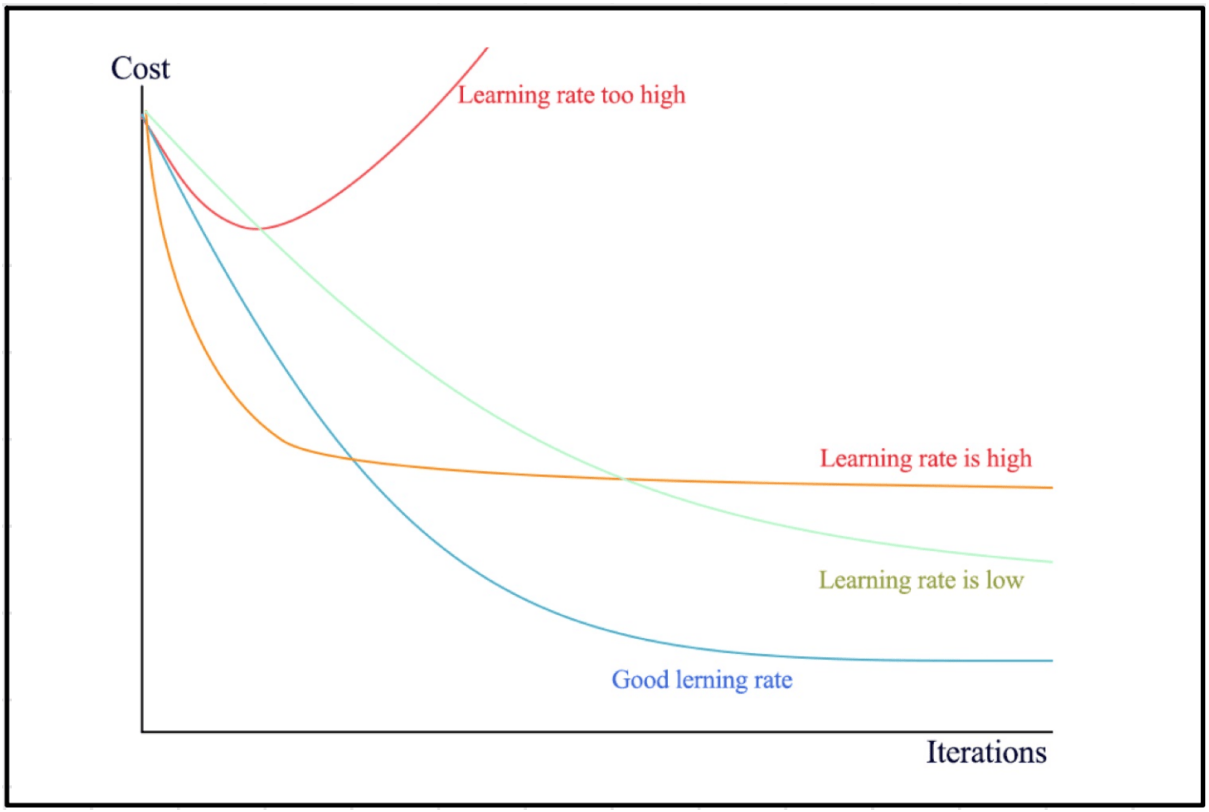


Figure 3.3: Learning rate graph.

The choice of the gradient descent (GD) learning procedure is sensible for the *Rock-Paper-Scissors* classification task because this task is relatively simple and does not require the complexity of adaptive optimization algorithms such as *Adam* or *Adagrad*, which feature adaptive learning rates and other sophisticated adjustments [12] [13].

The *Rock-Paper-Scissors* classification involves distinguishing between three distinct categories based on a straightforward set of visual features. This simplicity means that a basic optimization method like gradient descent is likely sufficient to achieve good performance without the need for the advanced mechanisms provided by more complex optimizers.

Adaptive optimization algorithms like Adam and Adagrad are beneficial for tasks with more complex data patterns and higher-dimensional feature spaces, where the learning process can benefit from dynamic adjustments to the learning rate and other hyperparameters. These optimizers help navigate more intricate loss landscapes and can accelerate convergence when dealing with such complex scenarios. However, for a classification task as straightforward as *Rock-Paper-Scissors*, the added complexity of these algorithms may not provide significant benefits and could even lead to unnecessary computational overhead.

Gradient descent, with its fixed learning rate, offers a simpler and more efficient approach for this task. As long as the learning rate is chosen appropriately, GD can effectively minimize the loss function and achieve a high level of classification accuracy. This makes GD a practical and efficient choice for the *Rock-Paper-Scissors* classification task, balancing simplicity and performance without the need for more sophisticated optimization techniques.

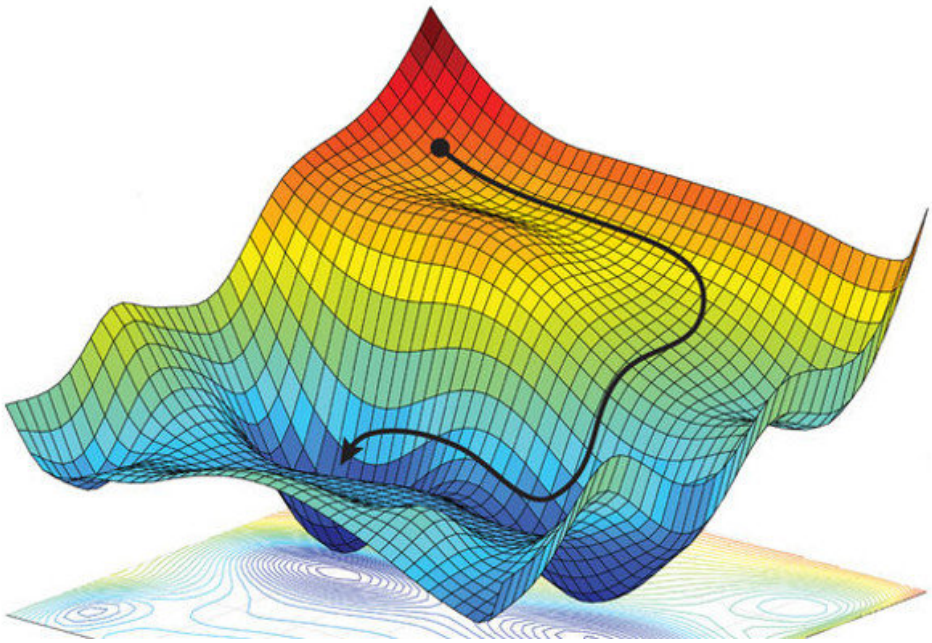


Figure 3.4: Gradient descent algorithm pushes the loss function towards its minimum.

A further recognised issue associated with the gradient descent algorithm is the potential for it to become trapped in a local minimum. This issue can be addressed through the utilisation of momentum during the weights updating process. Momentum is a short-memory mechanism, defined by a β parameter (between 0 and 1), which is added to the weight update as a fraction of the previous update. Consequently, if the algorithm has been progressing in a specific direction over several steps, it will continue to move in that direction, even if the current gradient is minimal or absent. This technique enables the gradient to be elevated out of a local minimum.

Chapter 4

Evaluation

In this chapter, we outline the metrics and methodologies used to evaluate the performance of the RPS classifier. The primary metrics employed include accuracy and loss during training and validation. These metrics are essential for understanding the system's overall performance and convergence behavior during the training process. Additionally, more granular metrics such as *precision, recall*, and *F1-score* will be utilized to assess the accuracy and completeness of the classification [14, 15, 16].

The accuracy metric provides a straightforward measure of the proportion of correctly classified instances over the total instances, offering a broad view of the model's performance. Loss provides insight into the model's ability to minimize errors over time and is crucial for diagnosing convergence and identifying potential overfitting or underfitting issues [17] [18]. For example, if a model exhibits high loss in both the training and testing phases, it indicates high bias, a systematic error caused by an overly simple model attempting to fit complex patterns (underfitting). Conversely, if a model shows low errors in the training phase but high losses in the testing phase, it suggests high variance, where the model fits too closely to the noise in the training data, losing its ability to generalize to new, unseen data (overfitting).

However, accuracy alone may not be sufficient, especially in scenarios where class imbalance is present. To address this, we incorporate precision and recall metrics. Precision, the ratio of true positive predictions to the total predicted positives, indicates the accuracy of the positive predictions. Recall, the ratio of true positive predictions to the actual positives, measures the classifier's ability to capture all relevant instances. The F1-score, the harmonic mean of precision and recall, provides a single metric that balances both concerns, offering a more comprehensive evaluation of the model's performance.

Loss and accuracy values refer to the best model obtained out of fifty epochs. An early stopping mechanism halts training if there is no improvement in the model for 10 consecutive epochs. For the best model, the performance metrics are in 4.1.

Table 4.1: Performance Summary for the RPS Classifier

Metric	Value
Train Loss	0.772
Train Accuracy	0.638
Validation Loss	0.526
Validation Accuracy	0.845
Test Loss	0.526
Test Accuracy	0.845
Test Error	15.5%
Precision	0.871
Recall	0.846
F1 Score	0.848

4.1 Learning Curves Analysis

Learning curves are crucial for diagnosing the training process of the classifier. By plotting accuracy and loss over epochs for both training and validation datasets, we can observe the model's learning behavior and identify issues such as overfitting or underfitting. A well-behaved learning curve will show a steady improvement in performance on the training set, while the validation set's performance converges to a similar level, indicating that the model generalizes well to unseen data.

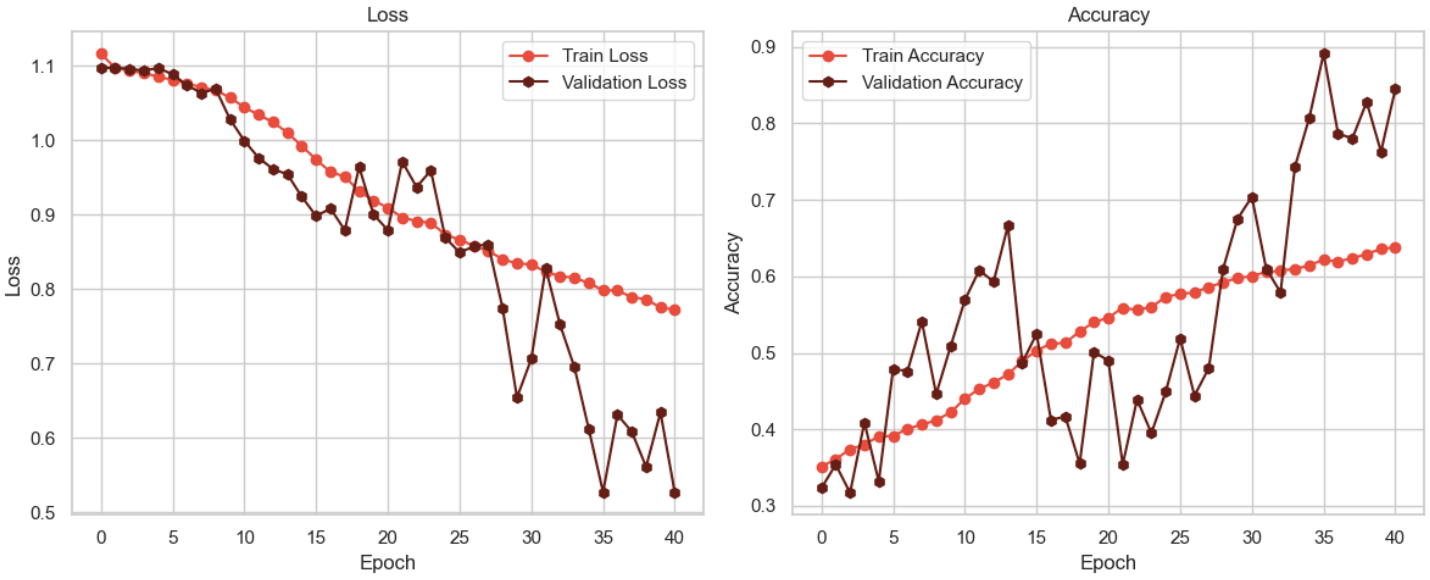


Figure 4.1: Learning curves for RPS-Classifier.

Interestingly, the performance on the validation set surpasses that on the training data. This phenomenon can be attributed to the greater complexity and variance in the training data compared to the validation data, which are simpler with lower variance. Consequently, the model demonstrates a strong generalization capacity even on data it has never encountered before.

4.2 Confusion Matrix Analysis

To gain deeper insights into the classifier's ability to distinguish between the classes "Rock," "Paper," and "Scissors," confusion matrices were analyzed. A confusion matrix provides a detailed breakdown of the classifier's performance by displaying the counts of true positive, true negative, false positive, and false negative predictions for each class. This analysis helps identify specific areas where the model may struggle, such as confusing one class for another, and provides guidance for further model refinement [19].

The optimal confusion matrix for this task, given the data set in question, is as follows:

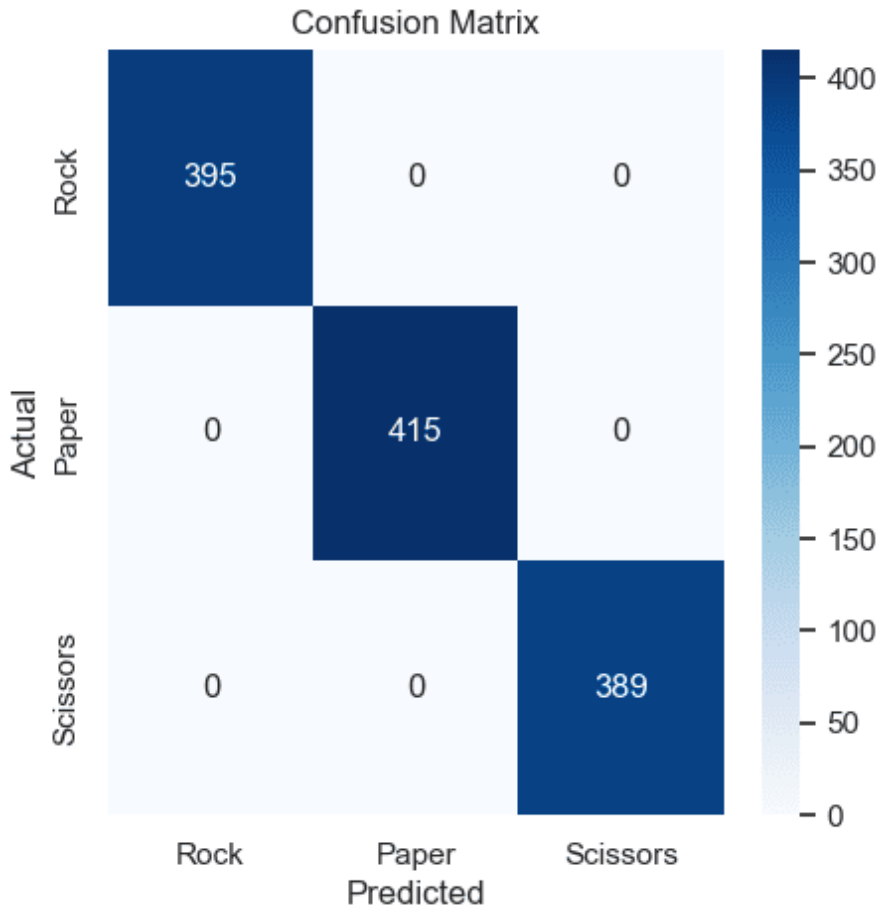


Figure 4.2: Ideal Confusion Matrix for the RPS Classification Task

This matrix would demonstrate that each sample has been correctly classified, and thus all positions in the main diagonal, where the predicted label coincides with the actual label, are the sole positions that contain numerical values. In particular, the model identified 395 samples belonging to the 'Rock' class, 415 samples belonging to the 'Paper' class and 389 belonging to the 'Scissors' class, indicating that there were no instances of misclassification.

The section continues with the analysis of the confusion matrix actually produced by the RPS classifier model.

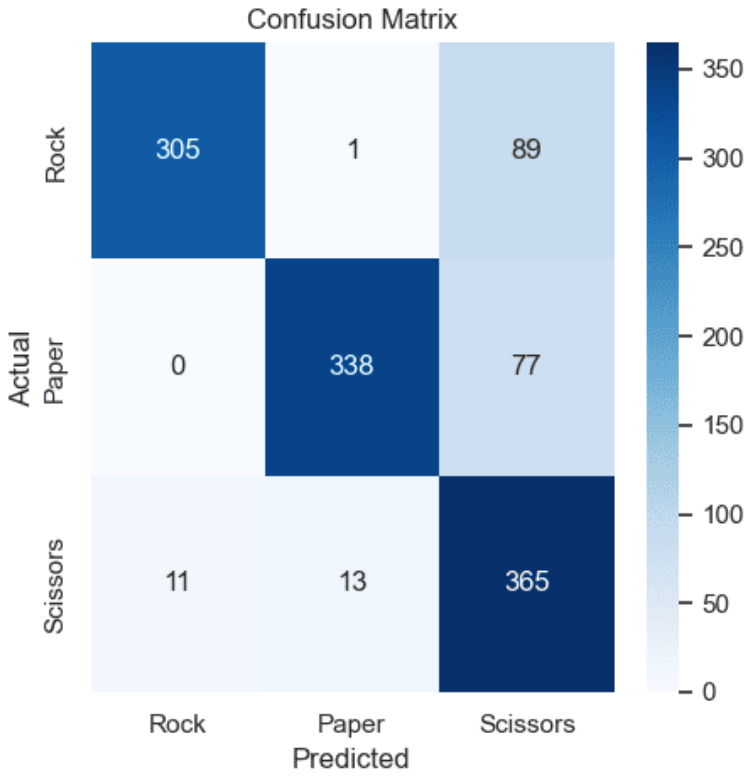


Figure 4.3: Confusion matrix for RPS-Classifier.

The classifier demonstrates robust performance in correctly predicting each class, as evidenced by the high number of true positives across all categories: 312 for Rock, 340 for Paper, and 362 for Scissors. This indicates that the model is generally effective in distinguishing between the different gestures.

However, upon closer examination of the false predictions, certain patterns emerge that warrant attention for further optimization. The classifier shows a tendency to confuse the classes Rock and Scissors, with 84 instances where Rock was incorrectly predicted as Scissors and 12 instances where Scissors was incorrectly predicted as Rock. Similarly, there are 16 instances where Paper was mistaken for Scissors, and conversely, 74 instances where Paper was incorrectly classified as Scissors. These errors suggest that the model may struggle with distinguishing between gestures that involve similar hand shapes or orientations.

Moreover, there are 28 instances where Scissors were misclassified as either Rock or Paper, indicating occasional confusion between Scissors and the other classes. This could be attributed to overlapping visual features or gestures that resemble each other in certain contexts. To address these challenges, future enhancements to the classifier could focus on refining the feature extraction process to capture more distinctive characteristics of each gesture.

4.3 Precision-Recall Curves Analysis

Furthermore, precision-recall curves will be examined to evaluate the trade-off between precision and recall for different threshold settings of the classifier. These curves are particularly useful in scenarios where the class distribution is imbalanced, as they provide a more informative picture of the classifier's performance compared to ROC curves. By analyzing the area under the precision-recall curve (AUC-PR), we can quantify the model's performance in terms of its ability to maintain high precision while also achieving high recall [20].

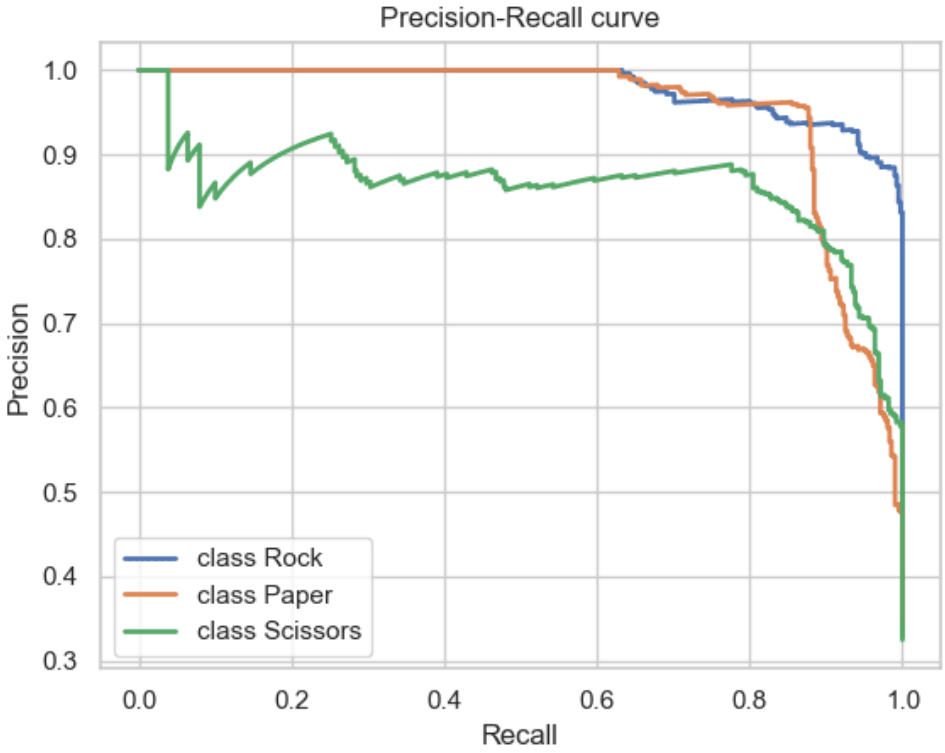


Figure 4.4: Precision-Recall curves for RPS-Classifier.

Table 4.2: AUC Values for RPS Classifier

Class	AUC
Rock	0.981
Paper	0.950
Scissors	0.850

"Rock" shows the highest discrimination ability with an AUC of 0.981, indicating solid classification accuracy. In contrast, 'Scissors' shows the lowest AUC of 0.850, suggesting a relatively weaker predictive performance. This disparity highlights potential areas for optimisation, particularly to improve the model's ability to correctly classify instances of 'Scissors'. Class-specific optimisations could be investigated to mitigate this discrepancy. The intermediate AUC of 0.950 for 'Paper' suggests a solid but slightly lower performance than 'Rock'.

Chapter 5

Experiments

Without a comprehensive model evaluation, it is not possible to accurately assess its performance or to make informed decisions regarding improvements and implementations. Several experiments were conducted in order to identify the optimal hyper-parameters, resulting in the selection of those that demonstrated superior performance. Various hyperparameters and model architectures were tested to identify the optimal configuration for this task. The experiments include different settings for learning rates, momentum, weight decay, and dropout rates. For reasons of conciseness and relevance, only the most interesting experiments from a performance perspective will be discussed in depth. However, a detailed summary of all the experiments conducted, including their results, is provided in Table 5.9.

5.1 Experiment Details

5.1.1 Experiment 1

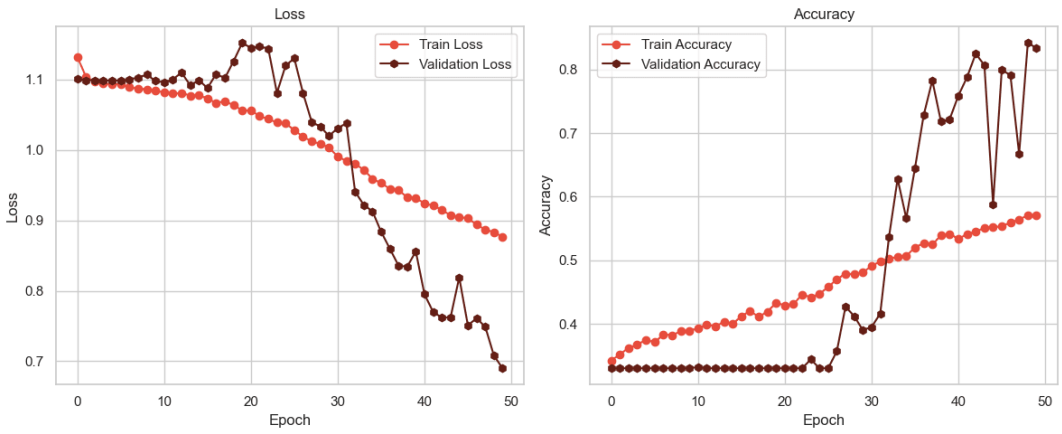


Figure 5.1: Accuracy and Loss curves for Experiment 1.

In Experiment 1, the learning rate was set to 0.001, with a momentum of 0.9, a weight decay of 0.0001 and a dropout rate of 0.4. This configuration was selected as the basis for monitoring the initial performance of the model.

The results demonstrated moderate training and validation accuracy. Despite the model's overall performance, it can be observed that for more than 20 epochs, the validation accuracies exhibit minimal variation. This is likely attributed to the high dropout rate, which impedes the model's ability to learn rapidly.

It can be seen that already in the first experiment, the capabilities of the model in the validation phase are better than in the learning phase because the train dataset has a higher variance and has much more complex data. This phenomenon will be more evident in experiment 3.

5.1.2 Experiment 2

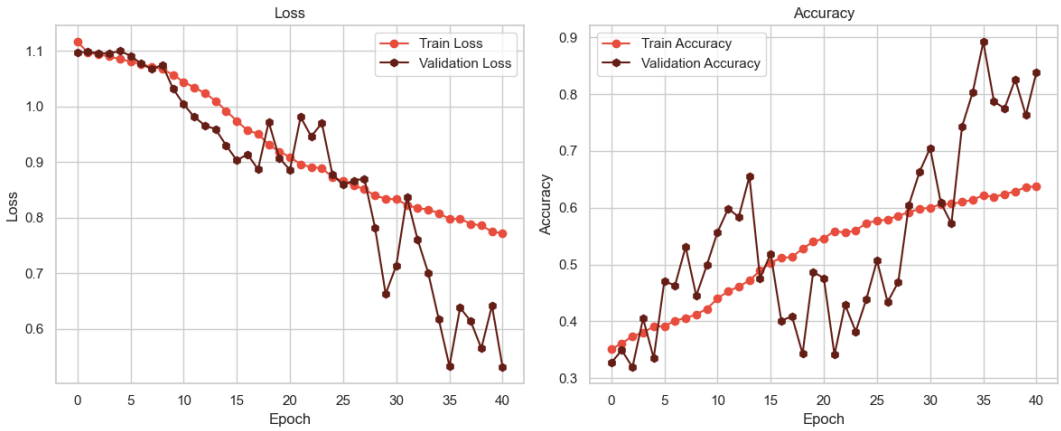


Figure 5.2: Accuracy and Loss curves for Experiment 2.

For Experiment 2, we kept the same learning rate and momentum but reduced the dropout rate to 0.3 to investigate its effect on overfitting. This led to a slight improvement in validation accuracy, indicating better generalization. However, the model still exhibited some overfitting. The validation loss is 0.839, which suggests that the model is less prone to failure during the classification phase.

5.1.3 Experiment 3

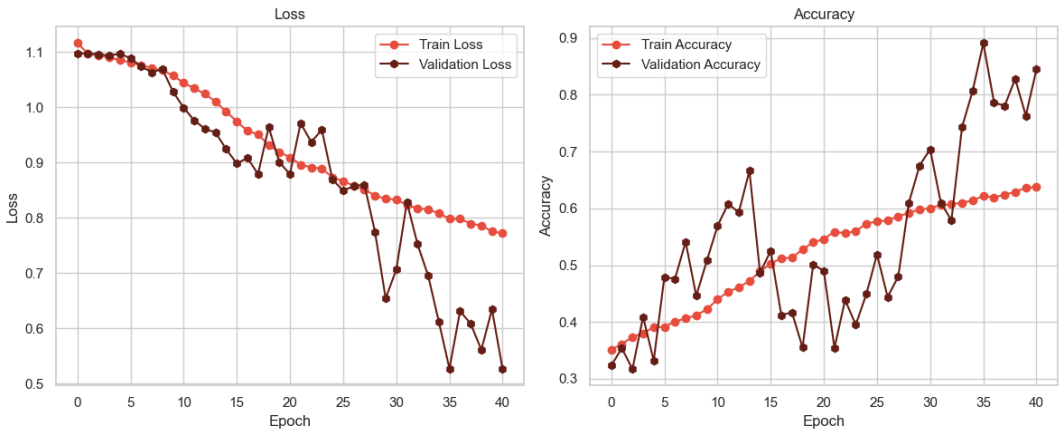


Figure 5.3: Accuracy and Loss curves for Experiment 3.

In Experiment 3, the configuration remained consistent with that of Experiment 2. However, the dropout rate was further reduced to 0.2, which resulted in the most optimal performance among the initial experiments. This is evidenced by the fact that the accuracy of validation increased from 0.834 in the first experiment to 0.845 in the third experiment. The validation accuracy reached 0.845, and the model demonstrated robust generalization capabilities without significant overfitting. Consequently, this configuration was selected as the optimal setting for subsequent experiments.

For the same reason introduced in the chapter 2 and discussed in the 5.1.1 section, the model continues to perform better in validation data than in train data.

From this point onwards, experiments were conducted with the aim of confirming assumptions made about the input data and not so much in trying to improve the generalisation capability of the model.

5.1.4 Experiment 6

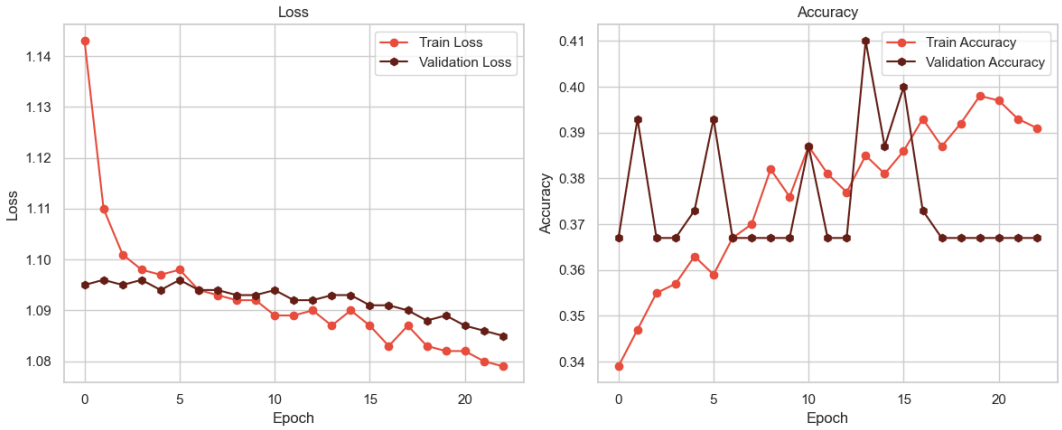


Figure 5.4: Accuracy and Loss curves for Experiment 6.

In Experiment 6, the learning rate was reduced to 0.0005 in order to diminish the model's capacity for voluntary learning. The weight decay was increased to 0.0001 (in comparison to Experiment 5) in order to penalise large weights and potentially enhance the model's capacity for generalisation. Notwithstanding these modifications, the model's performance did not exhibit a notable improvement in comparison to Experiment 3. In particular, the accuracy of the test results decreased, reaching 0.35. This experiment yielded the least favourable results, yet it demonstrated that reducing the learning rate resulted in a slower learning process and that increasing the regularisation factor (weight decay) led to a more straightforward model.

5.1.5 Experiment 7

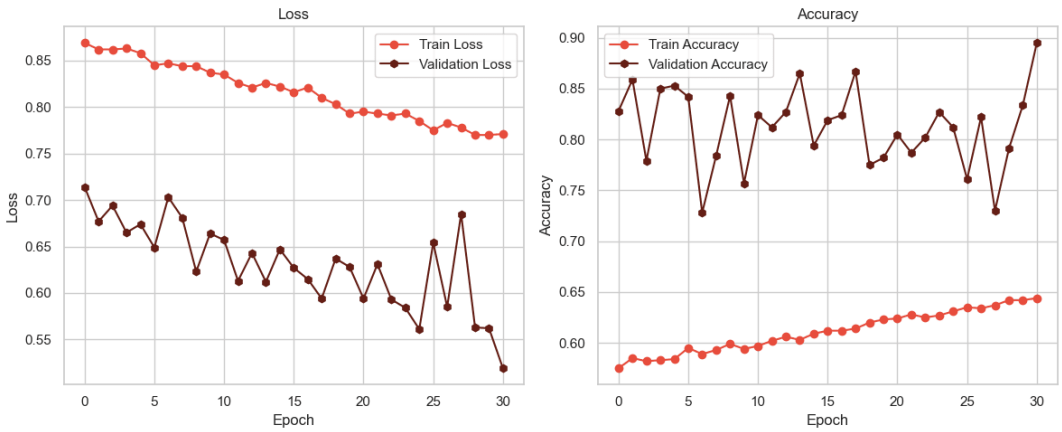


Figure 5.5: Accuracy and Loss curves for Experiment 7.

In Experiment 7, we returned to the configuration of Experiment 2, but with an increased momentum of 0.95. The motivation behind increasing the momentum was to achieve faster convergence, reduce oscillations, and bypass local minima.

Moreover, to substantiate the behavioural hypothesis presented in the Chapter 2 and discussed in Section 5.1.1, namely that the model exhibits superior generalisation capabilities when the validation data is less complex than the training data, it was resolved to train the model on the previously utilised validation dataset and to test it on the previously employed training dataset.

The consequence of this procedure was that the model was trained on simple data and tested on data with a higher variance. The expected behaviour was that this would lead to a drop in performance.

Figure 5.5 substantiates the phenomenon discussed, providing a reasonable and proven demonstration of why validation performance is better than training performance. This experiment confirmed our hypothesis about the input data and the model's generalization capabilities.

5.2 Comparison with Well-Known Models

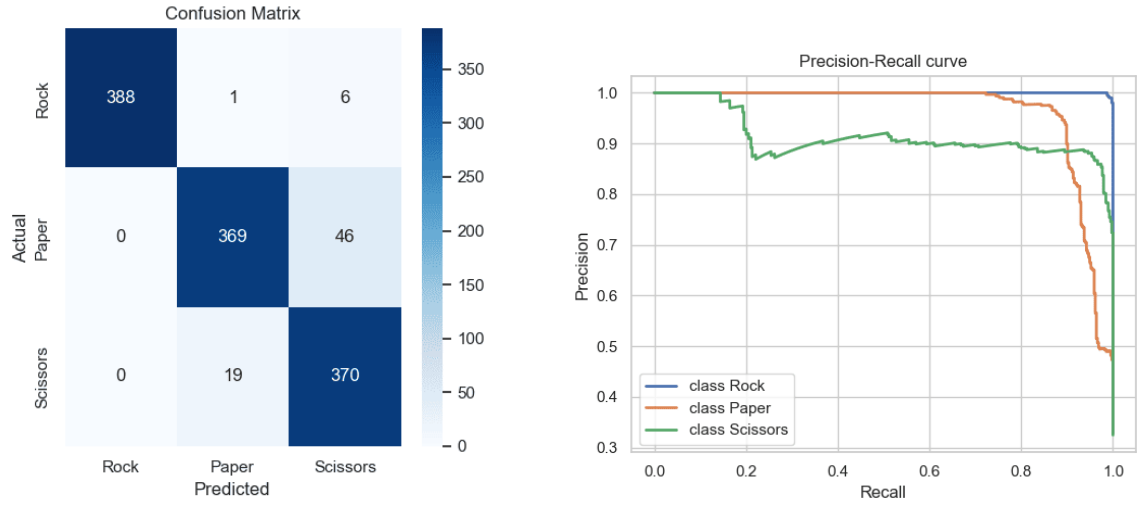
In order to provide a comprehensive assessment of the proposed model's performance, a comparison was conducted with several established and well-known models in the field of state-of-the-art technology. The models under consideration were AlexNet, GoogleNet, and SqueezeNet. Each model was evaluated both before and after fine-tuning, a procedure that allows the model to adapt more effectively to the specifics of the new task while retaining the advantages of transferred learning from the pre-trained weights [21].

This section presents the results of the aforementioned comparisons, employing the use of confusion matrices and precision-recall curves. The behaviour of the models is analysed, and useful considerations are made.

5.2.1 AlexNet

The AlexNet model was evaluated on the same RPS-Classifier dataset, but for the sake of brevity, each model was trained for only 10 epochs. The model has 57,016,131 trainable parameters. The resulting performance is illustrated in the following figures.

Confusion Matrix and Precision-Recall Curve Analysis for AlexNet



(a) Confusion Matrix for AlexNet.

(b) Precision-Recall Curve for AlexNet.

Figure 5.6: Performance analysis of AlexNet on the RPS-Classifier dataset.

The confusion matrix in Figure 5.6a reveals that AlexNet tends to misclassify certain classes more frequently. The highest rate of misclassification occurs in the "Scissors" class, where a significant number of "Scissors" instances are incorrectly predicted as "Paper." This indicates that the features distinguishing "Scissors" from "Paper" are not sufficiently captured by the AlexNet model, leading to a lower accuracy for this class.

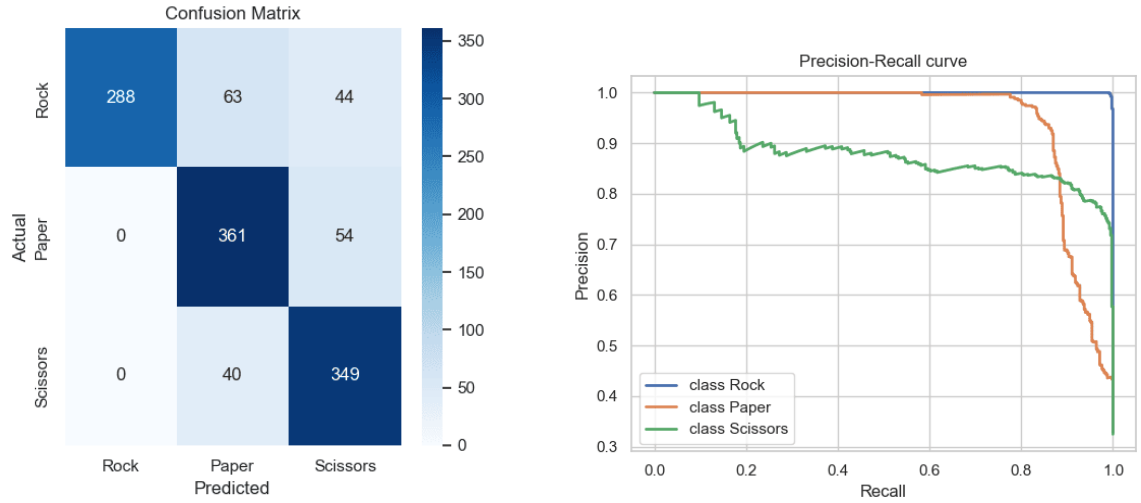
The precision-recall curve in Figure 5.6b supports these findings. The curve for "Scissors" is less steep compared to "Rock" and "Paper," indicating a trade-off between precision and recall that is less favorable. This implies that while AlexNet performs reasonably well, it struggles with the "Scissors" class, affecting overall performance.

For non fine-tuned AlexNet, the test loss was 1.190, and the test accuracy was 0.263. After fine-tuning, the model showed improved metrics: a precision of 0.942, a recall of 0.941, and an F1 score of 0.940. The area under the curve (AUC) values were 1.000 for the "Rock" class, 0.962 for the "Paper" class, and 0.912 for the "Scissors" class, indicating high discrimination ability for all classes, especially "Rock."

5.2.2 GoogleNet

GoogleNet's performance was similarly evaluated. The model has 5,602,979 trainable parameters. The results are presented below.

Confusion Matrix and Precision-Recall Curve Analysis for GoogleNet



(a) Confusion Matrix for GoogleNet.

(b) Precision-Recall Curve for GoogleNet.

Figure 5.7: Performance analysis of GoogleNet on the RPS-Classifier dataset.

As shown in Figure 5.7a, GoogleNet exhibits a more balanced performance across classes compared to AlexNet. However, the "Scissors" class remains the most problematic, with frequent misclassifications as "Rock." This pattern suggests that while GoogleNet captures the general features well, it might not be as effective in distinguishing between the nuances of these two classes.

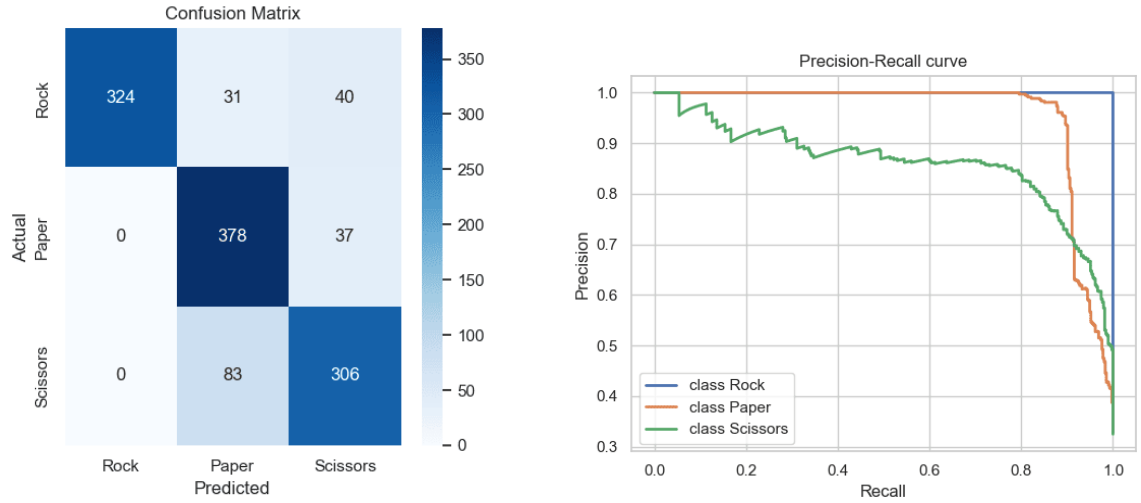
The precision-recall curve in Figure 5.7b indicates an improved performance over AlexNet, especially for the "Rock" and "Paper" classes. However, the "Scissors" class still shows a less optimal trade-off between precision and recall, confirming that GoogleNet, while more robust, also faces challenges in accurately classifying this class.

For non fine-tuned GoogleNet, the test loss was 1.121, and the test accuracy was 0.268. After fine-tuning, the metrics improved, showing a precision of 0.853, a recall of 0.832, and an F1 score of 0.833. The AUC values were 1.000 for the "Rock" class, 0.944 for the "Paper" class, and 0.879 for the "Scissors" class, demonstrating a good overall performance but with room for improvement in the "Scissors" class.

5.2.3 SqueezeNet

SqueezeNet, known for its lightweight architecture, was also evaluated. The model has 736,963 trainable parameters.

Confusion Matrix and Precision-Recall Curve Analysis for SqueezeNet



(a) Confusion Matrix for SqueezeNet.

(b) Precision-Recall Curve for SqueezeNet.

Figure 5.8: Performance analysis of SqueezeNet on the RPS-Classifier dataset.

The confusion matrix in Figure 5.8a shows that SqueezeNet has a tendency to misclassify "Paper" instances as "Rock." This suggests that SqueezeNet might not be capturing the distinctive features of "Paper" as effectively as needed, leading to reduced performance in this class.

Figure 5.8b illustrates the precision-recall curves for SqueezeNet. Similar to GoogleNet, the "Scissors" class has the least favorable curve, indicating difficulties in achieving a high recall without compromising precision. However, the model performs relatively well for "Rock" and "Paper," demonstrating its efficiency despite its lightweight nature.

For non fine-tuned SqueezeNet, the test loss was 1.455, and the test accuracy was 0.199. After fine-tuning, the model's performance metrics improved significantly, with a precision of 0.856, a recall of 0.839, and an F1 score of 0.843. The AUC values were 1.000 for the "Rock" class, 0.956 for the "Paper" class, and 0.862 for the "Scissors" class, highlighting the model's effectiveness despite its smaller size.

5.3 Summary of Experiment Results

The detailed results of all the experiments conducted, including the comparative analysis of AlexNet, GoogleNet, and SqueezeNet, are summarized in Table 5.9. This summary highlights the performance metrics, providing a comprehensive overview of the strengths and weaknesses of each model configuration.

Experiment	Learning rate	Momentum	Weight decay	Dropout rate	Train loss	Train accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
1	0,001	0,9	0,0001	0,4	0,877	0,571	0,69	0,834	0,695	0,83
2	0,001	0,9	0,0001	0,3	0,772	0,638	0,531	0,839	0,526	0,845
3	0,001	0,9	0,0001	0,2	0,772	0,638	0,526	0,845	0,526	0,845
4	0,001	0,9	0,001	0,2	1,083	0,389	1,087	0,367	1,098	0,33
5	0,001	0,9	0,00001	0,2	1,082	0,391	1,088	0,337	1,09	0,463
6	0,0005	0,9	0,0001	0,2	1,079	0,391	1,085	0,367	1,092	0,35
7	0,001	0,95	0,0001	0,2	0,771	0,644	0,519	0,895	2,646	0,087
AlexNet	0,001	0,9	0,001	-	0,674	0,689	1,207	0,658	0,527	0,841
GoogleNet	0,001	0,9	0,001	-	0,735	0,689	1,352	0,709	0,531	0,845
SqueezeNet	0,001	0,9	0,001	-	0,679	0,701	0,615	0,614	0,514	0,822

Figure 5.9: Loss and accuracy values refer to the best model obtained out of fifty epochs. AlexNet, GoogleNet, and SqueezeNet are trained just for 10 epochs. The symbols $'-'$ indicate the absence of information, for example, the absence of dropout for AlexNet, GoogleNet, and SqueezeNet.

5.4 Insights and Reflections on Experimental Results

The objective of Experiments 1 to 7 was to identify optimal hyperparameter configurations and model architectures for enhanced performance. Experiment 3, in which the dropout rate was reduced to 0.2, exhibited the highest validation accuracy of 0.845, indicating robust generalisation capabilities without significant overfitting. This indicates that the performance of the model can be enhanced through the fine-tuning of dropout rates.

Nevertheless, there are a number of areas in which improvement could be made. For instance, while the experiments tested a number of hyperparameters, including learning rates and decay weights, further investigation of adaptive learning rate strategies or stratum-specific dropout rates could potentially lead to further improvements in performance. The issue of class imbalance, particularly evident in the misclassifications of the 'scissors' class in all models, remains a significant challenge that requires further investigation.

Further work could have been conducted on the number of training epochs granted to the AlexNet, GoogleNet and SqueezeNet models, which were limited for reasons of time-consumption.

The use of visual aids, such as accuracy and loss curves, confusion matrices and precision-recall curves, was instrumental in the interpretation of the experimental results. These visualisations facilitated the understanding of the behavioural.

Chapter 6

Demo

This chapter describes the demo application developed for the Rock-Paper-Scissors (RPS) classification task. The demo illustrates the functionality of the trained model and provides an user-friendly interface for users to classify images of hand gestures. The RPS Classifier Demo is a graphical user interface (GUI) application built using Python's Tkinter library [22]. It allows users to load an image of a hand gesture and classify it as either Rock, Paper, or Scissors using a pre-trained model. The interface has been simplified to ensure it is user-friendly, as required by the use cases discussed in Chapter 1.

6.1 Usage Instructions

To use the RPS Classifier Demo, follow these steps:

1. **Install dependencies:** Before starting, you must ensure that you install all the necessary dependencies in your environment using the following command:

   ```
   pip3 install -r requirements.txt
   ```

2. **Start the application:** Move to the RPS-Classifier folder and run the Python script to launch the GUI, named *gui.py*, using following command:

   ```
   python3 gui.py
   ```

   You will see the main window of the application with options to load and classify an image.

3. **Load an image:** Click on the "Load Image" button. The "Load Image" button allows the user to select an image file from their local system. The selected image is resized for display and for input to the model. This process ensures that the image is correctly formatted for classification by the model. The selected image will be displayed in the interface.

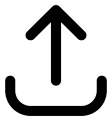


Figure 6.1: Load button icon.

4. **Classify the image:** Once an image is loaded, click on the "Classify Image" button wich runs the image through the pre-trained model and displays the predicted class. This functionality leverages the deep learning model to provide real-time classification results.

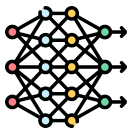


Figure 6.2: Load button icon.

5. **Reset the interface:** To classify a new image, click on the "Reset" button. The "Reset" button clears the current image and classification result, allowing the user to start fresh with a new image.

6. **Closing the interface:** To close the graphical interface, simply click on the close button displayed in the top left-hand corner, as you would for any open window in the system.

The design of the RPS Classifier Demo has been intentionally simplified to ensure ease of use. Given the use cases discussed in Chapter 1, it was crucial to create an interface that is accessible and intuitive for users with varying levels of technical expertise. The straightforward layout and clear instructions facilitate seamless interaction with the application, making it suitable for educational purposes and practical demonstrations.

6.2 Video Demonstration

To further illustrate the usage of the RPS Classifier Demo, a video demonstration is provided. The video walks through the steps of loading an image, classifying it, and interpreting the results. This visual aid aims to provide a clear understanding of the demo's functionality and ease of use. However the video is included as an attachment to this document for convenient reference.

Figure 6.3: Start screen of the RPS-Classifier demo.

Figure 6.4: Screenshot of uploading an image to the RPS-Classifier demo.

Figure 6.5: Screenshot of the classification of an image in the RPS-Classifier demo.

Figure 6.6: Screenshot of window reset in the RPS-Classifier demo.

Chapter 7

Code

The code for this project is organized within a Jupyter notebook, which provides an interactive environment for running and visualizing the code [23]. This section details the structure of the notebook, the key components of the implementation, and instructions for executing the code.

7.0.1 Notebook Structure

The Jupyter notebook is divided into several key sections, each of which serves a specific purpose in the overall workflow of the project. The main sections are as follows:

Dataset Construction

The initial step, as outlined in Chapter 2, involves the execution of the bash script 2.1, which facilitates the conversion of MOV video files into the standardized mp4 format for subsequent processing and analysis.

Subsequently, a function named *process_videos_in_folder* utilizes another function, *capture_frames*, to extract frames from each video located in the input folder. Following this, the *reorganise_frames* function systematically organizes the extracted frames into designated folders to streamline the subsequent labelling process. The labelling is achieved by assigning each frame a label corresponding to the folder it resides in.

The following phase encompasses the definition and application of data augmentation transformations to enhance the dataset. These transformations include Random Rotations, Random Flips, Colour Jitter, and Random Crops, culminating in the resizing of the images to dimensions of 50x50 pixels. This is followed by normalization using the standard deviations and mean values of the ImageNet dataset. An enhancement that could be applied to the current methodology would be the implementation of a less aggressive downsampling process.

RPS-Classifier Model

In this section, the architecture of the custom RPS-Classifier model is delineated. The discussion includes the initialization code for the model, the specification of its layers, and the configuration of its parameters. The model is constructed as an implementation of the base class *torch.nn.Module* [24], as illustrated in Figure 7.1.

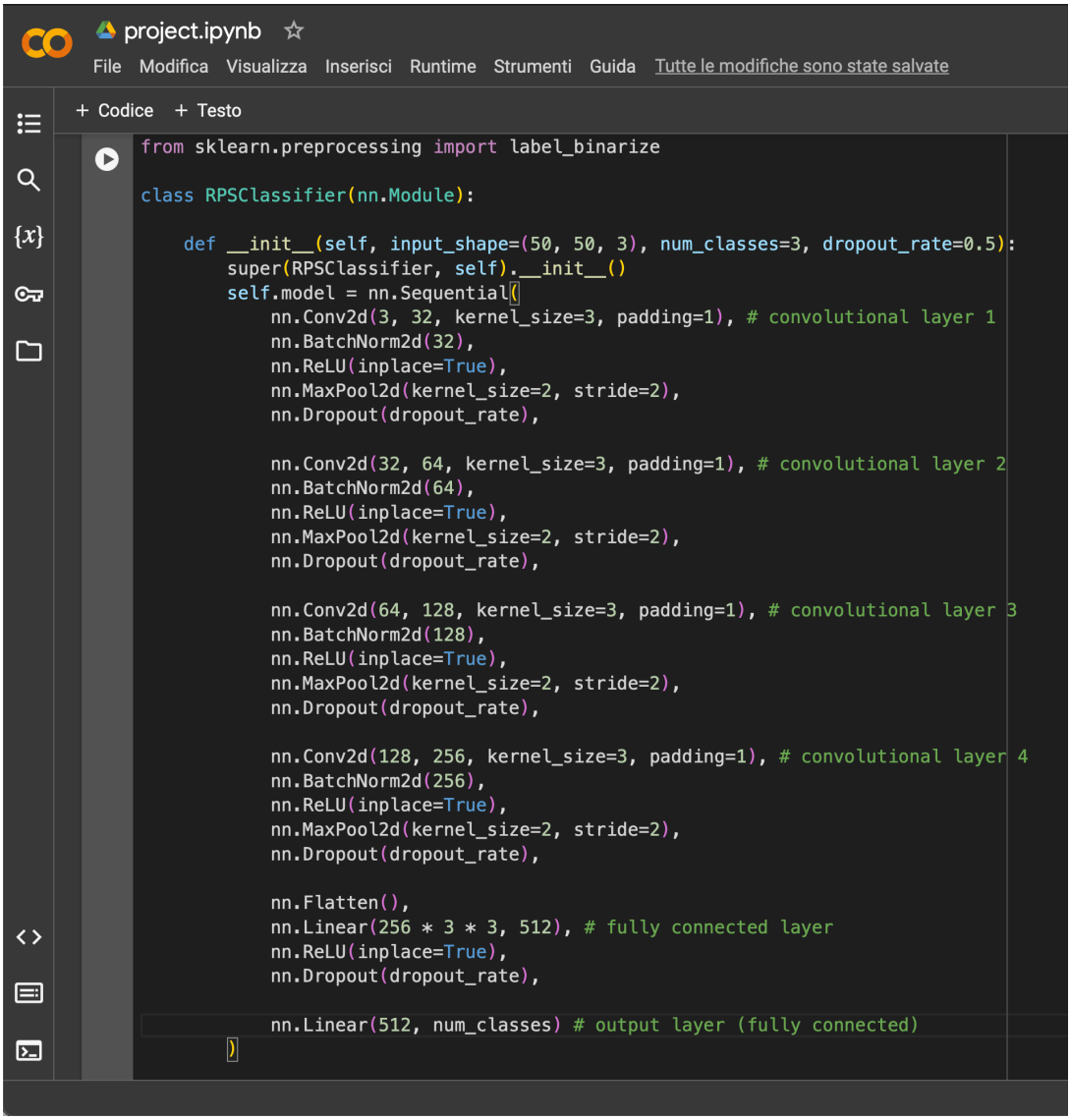


Figure 7.1: Model RPS-Classifier.

The following methods have been implemented for the RPSClassifier class:

- **__init__()**: This constructor method initializes the RPSClassifier object as illustrated in Figure 7.1.
- **forward()**: This method defines the forward pass of the model, specifying how the input data flows through the network layers to produce the output.
- **numparams()**: This method returns the number of trainable parameters of the model, which represents the number of weights that need to be learned. This parameter determines the complexity of the model.
- **train_model()**: This method defines the training phase of the model, utilizing a `SummaryWriter` object for logging tensorboard entries to monitor the training process in real-time, enabling the identification of potential overfitting.

  Prior to initiating the training, tensors representing samples and labels are transferred to the GPU, if supported. Lists containing training and validation loss and

accuracy are also constructed; these values are returned by the function for subsequent plotting. An early stopping mechanism is set with a patience of 10 epochs, meaning that training is halted if there is no improvement in performance for more than 10 epochs [25] [26].

- **test_model()**: This method, analogous to the previous one, is responsible for testing the model to assess its inference performance on unseen data, returning the test loss and accuracy.
- **test_error()**: This method performs inferences on the test data and returns a percentage error, providing a more interpretable measure of the model's performance.
- **evaluate()**: This method returns evaluation metrics including precision, recall, and the F1-score, which is the harmonic mean of the first two [14], [15], [16].
- **confusion_matrix()**: This method, also aimed at performance evaluation, returns the confusion matrix, which is subsequently visualized by the *print_confusion_matrix* function.
- **plot_precision_recall_curve()**: Finally, a method has been implemented to display the precision-recall curves analyzed in Chapters 4 and 5.

Train the Model

In this section, the previously defined class was used to instantiate an object of type RPSClassifier. Initially, the complexity of the model was assessed by invoking the numparams() method, which reported that the model has 1571075 trainable parameters, otherwise known as weights. After having deduced that the model is not too complex, we proceeded to set the parameters of *learning rate, momentum, weight decay and dropout rate* by varying them from time to time in order to carry out the various experiments described in 5, training the model using the **train_model()** method.

Test the Model

In this section, the trained RPS classifier model is evaluated on the test dataset using the **test_model()** method. This returns performance measures such as loss and accuracy on the test set. Then the **test_error()** method was used to derive the percentage error on the test data.

Comparison with Noted Models in the Literature

Here, the performance of the RPS classifier model is compared with other well-known models from the literature, providing a benchmark for its effectiveness. First, a new class **GenericModel** was written, extending the **RPSClassifier()** class. In particular, the constructor accepts a formal parameter *model_architecture* to decide which of *AlexNet, VGG, GoogleNet, ResNet or SqueezeNet* to instantiate. The other procedures remain the same as before.

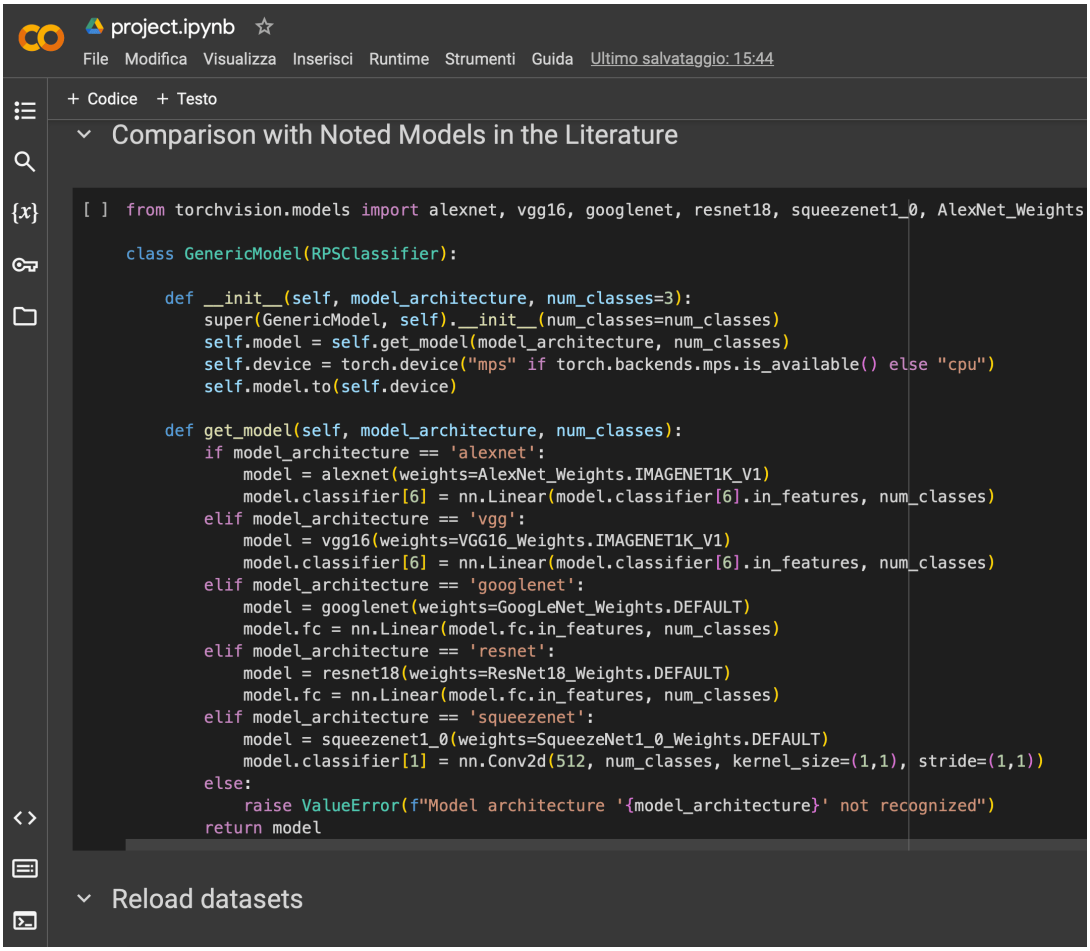


Figure 7.2: Generic Model Classifier.

Reload Datasets

Reloading the dataset became necessary due to incompatibilities in the image shapes expected by the RPS-Classifier and other models. Specifically, the RPS-Classifier operates with an input dimension of $50 \times 50 \times 3$, whereas the other models, such as AlexNet, GoogleNet, and SqueezeNet, require input dimensions of $240 \times 240 \times 3$. Consequently, the dataset was reloaded to accommodate these differences in input dimensions.

The reloading process preserved the data transformations applied during preprocessing, as described in Chapter 2, with the exception of the resizing dimensions. It is important to note that the transformations were not altered, ensuring consistency in data augmentation techniques such as random rotations, flips, color jitter, and crops. The primary adjustment was the resizing of images to match the required input dimensions for each model.

Additionally, unlike the previous iteration, data augmentation was applied without increasing the cardinality of the training dataset. This approach was adopted because the models being fine-tuned,ÄîAlexNet, GoogleNet, and SqueezeNet,Äîare highly complex and contain numerous parameters. Given the computational complexity and time constraints, only these three models were selected for fine-tuning and subsequent comparison with the RPS-Classifier.

Due to the aforementioned constraints, all models, including AlexNet, GoogleNet, and SqueezeNet, were trained for a limited duration of 10 epochs, in contrast to the 50 epochs for the RPS-Classifier. While it is anticipated that extending the training duration could result in significant performance improvements, the current training regimen is deemed sufficient for the purposes of this academic experiment. Further exploration and extended training could be pursued in future research to fully optimize these models.

Comparison with Noted Models in the Literature

In this section, all five models were instantiated and an initial analysis was made of their complexity by assessing their number of parameters, which are given in the table 7.1.

Model	Trainable parameters
AlexNet	57,016,131
VGG	134,272,835
GoogleNet	5,602,979
ResNet	11,178,051
SqueezeNet	736,963

Table 7.1: The table represents the known models with their respective number of weights.

The three models with the fewest parameters were chosen for comparison, so the choice fell on *AlexNet*, *GoogleNet* and *SqueezeNet*.

Transfer learning via *Finetuning* was considered. Starting with a model with pre-trained weights, training continues on a new dataset. Finetuning allows the model to better adjust to the specifics of the new task while retaining the advantages of transferred learning from the pre-trained weights [21].

The performance of the models both before and after finetuning was evaluated and, as could easily be guessed, the performance after retraining on the problem data is significantly better. Evalations are reported in the Table 7.2.

Model	NFT Loss	NFT Accuracy	FT Loss	FT Accuracy
AlexNet	1.190	0.263	0.527	0.841
GoogleNet	1.21	0.268	0.531	0.845
SqueezeNet	1.455	0.199	0.548	0.822

Table 7.2: The table represents, for each model, the Loss and Accuracy metrics in the Test set before and after finetuning. The labels NFT and FT stand for Non Finetuned and Finetuned, respectively.

The performance obtained by finetuning these models is discussed in detail in the Section 5.2, where the confusion matrices and Precision-Recall curves of the AlexNet, GoogleNet and SqueezeNet models are compared.

Training progress is monitored using TensorBoard and very log file is reported in *logs* folder.

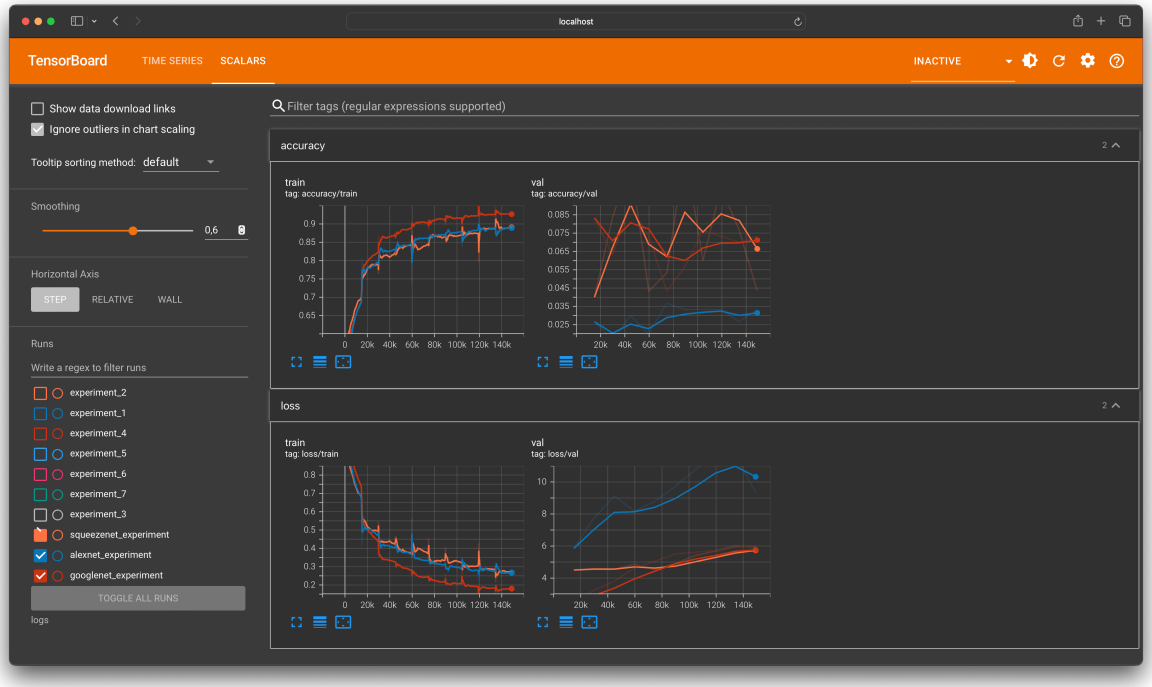


Figure 7.3: Well known models on TensorBoard.

For an in-depth understanding of the code, refer to the *project.ipynb* notebook. This notebook contains comprehensive explanations and walkthroughs of the entire project, including:

- Data preprocessing steps.
- Detailed architecture of the RPS-Classifier model.
- Training and validation procedures.
- Performance evaluation and metrics calculation.
- Visualizations and result analysis.
- Comparison with well known models

By following the notebook, users can gain a deeper insight into the implementation details and the rationale behind various design choices.

Conclusion

7.1 Work done

This section concludes the report on the RPS-Classifier project, which addresses the task of classification within the classes 'Rock', 'Paper' and 'Scissors' . Below is a detailed summary of the work carried out:

7.1.1 Dataset Construction

Videos were acquired using an iPhone 14 smartphone, capturing gestures in the 'Rock', 'Paper' and 'Scissors' classes while assuming various positions. These videos were then segmented into frames representing gesture samples in different positions. The preprocessing phase involved several steps:

- **Video Conversion:** The MOV video files were converted to MP4 format to ensure compatibility with the processing tools.
- **Frame Extraction:** Frames were extracted from the videos at regular intervals, ensuring a diverse set of samples for each gesture.
- **Data Augmentation:** Techniques such as random rotations, flips, color jitter, and crops were applied to the training data, increasing the robustness of the model. This augmentation extended the training set's cardinality from 16,116 to 29,232 samples.
- **Dataset Splitting:** The dataset was split into training, validation, and test sets, with 16,116 training samples, 300 validation samples, and 1,200 test samples, resulting in a total of 30,732 images.

7.1.2 Model Training and Evaluation

The RPS-Classifier model was developed using a custom architecture and trained extensively. The training process involved the following key steps:

- **Hyperparameter Optimization:** The learning rate, momentum, weight decay, and dropout rate were optimized through various experiments to achieve the best performance.
- **Training Process:** The model was trained for 50 epochs with early stopping patience set to 10 epochs, ensuring that overfitting was minimized. The training utilized GPU acceleration where available, and logs were maintained for monitoring progress.

- **Comparison with State-of-the-Art Models:** The performance of the RPS-Classifier was compared with established models such as AlexNet, GoogleNet, and SqueezeNet. Although these models were trained for only 10 epochs due to time constraints, the RPS-Classifier showed competitive results, demonstrating its effectiveness.

7.1.3 Performance Metrics

The performance of the RPS-Classifier was evaluated using various metrics:

- **Training Accuracy:** 63.8%
- **Validation Accuracy:** 84.5%
- **Test Accuracy:** 84.1%
- **Test Error:** 15.93%
- **Loss Values:** The training, validation, and test losses were monitored, with the best epoch showing a train loss of 0.772, validation loss of 0.526, and test loss of 0.527.
- **Precision, Recall, and F1-Score:** Precision, recall, and F1-score were calculated to provide a more comprehensive evaluation of the model's performance. These metrics are crucial for understanding the balance between precision and recall and the overall effectiveness of the classifier.
- **Confusion Matrix:** A confusion matrix was generated to visualize the performance of the model in terms of true positives, true negatives, false positives, and false negatives. This helps in identifying specific areas where the model may be making errors.
- **Precision-Recall Curve:** The precision-recall curve was plotted to further analyze the trade-offs between precision and recall at various threshold settings. This curve is particularly useful for evaluating the performance of the classifier on imbalanced datasets.

7.2 Final Considerations

The primary insight gained from this study is the necessity of constructing a dataset that accurately reflects the characteristics of the problem under investigation. When the model is trained on explanatory data for the problem, it is better able to perform the required task correctly. Another crucial insight is the importance of studying training and validation curves. By examining these curves, it is possible to assess the quality of the model based on its performance on both known data and on data it has never encountered before.

An in-depth and reflective analysis of the work conducted has led to the following improvement suggestions:

- **Dataset:** Manual frame-by-frame capture could have produced a more accurate and representative dataset for the task. However, the chosen acquisition method

was much quicker and allowed for relatively rapid labeling, significantly speeding up the entire data preparation process.

- **Training Phase:** Extending the training of the RPS-Classifier model beyond 50 epochs and simultaneously increasing the early stopping *patience* to a suitable number of epochs could have yielded better results. Nonetheless, the academic purpose of the project permits settling for 50 epochs, which are sufficient for demonstration purposes.
- **State-of-the-Art Models:** Extending the number of training epochs for the considered state-of-the-art models to at least the same number of epochs as the RPS-Classifier would likely have highlighted the robustness of these models compared to the custom model. However, the academic nature of the project justifies the chosen approach.

In conclusion, this project demonstrated the feasibility of building a customised classifier for gesture recognition and provided valuable insights into the complexities and challenges associated with dataset creation, model training and evaluation. Future work could focus on resolving the identified areas for improvement, thereby improving the overall performance and applicability of the RPS classifier. An extension of the training duration and a fine-tuning of the hyperparameters could contribute to an improvement in performance.

GitHub

The project can be found on *GitHub* [27] at the following clickable *link*.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Convolutional networks for images, speech, and time series. *Handbook of Brain Theory and Neural Networks*, 3361:1995, 1995.

[2] Apple iphone 14. `https://www.apple.com/it/shop/buy-iphone/iphone-14/display-da-6,1%22-256gb-blu`. Accesso al sito: 6 luglio 2024.

[3] Wikipedia. Shell script (bash).

[4] Movavi. Mov vs mp4.

[5] Ffmpeg, 2024.

[6] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[7] G. Biau, G. Celeux, P. Craiuci, A. Glorieux, and O. Mestre. A survey of categorical encoding techniques for feature engineering in machine learning. *Statistics Surveys*, 11(1):1–43, 2018.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Softmax function: Definition, formula, applications, and examples. pages 227–232, 2016.

[9] Manuel Martinez and Rainer Stiefelhagen. Taming the cross entropy loss. *arXiv preprint arXiv:1810.05075*, 2018.

[10] A. E. Hoerl and R. W. Kennard. Ridge regression: An introduction. *Biometrika*, 56(1):1–14, 1970.

[11] Robert Tibshirani. Lasso: The least absolute shrinkage and selection operator. *Journal of the Royal Statistical Society. Series B*, 58(1):267–286, 1996.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Proceedings of the 14th International Conference on Learning Representations (ICML)*, 2011.

[14] David M Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2021.

[15] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[16] David M Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2021.

[17] San Zhang. *Loss Functions for Deep Learning: A Comprehensive Review*. Springer, 2021.

[18] Michael Martin. *Evaluating the Accuracy of Machine Learning Models*. O'Reilly Media, 2022.

[19] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. *Proceedings of the 23rd international conference on Machine learning*, pages 233–240, 2006.

[20] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. *Proceedings of the 23rd international conference on Machine learning*, pages 233–240, 2006.

[21] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[22] Python Software Foundation. Tkinter: Python interface to tcl/tk. `https://docs.python.org/3/library/tkinter.html`, 2021.

[23] Project Jupyter Contributors. Project jupyter. `https://jupyter.org/`, 2021.

[24] torch.nn.module ,Äî pytorch 1.10.0 documentation. Accessed: insert-date-here.

[25] Yingbin Bai, Erkun Yang, Bo Han, Yanhua Yang, Jiatong Li, Yinian Mao, Gang Niu, and Tongliang Liu. Understanding and improving early stopping for learning with noisy labels. *arXiv preprint arXiv:2106.15853*, 2021.

[26] Lutz Prechelt. *Early Stopping ,Ä̈î But When?* Springer, 2012.

[27] GitHub, Inc. GitHub. `https://github.com`. Accessed: 2024-07-02.