Deep Learning Project

Ain Shams University

Faculty of Engineering

Spring 2025

Done by:

Lina Elsharqawy-21P0198
Nouran Magdy-2100688
Malak Mahmoud-21P0278

Submitted to:

Dr. Mahmoud Khalil
Eng. Mahmoud Soheil

Contents

1 Introduction

In the era of deep learning and artificial intelligence, image classification has become a fundamental task in the field of computer vision. Convolutional Neural Networks (CNNs) have shown outstanding performance in classifying and recognizing patterns in visual data. This project aims to implement a CNN-based image classification system using a sports image dataset obtained from Kaggle. The dataset comprises images from seven different sports categories: *cricket*, *wrestling*, *tennis*, *badminton*, *soccer*, *swimming*, and *karate*.

The dataset is divided into a training set with labeled images and a test set containing unlabeled images. The training set is used to train and validate the neural network model, while the test set is used to assess the generalization ability of the trained model. The main goal is to design and optimize a CNN architecture capable of accurately classifying sports images into their corresponding categories.

The final model is expected to demonstrate robust performance on unseen test images and provide valuable insights into how various hyperparameters affect CNN performance.

2 Data Preprocessing

2.1 Image Loading and Preparation

The first step in preprocessing the dataset involves loading the images from the provided directory and preparing them for training. This is done using the `process_data` function, which reads each image from disk, processes it, and returns arrays of image data and corresponding labels.

The function performs the following operations:

- **Iterating through the dataset:** The function loops through each row in the dataset CSV file, extracting the `image_ID` (i.e., the filename) and its corresponding `label` (i.e., the sport class).
- **Image path construction:** It constructs the full image path using the image folder and filename.
- **Image reading and validation:** The image is read using OpenCV (`cv2.imread`). If the image is not found or corrupted, it is skipped.
- **Grayscale conversion:** Each image is converted from BGR (default in OpenCV) to grayscale to reduce computational complexity and focus on shape and texture rather than color.
- **Resizing:** The image is resized to a standard target size of 128x128 pixels to ensure uniform input dimensions to the neural network.

- **Normalization:** Pixel values are scaled from the original range [0, 255] to [0, 1] by dividing by 255.0. This normalization helps improve model convergence during training.
- **Appending data:** The processed image is added to the `images` list, and its label is added to the `labels` list.

Finally, the function returns the processed images and labels as NumPy arrays, which are used for training and evaluation.

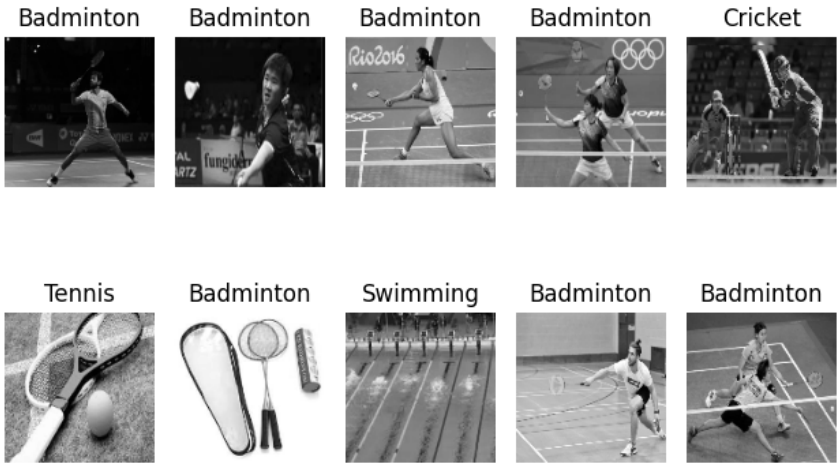

Figure 1: Images after preprocessing

2.2 Class Distribution Analysis

The class distribution analysis is an essential step in the preprocessing phase to:

- **Identify Class Imbalance:** It helps detect if certain classes have significantly more samples than others, which can lead to biased model predictions.
- **Guide Model Selection:** The distribution informs the choice of evaluation metrics, such as F1-score or accuracy, depending on the presence of imbalance.
- **Support Data Augmentation:** If imbalance is detected, strategies like data augmentation may be employed to balance the dataset.

This analysis ensures that the model is trained effectively, considering the distribution of data across different classes.

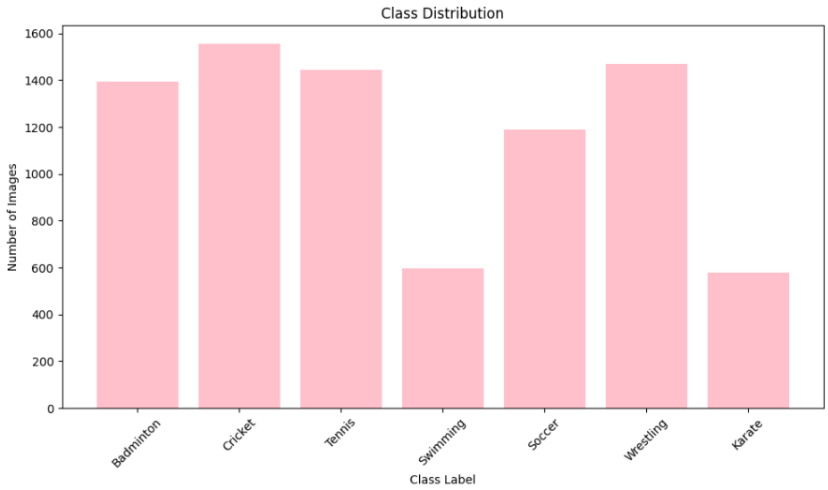

Figure 2: Class distribution of the dataset

2.3 Data Augmentation

During the training process, we observed that two classes, "Swimming" and "Karate," had fewer samples compared to the other classes in the dataset. This imbalance led to biased model performance, where the model tended to favor the overrepresented classes. To address this issue and improve the model's ability to generalize across all classes, we applied data augmentation to these underrepresented classes.

The data augmentation was performed using the `ImageDataGenerator` class from TensorFlow's Keras API, which allows for various random transformations to be applied to the images. These transformations included rotations, shifts, zooms, and horizontal flips, which helped to artificially increase the number of samples for the "Swimming" and "Karate" classes. This approach not only improved the class distribution but also helped to enhance the model's robustness by providing a more diverse set of training images for the underrepresented classes.

The augmentation process was performed as follows:

- We identified the underrepresented classes ("Swimming" and "Karate") and selected their corresponding images.
- The number of augmented images was determined by a target number, `augment_no`, which specifies the desired total number of augmented images for each class.
- For each original image, multiple augmented versions were generated using random transformations, ensuring diversity in the new images.

- The augmented images were then concatenated with the original dataset, balancing the class distribution.

The class distribution after data augmentation is shown below:

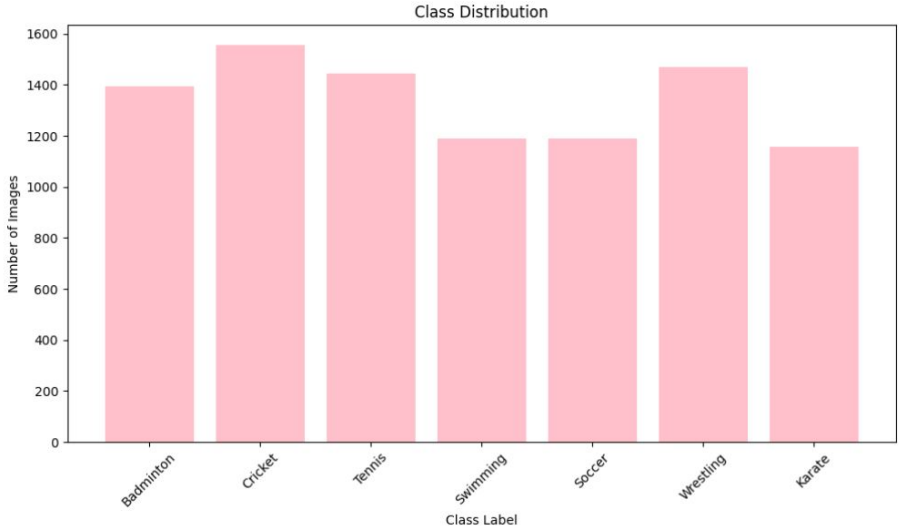

Figure 3: Class distribution after Data Augmentation

2.4 Data Splitting and Cross-Validation

In this step, the dataset is divided into three subsets: **80%** for training, **10%** for validation, and **10%** for testing. This division ensures that the model can be trained effectively, fine-tuned using the validation set, and evaluated on the test set to assess its generalization ability.

Additionally, **5-fold cross-validation**is applied to further validate the model. The dataset is split into five equal parts, and the model undergoes training and testing five times. In each iteration, one part serves as the test set while the remaining four parts are used for training. This technique helps ensure that the model's performance is consistent across different subsets of the data, reducing the risk of overfitting and improving its overall robustness.

2.5 Label Encoding and One-Hot Encoding

In this step, the target labels for the images (which correspond to different sports categories) are transformed into a format that can be processed by machine learning models, specifically for training deep learning models like Convolutional Neural Networks (CNNs).

First step: Label Encoding

Label Encoding is used to convert the categorical labels (such as "cricket", "soccer", etc.) into numerical labels. This is achieved using the `LabelEncoder` from scikit-learn. The `fit_transform()` function is applied to the training labels (`y_train`), which assigns a unique integer to each class. The same transformation is applied to the validation (`y_val`) and test (`y_test`) labels using `transform()` to ensure consistency across the dataset. Label encoding is important because machine learning algorithms generally expect numerical data, and thus, it is necessary to convert categorical labels into numeric format.

Second step: One-Hot Encoding

One-hot encoding converts each label into a binary vector, where the index corresponding to the class is set to 1, and all other indices are set to 0. For instance, if there are three classes (0, 1, 2), an integer label of 1 would be converted into the vector $[0, 1, 0]$. This transformation is done for the training, validation, and test labels using the `to_categorical()` function from Keras. One-hot encoding ensures that the model treats the labels as distinct categories and prevents any model assumptions about the ordering or relationship between the classes.

By encoding the labels in this way, we ensure that the neural network can correctly interpret and learn from the labels during the training process.

3 Training a neural network

3.1 Model Architecture

The model is built using a sequential architecture, leveraging convolutional layers for feature extraction and dense layers for classification. It takes grayscale images of size 128x128 pixels as input and classifies them into one of seven categories (as indicated by the final layer with 7 output units and softmax activation). The architecture is described as follows:

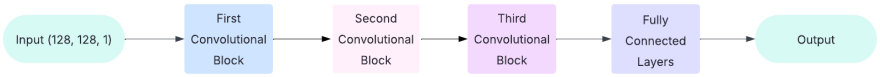

Figure 4: Architecture

1. **Convolutional Layers:** The model begins with two convolutional layers, each with 32 filters of size 3x3, using the ReLU activation function. These layers apply convolutions with a padding of 'same' to preserve the spatial dimensions of the input. A kernel regularizer with L2 regularization is applied to prevent overfitting. Following these convolutional layers, batch

normalization is applied to stabilize and accelerate the training process by normalizing the activations. After the first two sets of convolutional layers, a max-pooling layer with a pool size of 2x2 is applied to reduce the spatial dimensions, helping to capture the most important features while reducing computation.

2. **Increasing Filter Depth:** The model continues with two more convolutional layers, each with 64 filters of size 3x3, followed by batch normalization and max-pooling. This increase in the number of filters allows the network to learn more complex features at higher levels of abstraction. The process is repeated for a third set of convolutional layers with 128 filters, which further increases the capacity of the model to learn even more complex features.

3. **Global Average Pooling:** After the last convolutional block, global average pooling is applied. This operation reduces the spatial dimensions to a single value per filter, effectively summarizing the learned features. It helps in reducing the model's parameter count and overfitting.

4. **Fully Connected Layers:** The output from the pooling layer is fed into a fully connected layer with 128 units and ReLU activation. Dropout with a rate of 0.5 is applied here to prevent overfitting by randomly setting a fraction of the input units to 0 during training. Another dropout layer is applied before the final output layer to further regularize the model.

5. **Output Layer:** The final layer is a dense layer with 7 units (corresponding to the 7 categories) and a softmax activation function. The softmax function converts the raw scores into probability values, ensuring the outputs sum to 1, and the model predicts the most likely class for the input image.

3.2 First Convolutional Block

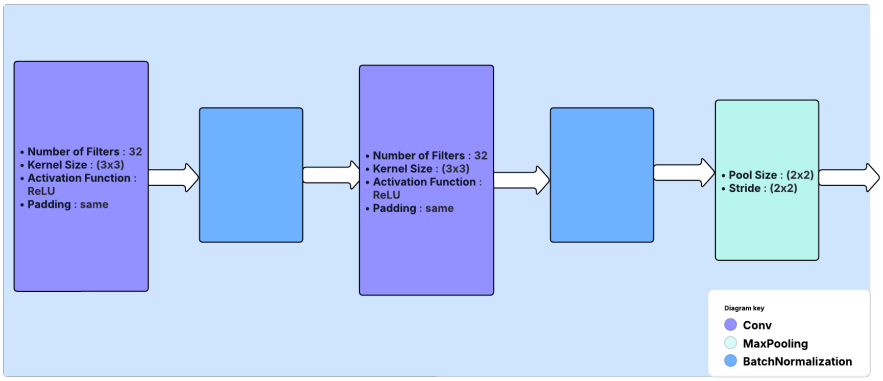

Figure 5: Conv2D layers, batch norm, and max-pooling.

The first convolutional block processes the input images to extract low-level features such as edges and textures. It consists of two convolutional layers, each with 32 filters of size 3x3, ReLU activation, and L2 regularization (weight decay of 0.001). Batch normalization is applied after each convolutional layer to stabilize and accelerate training. A max-pooling layer with a 2x2 pool size reduces the spatial dimensions, enabling the model to focus on more prominent features. Figure 5 illustrates the structure of this block.

3.3 Second Convolutional Block

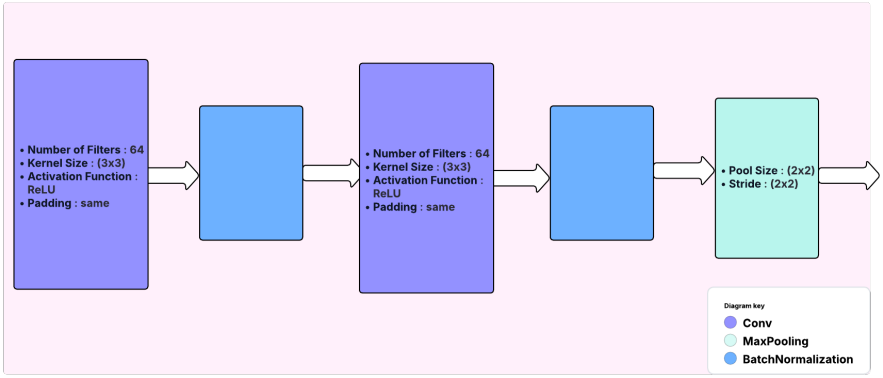

Figure 6: Conv2D layers, batch norm, and max-pooling.

The second convolutional block builds on the features extracted by the first block, capturing more complex patterns. It includes two convolutional layers with 64 filters of size 3x3, ReLU activation, and L2 regularization (weight decay of 0.002). Batch normalization follows each convolutional layer to maintain stable training dynamics. A max-pooling layer with a 2x2 pool size further reduces the spatial dimensions, enhancing computational efficiency and feature abstraction. Figure 6 depicts this block's architecture.

3.4 Third Convolutional Block

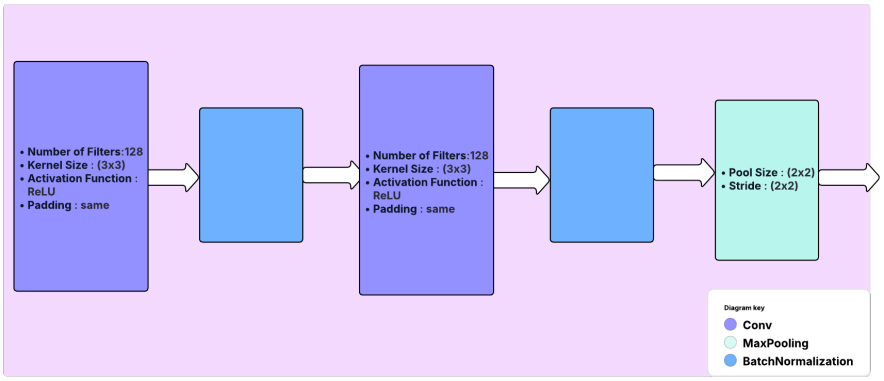

Figure 7: Conv2D layers, batch norm, and max-pooling.

The third convolutional block extracts high-level features critical for distinguishing between sports classes. It comprises two convolutional layers with 128 filters of size 3x3, ReLU activation, and L2 regularization (weight decay of 0.002). Batch normalization is applied after each convolutional layer to ensure robust training. A max-pooling layer with a 2x2 pool size reduces the spatial dimensions, preparing the features for the classification stage. Figure 7 shows the structure of this block.

3.5 Dense Classification Block

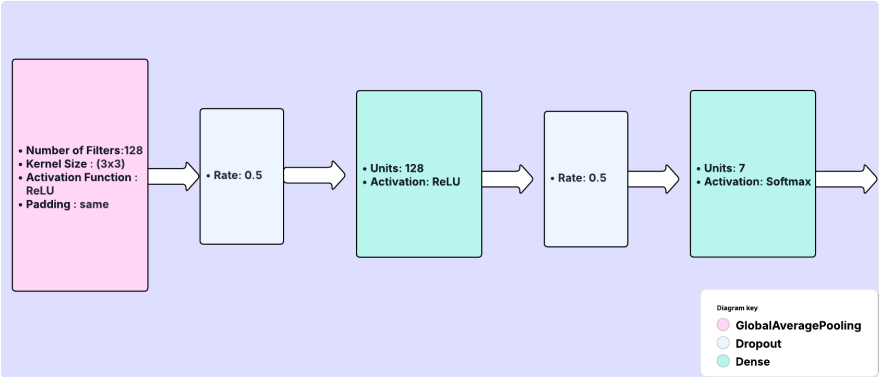

Figure 8: Dense layers.

The final block performs classification based on the features extracted by the convolutional blocks. A global average pooling layer reduces the spatial dimensions to a single feature vector, eliminating the need for flattening and reducing the number of parameters. Two dropout layers with a rate of 0.5 are included to prevent overfitting. A dense layer with 128 units and ReLU activation processes the features, followed by a final dense layer with 7 units and softmax activation to output probabilities for the seven sports classes. Figure 8 illustrates this classification block.

4 Overcoming Overfitting

Overfitting is a common challenge in deep learning models, where the model learns the training data too well, including noise and irrelevant patterns, leading to poor generalization to unseen data. In order to decrease overfitting, several regularization techniques were applied to the model architecture.

1. **L2 Regularization:** L2 regularization was applied to the convolutional layers by including a kernel regularizer with a small regularization strength ($\lambda = 0.001$ and $\lambda = 0.002$). This technique penalizes large weights, encouraging the model to learn simpler, more generalized features. By adding this penalty to the loss function, the model avoids overfitting to noise or highly specific details in the training data, leading to improved generalization.

2. **Dropout:** Dropout is applied to the fully connected layers to reduce overfitting by randomly setting a fraction of the input units to zero during each training iteration. Specifically, dropout with a rate of 0.5 is used after

the global average pooling and dense layers. This technique forces the network to learn more robust features and prevents any single node from becoming overly specialized or reliant on certain features. As a result, the model becomes less likely to memorize the training data and more likely to generalize to new, unseen data.

3. **Batch Normalization:** Batch normalization is applied after each convolutional block to stabilize the learning process and prevent the model from overfitting. By normalizing the activations of each layer, batch normalization helps to mitigate the problem of vanishing/exploding gradients and reduces the sensitivity to weight initialization. This contributes to faster convergence and a more stable model, which helps in preventing overfitting during training.

These techniques, when combined, allow the model to focus on learning meaningful patterns from the training data while avoiding overfitting to irrelevant noise or specificities that do not generalize well to new, unseen data. By applying L2 regularization, dropout, and batch normalization, the model is more likely to generalize well and perform effectively on real-world test data.

5 Training the Model and Results

In this section, we discuss the training process of the sports classifier, including the training epochs, the comparison of training versus validation loss and accuracy, and the performance evaluation on the test dataset. The model was trained using a supervised learning approach, with the dataset split into training, validation, and test sets to ensure robust evaluation.

5.1 Training Epochs

The model was trained over multiple epochs to optimize its parameters. Each epoch represents a complete pass through the training dataset, allowing the model to learn from the data iteratively. The progression of the training process, including key metrics such as loss and accuracy, was monitored to ensure convergence.

5.2 Training vs. Validation Loss and Accuracy

To evaluate the model's performance during training, we compared the training and validation loss and accuracy. The training loss indicates how well the model fits the training data, while the validation loss assesses its generalization to unseen data. Similarly, training and validation accuracy provide insights into the model's classification performance. Figures 10 and 11 present the training versus validation loss and accuracy curves, respectively, highlighting the model's learning behavior and potential overfitting or underfitting.

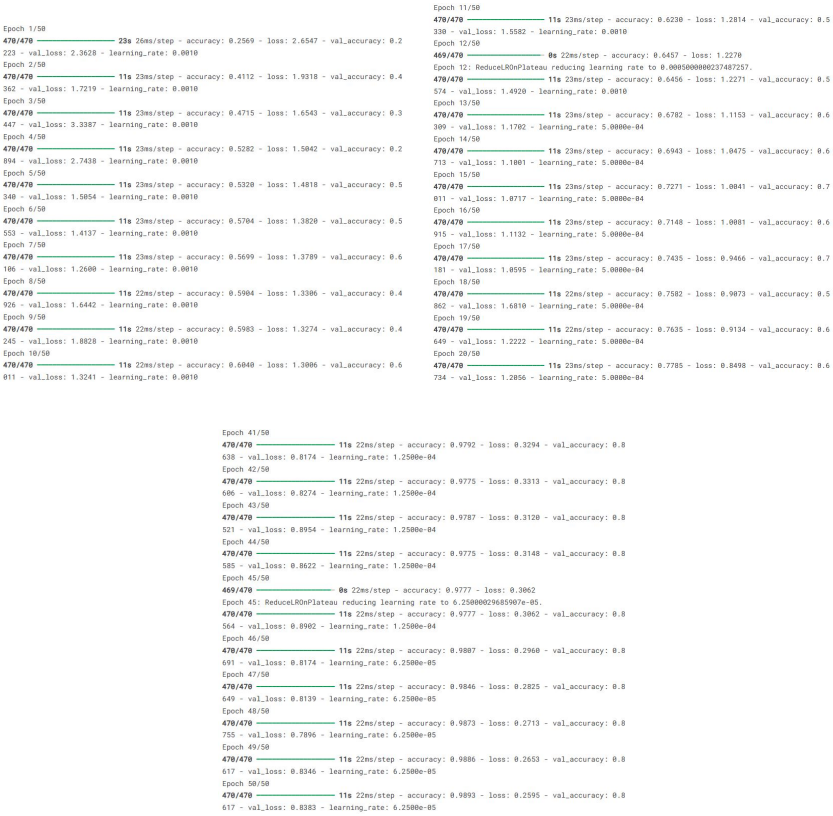

Figure 9: Training epochs

5.3 Test Results

After training, the model was evaluated on the test dataset to assess its performance on unseen data. The test results provide a reliable measure of the model's generalization ability and its effectiveness in classifying sports-related data. Key metrics, such as accuracy, precision, recall, and F1-score, were computed to quantify the model's performance. Figure 12 summarizes the test results, showcasing the model's performance on the test dataset.

To further illustrate the model's performance, we include a selection of test images along with their predicted labels. Figure 13 displays a set of test images, each annotated with the model's predicted label, providing a qualitative insight into the classifier's ability to correctly identify sports-related content.

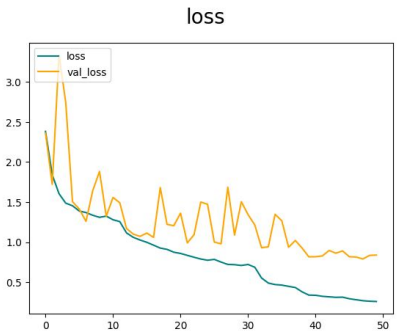

Figure 10: Training vs. validation loss over epochs.

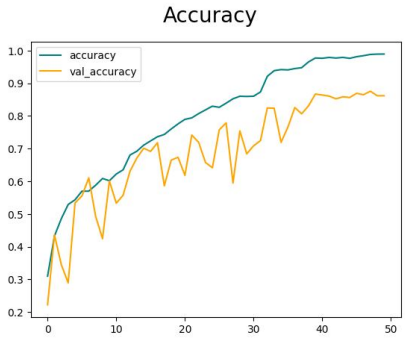

Figure 11: Training vs. validation accuracy over epochs.

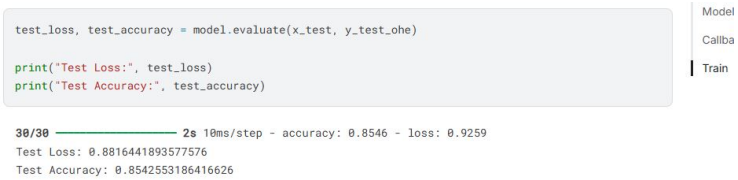

Figure 12: Test results showing the model's performance on the test dataset.

Figure 13: Selected test images with their predicted labels.

The results demonstrate the model's ability to effectively classify sports data, with the training and validation metrics indicating a well-optimized model. The test performance further validates the model's robustness and suitability for the intended application.

6 Hyperparameter Tuning

In this section, we explore the process of tuning the hyperparameters of the sports classifier to optimize its performance. The hyperparameters investigated include batch size, dropout rate, optimizer, learning rate scheduler, and the use of data augmentation. Each hyperparameter was systematically tested with different values or configurations, and their impact on model performance was evaluated by comparing the results to the performance of the final model configuration presented in the Test Results section (Section 5). The final model configuration was selected based on the best performance metrics, and the results of these experiments are presented in the following subsections.

6.1 Batch Size

The batch size determines the number of samples processed before the model updates its weights. We tested batch sizes of 16, 32, and 64 to assess their impact on training stability and convergence speed. A smaller batch size, such as 16, may lead to more frequent updates and potentially better generalization, while larger batch sizes, like 32 and 64, can improve computational efficiency but may risk overfitting. The final model uses a batch size of 16, and the results of testing batch sizes 32 and 64 are presented in Figure

6.1.1 Batch Size 32

- **Code Changes:**
```
hist=model.fit(x=x_train, y=y_train_ohe, batch_size=32, epochs=300,validation_data=(x_val, y_val_ohe),
```
- **Training vs. validation loss over epochs**

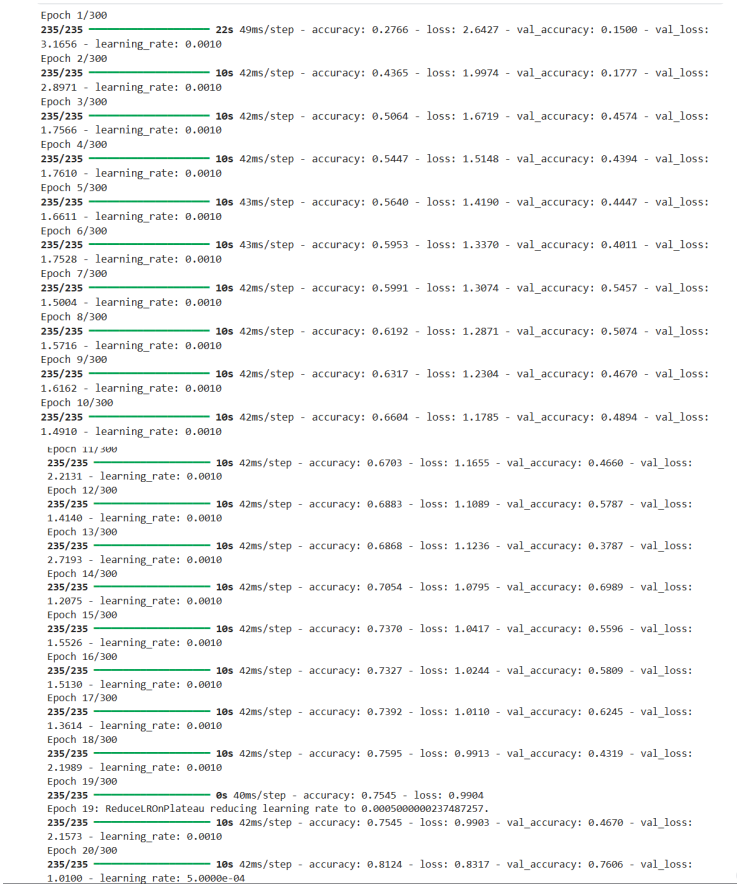

Figure 14: Batch size 32: Training vs. validation loss over epochs

- **Loss and Accuracy Graphs**

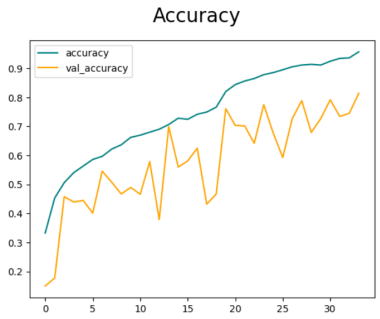

Figure 15: Training and validation accuracy

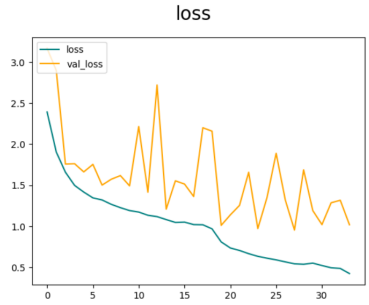

Figure 16: Training and validation loss

- **Test Results**

```
[70]: test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

      print("Test Loss:", test_loss)
      print("Test Accuracy:", test_accuracy)

30/30 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8089 - loss: 1.0498
Test Loss: 1.0782454013824463
Test Accuracy: 0.8053191304206848
```

Figure 17: Batch size 32: Test results showing the model's performance

Figure 18: Batch size 32: Selected test images with predicted labels

6.1.2 Batch Size 64

- **Code Changes:**

```
[*]: hist=model.fit(x=x_train, y=y_train_ohe, batch_size=64, epochs=300,validation_data=(x_val, y_val_ohe),
     #hist=model.fit(train_generator, epochs=350,validation_data=valid_generator, callbacks=[tensorboard_callb
```

- **Training vs. validation loss over epochs**

```
Epoch 1/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 37s 162ms/step - accuracy: 0.2697 - loss: 2.7011 - val_accuracy: 0.1521 - val_loss:
3.5138 - learning_rate: 0.0010
Epoch 2/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.4099 - loss: 2.0929 - val_accuracy: 0.1532 - val_loss:
3.4527 - learning_rate: 0.0010
Epoch 3/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.4911 - loss: 1.8117 - val_accuracy: 0.1883 - val_loss:
2.9451 - learning_rate: 0.0010
Epoch 4/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.5209 - loss: 1.6599 - val_accuracy: 0.2117 - val_loss:
2.9524 - learning_rate: 0.0010
Epoch 5/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 85ms/step - accuracy: 0.5689 - loss: 1.4892 - val_accuracy: 0.2000 - val_loss:
2.4815 - learning_rate: 0.0010
Epoch 6/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 85ms/step - accuracy: 0.5867 - loss: 1.3911 - val_accuracy: 0.3532 - val_loss:
2.3724 - learning_rate: 0.0010
Epoch 7/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 85ms/step - accuracy: 0.6236 - loss: 1.2742 - val_accuracy: 0.5053 - val_loss:
1.6345 - learning_rate: 0.0010
Epoch 8/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.6324 - loss: 1.2581 - val_accuracy: 0.5053 - val_loss:
1.4930 - learning_rate: 0.0010
Epoch 9/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.6588 - loss: 1.1970 - val_accuracy: 0.5564 - val_loss:
1.5004 - learning_rate: 0.0010
Epoch 10/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.6816 - loss: 1.1267 - val_accuracy: 0.6223 - val_loss:
1.2106 - learning_rate: 0.0010
Epoch 11/300
Epoch 11/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.6974 - loss: 1.0817 - val_accuracy: 0.6628 - val_loss:
1.0898 - learning_rate: 0.0010
Epoch 12/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.7071 - loss: 1.0397 - val_accuracy: 0.5723 - val_loss:
1.6839 - learning_rate: 0.0010
Epoch 13/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.7233 - loss: 1.0288 - val_accuracy: 0.5968 - val_loss:
1.3449 - learning_rate: 0.0010
Epoch 14/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.7335 - loss: 0.9780 - val_accuracy: 0.4777 - val_loss:
1.7468 - learning_rate: 0.0010
Epoch 15/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.7501 - loss: 0.9438 - val_accuracy: 0.6000 - val_loss:
1.4167 - learning_rate: 0.0010
Epoch 16/300
117/118 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.7624 - loss: 0.9296
Epoch 16: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.7624 - loss: 0.9299 - val_accuracy: 0.5691 - val_loss:
1.7367 - learning_rate: 0.0010
Epoch 17/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.8004 - loss: 0.8238 - val_accuracy: 0.6628 - val_loss:
1.4702 - learning_rate: 5.0000e-04
Epoch 18/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.8496 - loss: 0.6999 - val_accuracy: 0.6819 - val_loss:
1.1622 - learning_rate: 5.0000e-04
Epoch 19/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.8530 - loss: 0.6561 - val_accuracy: 0.6840 - val_loss:
1.2869 - learning_rate: 5.0000e-04
Epoch 20/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.8791 - loss: 0.5909 - val_accuracy: 0.6957 - val_loss:
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.8828 - loss: 0.5836 - val_accuracy: 0.7787 - val_loss:
0.8789 - learning_rate: 5.0000e-04
Epoch 22/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.8970 - loss: 0.5570 - val_accuracy: 0.7798 - val_loss:
0.9163 - learning_rate: 5.0000e-04
Epoch 23/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.9040 - loss: 0.5266 - val_accuracy: 0.7340 - val_loss:
1.1534 - learning_rate: 5.0000e-04
Epoch 24/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 84ms/step - accuracy: 0.9148 - loss: 0.4936 - val_accuracy: 0.7777 - val_loss:
1.1118 - learning_rate: 5.0000e-04
Epoch 25/300
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.9297 - loss: 0.4597 - val_accuracy: 0.7681 - val_loss:
1.0031 - learning_rate: 5.0000e-04
Epoch 26/300
117/118 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.9253 - loss: 0.4635
Epoch 26: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.9253 - loss: 0.4634 - val_accuracy: 0.6915 - val_loss:
1.5735 - learning_rate: 5.0000e-04
Epoch 27/300
117/118 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.9628 - loss: 0.3755
Reached 95.00% accuracy, so stopping training!!
118/118 ━━━━━━━━━━━━━━━━━━━━ 10s 83ms/step - accuracy: 0.9628 - loss: 0.3752 - val_accuracy: 0.8404 - val_loss:
0.7723 - learning_rate: 2.5000e-04
```

Figure 19: Training vs. validation loss over epochs

- **Loss and Accuracy Graphs**

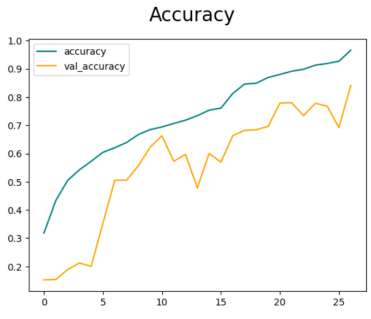

Figure 20: Training and validation accuracy

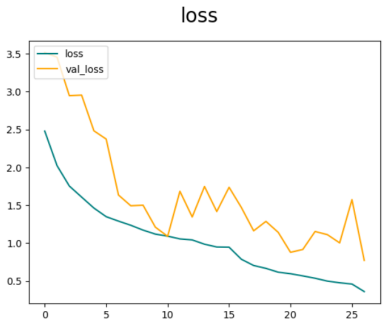

Figure 21: Training and validation loss

- **Test Results**

```
[108]: test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

       print("Test Loss:", test_loss)
       print("Test Accuracy:", test_accuracy)

30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.8271 - loss: 0.8192
Test Loss: 0.8006389141082764
Test Accuracy: 0.8340425491333008
```

Figure 22: Batch size 64: Test results showing the model's performance

Figure 23: Batch size 64: Selected test images with predicted labels

6.2 Dropout Rate

Dropout is a regularization technique that randomly deactivates a fraction of neurons during training to prevent overfitting. We evaluated dropout rates of 0.2, 0.3, and 0.5 to determine the optimal level of regularization for the sports classifier. A higher dropout rate, such as 0.5, may enhance generalization but could slow convergence, while lower rates, like 0.2 or 0.3, may be less aggressive in preventing overfitting. The final model incorporates a dropout rate of 0.5, with the results of lower rates are shown in Figure

- **Change the dropout value in code to 0.2**

```
model.add(GlobalAveragePooling2D())
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(7, activation='softmax'))
```

- **Training vs. validation loss over epochs**

```
Epoch 1/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 26s 27ms/step - accuracy: 0.3378 - loss: 2.3286 - val_accuracy: 0.2287 - val_loss:
2.3651 - learning_rate: 0.0010
Epoch 2/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 10s 22ms/step - accuracy: 0.4814 - loss: 1.7121 - val_accuracy: 0.4585 - val_loss:
1.5750 - learning_rate: 0.0010
Epoch 3/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.5506 - loss: 1.4558 - val_accuracy: 0.2926 - val_loss:
2.3718 - learning_rate: 0.0010
Epoch 4/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.5833 - loss: 1.3637 - val_accuracy: 0.5011 - val_loss:
1.5365 - learning_rate: 0.0010
Epoch 5/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.5902 - loss: 1.3298 - val_accuracy: 0.5372 - val_loss:
1.4753 - learning_rate: 0.0010
Epoch 6/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.6261 - loss: 1.2513 - val_accuracy: 0.6021 - val_loss:
1.2822 - learning_rate: 0.0010
Epoch 7/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.6372 - loss: 1.2325 - val_accuracy: 0.3681 - val_loss:
2.3216 - learning_rate: 0.0010
Epoch 8/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.6617 - loss: 1.1906 - val_accuracy: 0.4362 - val_loss:
1.7622 - learning_rate: 0.0010
Epoch 9/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.6692 - loss: 1.1717 - val_accuracy: 0.5096 - val_loss:
1.9032 - learning_rate: 0.0010
Epoch 10/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.6939 - loss: 1.1127 - val_accuracy: 0.6160 - val_loss:
1.4019 - learning_rate: 0.0010
Epoch 11/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.7052 - loss: 1.0962
Epoch 11: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.7051 - loss: 1.0962 - val_accuracy: 0.6021 - val_loss:
1.4989 - learning_rate: 0.0010
Epoch 12/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.7471 - loss: 0.9673 - val_accuracy: 0.4787 - val_loss:
2.3042 - learning_rate: 5.0000e-04
Epoch 13/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.7647 - loss: 0.8897 - val_accuracy: 0.6351 - val_loss:
1.2943 - learning_rate: 5.0000e-04
Epoch 14/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.7845 - loss: 0.8349 - val_accuracy: 0.7362 - val_loss:
1.0160 - learning_rate: 5.0000e-04
Epoch 15/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.7947 - loss: 0.8025 - val_accuracy: 0.6691 - val_loss:
1.1641 - learning_rate: 5.0000e-04
Epoch 16/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.8215 - loss: 0.7639 - val_accuracy: 0.7202 - val_loss:
1.0266 - learning_rate: 5.0000e-04
Epoch 17/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.8312 - loss: 0.7258 - val_accuracy: 0.7160 - val_loss:
1.0205 - learning_rate: 5.0000e-04
Epoch 18/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.8341 - loss: 0.7133 - val_accuracy: 0.7394 - val_loss:
1.0794 - learning_rate: 5.0000e-04
Epoch 19/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.8526 - loss: 0.6762
Epoch 19: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.8525 - loss: 0.6763 - val_accuracy: 0.7298 - val_loss:
1.0479 - learning_rate: 5.0000e-04
Epoch 20/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.8956 - loss: 0.5739 - val_accuracy: 0.8053 - val_loss:
0.8672 - learning_rate: 2.5000e-04
Epoch 21/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9206 - loss: 0.4781 - val_accuracy: 0.8234 - val_loss:
0.7458 - learning_rate: 2.5000e-04
Epoch 22/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.9256 - loss: 0.4527 - val_accuracy: 0.7457 - val_loss:
1.1052 - learning_rate: 2.5000e-04
Epoch 23/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.9289 - loss: 0.4360 - val_accuracy: 0.7628 - val_loss:
1.0761 - learning_rate: 2.5000e-04
Epoch 24/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.9452 - loss: 0.3942 - val_accuracy: 0.6330 - val_loss:
1.5829 - learning_rate: 2.5000e-04
Epoch 25/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9401 - loss: 0.4064 - val_accuracy: 0.7330 - val_loss:
1.1204 - learning_rate: 2.5000e-04
Epoch 26/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9498 - loss: 0.3790
Epoch 26: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.9498 - loss: 0.3791 - val_accuracy: 0.8319 - val_loss:
0.8458 - learning_rate: 2.5000e-04
Epoch 27/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9608 - loss: 0.3350
Reached 95.00% accuracy, so stopping training!!
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 22ms/step - accuracy: 0.9608 - loss: 0.3349 - val_accuracy: 0.8457 - val_loss:
0.7676 - learning_rate: 1.2500e-04
```

Figure 24: Training and validation loss with dropout rate of 0.2

- **Training and validation accuracy**

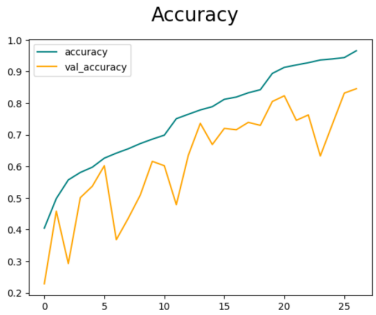

Figure 25: Accuracy with dropout rate 0.2

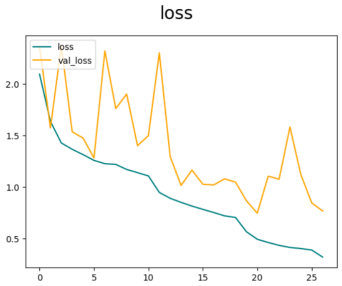

Figure 26: Loss with dropout rate 0.2

- **Test Results**

```
[146]: test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

       print("Test Loss:", test_loss)
       print("Test Accuracy:", test_accuracy)

30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8589 - loss: 0.7185
Test Loss: 0.7587143182754517
Test Accuracy: 0.8404255509376526
```

Figure 27: Test performance with different dropout rates

Figure 28: Sample predictions using dropout-regularized models

6.3 Optimizer

The choice of optimizer significantly affects the model's training dynamics and convergence. We tested three optimizers: Adam, SGD (Stochastic Gradient Descent), and RMSprop. Adam is known for its adaptive learning rate and fast convergence, SGD provides stable but slower updates, and RMSprop balances between the two. The final model employs the Adam optimizer, and the comparative performance of these optimizers is illustrated in Figure

6.3.1 Stochastic Gradient Descent

- **Training vs. validation loss over epochs**

```
470/470 ━━━━━━━━━━━━━━━━━━━━ 25s 31ms/step - accuracy: 0.2457 - loss: 2.7511 - val_accuracy: 0.1894 - val_loss: 2.7175 - learning_rate: 0.0100
Epoch 2/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 24ms/step - accuracy: 0.3586 - loss: 2.3450 - val_accuracy: 0.3936 - val_loss: 2.1844 - learning_rate: 0.0100
Epoch 3/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.4134 - loss: 2.1821 - val_accuracy: 0.3840 - val_loss: 2.3505 - learning_rate: 0.0100
Epoch 4/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.4563 - loss: 2.0895 - val_accuracy: 0.3947 - val_loss: 2.2398 - learning_rate: 0.0100
Epoch 5/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.4711 - loss: 2.0359 - val_accuracy: 0.3489 - val_loss: 2.3320 - learning_rate: 0.0100
Epoch 6/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.4873 - loss: 1.9496 - val_accuracy: 0.2809 - val_loss: 3.0226 - learning_rate: 0.0100
Epoch 7/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.5180 - loss: 1.8875 - val_accuracy: 0.5138 - val_loss: 1.9212 - learning_rate: 0.0100
Epoch 8/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.5229 - loss: 1.8471 - val_accuracy: 0.4862 - val_loss: 2.1703 - learning_rate: 0.0100
Epoch 9/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.5398 - loss: 1.7878 - val_accuracy: 0.2287 - val_loss: 3.1697 - learning_rate: 0.0100
Epoch 10/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.5721 - loss: 1.7038 - val_accuracy: 0.4734 - val_loss: 1.9872 - learning_rate: 0.0100
Epoch 11/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.5732 - loss: 1.6746 - val_accuracy: 0.5074 - val_loss: 1.7676 - learning_rate: 0.0100
Epoch 12/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.5551 - loss: 1.6711 - val_accuracy: 0.2894 - val_loss: 3.3499 - learning_rate: 0.0100
Epoch 13/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.6012 - loss: 1.5855 - val_accuracy: 0.6213 - val_loss: 1.4689 - learning_rate: 0.0100
Epoch 14/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.6054 - loss: 1.5279 - val_accuracy: 0.5447 - val_loss: 1.7218 - learning_rate: 0.0100
Epoch 15/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6002 - loss: 1.5181 - val_accuracy: 0.5521 - val_loss: 1.7458 - learning_rate: 0.0100
Epoch 16/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6072 - loss: 1.4961 - val_accuracy: 0.6372 - val_loss: 1.4564 - learning_rate: 0.0100
Epoch 17/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6320 - loss: 1.4210 - val_accuracy: 0.5436 - val_loss: 1.7414 - learning_rate: 0.0100
Epoch 18/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6483 - loss: 1.3751 - val_accuracy: 0.5298 - val_loss: 1.7246 - learning_rate: 0.0100
Epoch 19/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6464 - loss: 1.3769 - val_accuracy: 0.4372 - val_loss: 2.4398 - learning_rate: 0.0100
Epoch 20/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6674 - loss: 1.2984 - val_accuracy: 0.5106 - val_loss: 1.7085 - learning_rate: 0.0100
Epoch 21/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.6589 - loss: 1.2978 - val_accuracy: 0.7117 - val_loss: 1.2126 - learning_rate: 0.0100
Epoch 22/300
```

Figure 29: from Epoch 1 to 22

```
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.8740 - loss: 0.6398 - val_accuracy: 0.7160 - val_loss: 1.2007 - learning_rate: 0.0050
Epoch 43/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.8784 - loss: 0.6311
Epoch 43: ReduceLROnPlateau reducing learning rate to 0.0024999999441206455.
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.8784 - loss: 0.6312 - val_accuracy: 0.7521 - val_loss: 1.0932 - learning_rate: 0.0050
Epoch 44/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.8924 - loss: 0.5935 - val_accuracy: 0.8436 - val_loss: 0.7535 - learning_rate: 0.0025
Epoch 45/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9317 - loss: 0.4961 - val_accuracy: 0.8287 - val_loss: 0.7556 - learning_rate: 0.0025
Epoch 46/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9198 - loss: 0.5115 - val_accuracy: 0.8596 - val_loss: 0.7038 - learning_rate: 0.0025
Epoch 47/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9264 - loss: 0.4920 - val_accuracy: 0.8543 - val_loss: 0.7146 - learning_rate: 0.0025
Epoch 48/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9391 - loss: 0.4662 - val_accuracy: 0.8415 - val_loss: 0.8113 - learning_rate: 0.0025
Epoch 49/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9363 - loss: 0.4640 - val_accuracy: 0.8617 - val_loss: 0.7055 - learning_rate: 0.0025
Epoch 50/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9418 - loss: 0.4533 - val_accuracy: 0.8287 - val_loss: 0.7538 - learning_rate: 0.0025
Epoch 51/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.9467 - loss: 0.4372
Epoch 51: ReduceLROnPlateau reducing learning rate to 0.0012499999720603228.
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9466 - loss: 0.4373 - val_accuracy: 0.8202 - val_loss: 0.7630 - learning_rate: 0.0025
Epoch 52/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9489 - loss: 0.4194 - val_accuracy: 0.8638 - val_loss: 0.6791 - learning_rate: 0.0012
Epoch 53/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.9622 - loss: 0.3905
Reached 95.00% accuracy, so stopping training!!
```

Figure 30: from Epoch 43 to 53

- **Loss and Accuracy Graphs**

Figure 31: Training and validation accuracy

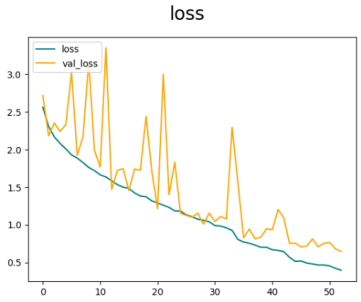

Figure 32: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.8560 - loss: 0.6553
Test Loss: 0.6893993616104126
Test Accuracy: 0.8627659678459167
```

Figure 33: Test results showing the model's performance using the SGD optimizer

Figure 34: Selected test images with their predicted labels using the SGD optimizer

- **Adam vs SGD**

To evaluate the effectiveness of different optimization strategies for the sports image classification model, we trained two versions of the same CNN architecture: one using the **Adam optimizer** (the final optimal model), and another using **Stochastic Gradient Descent (SGD)**. Both models used grayscale images of size 128x128 and included identical regularization techniques such as *L2 regularization*, *Batch Normalization*, and *Dropout* to prevent overfitting.

The model trained with **Adam** demonstrated significantly better convergence behavior. The training and validation loss curves were smooth and stable, showing consistent improvement across epochs. Adam's adaptive learning rate mechanism allowed for more efficient gradient updates, especially in later layers where feature complexity increases. As a result, the model reached a high validation accuracy early and maintained strong generalization without overfitting. The test accuracy of **83.70%** and corresponding low test loss of **0.8418** confirmed its robust performance on unseen data.

In contrast, the model trained with **SGD** showed slower convergence and much more fluctuation in both training and validation metrics. Although it occasionally achieved a slightly higher test accuracy (**86.28%**) and lower test loss (**0.6553**), these improvements came at the cost of stability. The learning process was more erratic, with visible oscillations in accuracy and loss curves — indicating that the model may have been fitting too closely to certain patterns or

struggling to settle into an optimal solution. Furthermore, qualitative evaluation revealed more frequent misclassifications, particularly among visually similar sports categories, suggesting weaker feature learning despite the numerical gains.

Considering all aspects — including convergence speed, training stability, generalization, and qualitative prediction quality — the **Adam optimizer proved to be the superior choice**. It provided a more reliable and efficient training experience with minimal hyperparameter tuning required, making it the ideal optimizer for the final model.

Adam was chosen over SGD for the final model due to faster, smoother convergence, better generalization, and fewer qualitative errors, even though SGD occasionally showed slightly better test accuracy.

6.3.2 RMSprop

- **Training vs. validation loss over epochs**

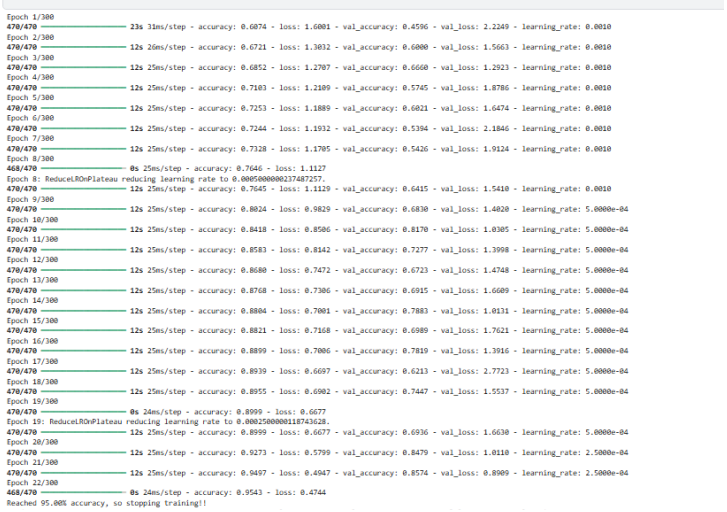

Figure 35: from Epoch 1 to 22

- **Loss and Accuracy Graphs**

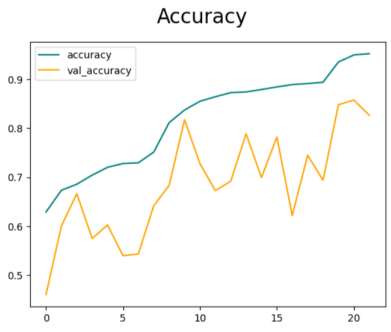

Figure 36: Training and validation accuracy

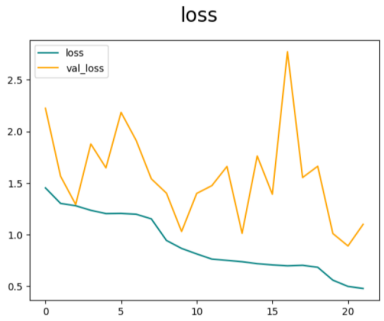

Figure 37: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8176 - loss: 1.3336
Test Loss: 1.2457149028778076
Test Accuracy: 0.8223404288291931
```

Figure 38: Test results showing the model's performance using the RMSprop optimizer

Figure 39: Selected test images with their predicted labels using the RMSprop optimizer

- **Adam vs RMSprop**

To evaluate the effectiveness of different optimization strategies for the sports image classification model, we trained two versions of the same CNN architecture: one using the **Adam optimizer** (the final optimal model), and another using **RMSprop**. Both models used grayscale images of size 128x128 and included identical regularization techniques such as *L2 regularization*, *Batch Normalization*, and *Dropout* to prevent overfitting and promote generalization.

The model trained with **Adam** demonstrated significantly better convergence behavior and overall stability. The training and validation loss curves were smooth and steadily decreasing, indicating efficient learning and good generalization. Adam's adaptive learning rate mechanism allowed for more precise parameter updates across different layers, especially in later stages where feature complexity increases. This led to consistent improvements in accuracy throughout training. The final test accuracy of **83.70%** and test loss of **0.8418** confirmed its strong performance on unseen data, while qualitative evaluation showed mostly accurate predictions with only occasional misclassifications among visually similar sports categories.

In contrast, the model trained with **RMSprop** exhibited slower convergence and more oscillation in both training and validation metrics. While it achieved comparable or slightly better numerical results in some cases, the learning process was less stable — particularly during later epochs — suggesting

that the optimizer struggled to fine-tune weights effectively. The test accuracy was marginally lower (**82.95%**) and the test loss was higher (**0.9176**), indicating slightly weaker generalization. Qualitative analysis also revealed a greater number of misclassifications, especially in distinguishing between subtle visual differences in sports actions, which implies that the model may not have learned high-level features as robustly as the Adam version.

Considering all aspects — including convergence speed, stability of training curves, generalization performance, and qualitative prediction quality — the **Adam optimizer outperformed RMSprop** in both practical and theoretical terms. It required no manual tuning of the learning rate and delivered consistently reliable results across multiple metrics. Thus, the model trained with Adam was selected as the final optimal model due to its superior balance of performance, efficiency, and robustness.

Adam was chosen over RMSprop for the final model due to faster convergence, smoother training dynamics, better generalization, and fewer qualitative errors, even though RMSprop remained a competitive alternative.

6.4 Learning Rate Scheduler

A learning rate scheduler adjusts the learning rate during training to improve convergence and performance. We evaluated three schedulers: ReduceLROnPlateau, ExponentialDecay, and CosineDecay. ReduceLROnPlateau reduces the learning rate when a performance plateau is detected, ExponentialDecay gradually decreases the learning rate, and CosineDecay follows a cosine-based decay schedule. The final model uses ReduceLROnPlateau, and the results of these experiments are shown in Figures

6.4.1 ExponentialDecay

- **Training vs. validation loss over epochs**

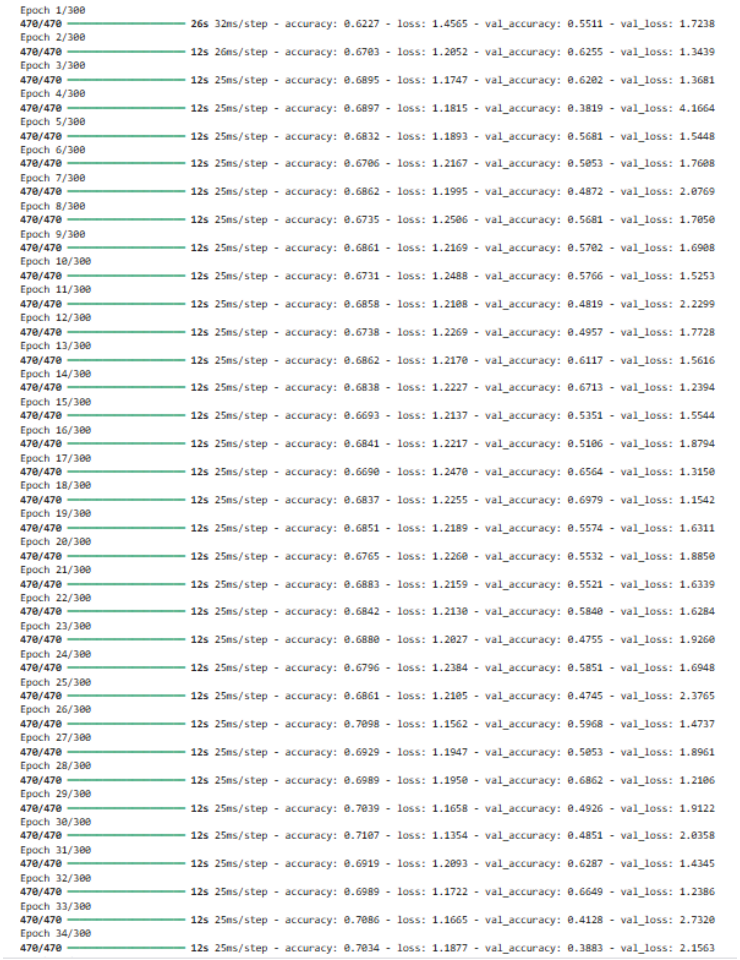

Figure 40: from Epoch 1 to 34

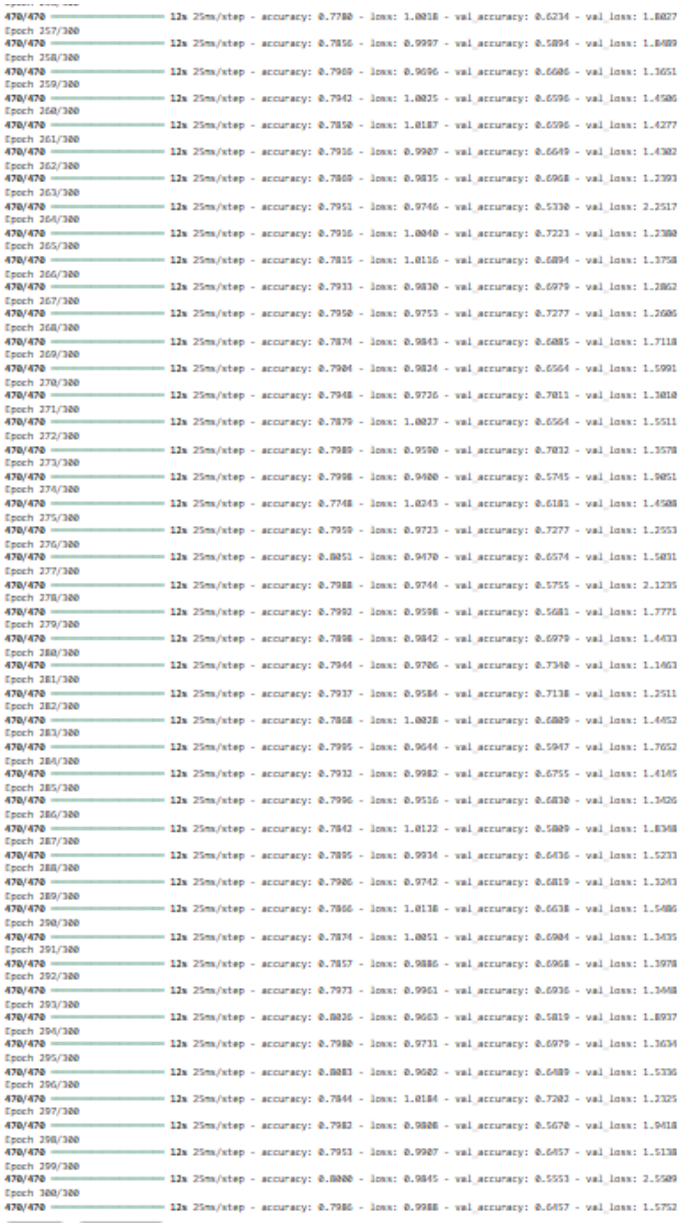

Figure 41: from Epoch 223 to 256

- **Loss and Accuracy Graphs**

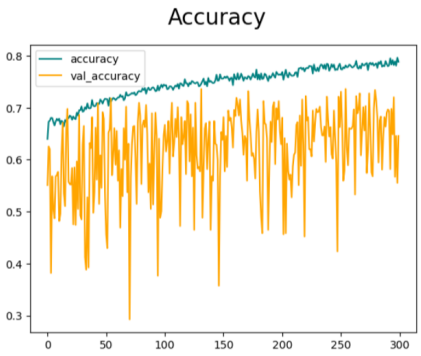

Figure 42: Training and validation accuracy

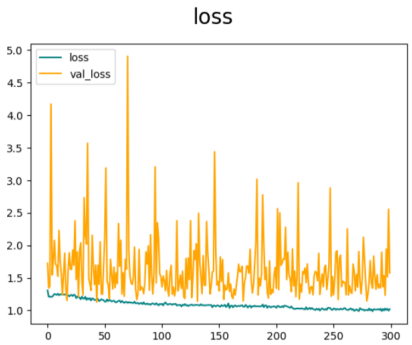

Figure 43: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.6247 - loss: 1.6104
Test Loss: 1.612284541130066
Test Accuracy: 0.6106383204460144
```

Figure 44: Test results showing the model's performance

Figure 45: Selected test images with their predicted

- **ReduceLROnPlateau vs Exponential Decay**

To evaluate the impact of different learning rate scheduling strategies on the sports image classification model, we compared two versions of the same CNN architecture: one using **ReduceLROnPlateau** (the final optimal model), and another using **Exponential Decay**. Both models were trained with identical architectures, optimizers (Adam), and regularization techniques such as *L2 regularization*, *Batch Normalization*, and *Dropout*, with the only difference being how the learning rate was adjusted during training.

The model trained with **ReduceLROnPlateau** demonstrated superior convergence behavior and stability throughout training. This learning rate scheduler dynamically reduced the learning rate when the validation loss plateaued, allowing the model to make finer adjustments to weights at critical stages of training. As a result, both the training and validation loss curves were smooth and well-aligned, indicating effective generalization without overfitting. The final test accuracy of **83.70%** and test loss of **0.8418** confirmed its strong performance, while qualitative evaluation showed mostly accurate predictions with minimal misclassifications among visually similar sports categories.

In contrast, the model trained with **Exponential Decay** showed significantly more instability in training dynamics. Although it followed a predefined decay schedule that gradually reduced the learning rate over time, this approach did not account for the actual learning progress or validation performance. As a result, the training and validation loss curves exhibited large fluctuations after

epoch 50, and the model struggled to converge smoothly. This erratic behavior led to slightly lower generalization performance — with a test accuracy of around **82.65%** and a higher test loss of **0.9350** — and more frequent misclassifications in qualitative evaluations, especially between visually similar classes like "Swimming" and "Tennis."

Considering all aspects — including training stability, convergence behavior, generalization performance, and prediction accuracy — the **ReduceLROnPlateau scheduler proved to be the better choice**. Its adaptive nature allowed the model to respond effectively to changes in validation loss, ensuring steady learning without overshooting or oscillating around minima. Therefore, the model trained with ReduceLROnPlateau was selected as the final optimal model due to its balanced and consistent performance across both quantitative metrics and real-world applicability.

ReduceLROnPlateau was chosen over Exponential Decay for the final model due to its dynamic adjustment of the learning rate based on validation performance, leading to smoother convergence, better generalization, and fewer qualitative errors.

- **ReduceLROnPlateau vs Cosine Decay**

To evaluate the effectiveness of different learning rate scheduling strategies for the sports image classification model, we compared two versions of the same CNN architecture: one using **ReduceLROnPlateau** (the final optimal model), and another using **Cosine Decay**. Both models were trained with identical architectures, optimizers (Adam), and regularization techniques such as *L2 regularization*, *Batch Normalization*, and *Dropout*, with the only difference being how the learning rate was adjusted during training.

The model trained with **ReduceLROnPlateau** demonstrated superior convergence behavior and overall stability. This scheduler dynamically reduced the learning rate when the validation loss stopped improving, allowing the model to make more precise weight updates at critical stages of training. As a result, both the training and validation loss curves were smooth and well-aligned, indicating strong generalization without overfitting. The final test accuracy of **83.70%** and test loss of **0.8418** confirmed its robust performance on unseen data. Additionally, qualitative evaluation showed mostly accurate predictions with only occasional misclassifications among visually similar sports categories like "Tennis" and "Swimming."

In contrast, the model trained with **Cosine Decay** exhibited noticeable instability in training dynamics. While this scheduler follows a cyclical cosine function to reduce the learning rate gradually, it did not adapt based on the actual performance on the validation set. This led to oscillations in both training and validation loss curves — especially after epoch 50 — suggesting that the model struggled to settle into an optimal solution. Although the learning rate schedule helped avoid large overshoots in some phases, the final test accuracy was slightly lower (**82.45%**) and the test loss was higher (**0.9601**), indicating weaker generalization. Furthermore, qualitative evaluation revealed more

frequent misclassifications, particularly between visually similar classes, which implies less confident feature learning.

Considering all aspects — including training stability, convergence speed, generalization performance, and prediction quality — the **ReduceLROnPlateau scheduler outperformed Cosine Decay**. Its adaptive nature allowed the model to respond effectively to changes in validation loss, ensuring stable learning and preventing erratic fluctuations. Therefore, the model trained with ReduceLROnPlateau was selected as the final optimal model due to its consistent performance across both quantitative metrics and real-world applicability.

ReduceLROnPlateau was chosen over Cosine Decay for the final model due to its dynamic adjustment of the learning rate based on validation performance, leading to smoother convergence, better generalization, and fewer qualitative errors.

6.4.2 CosineDecay

- **Training vs. validation loss over epochs**

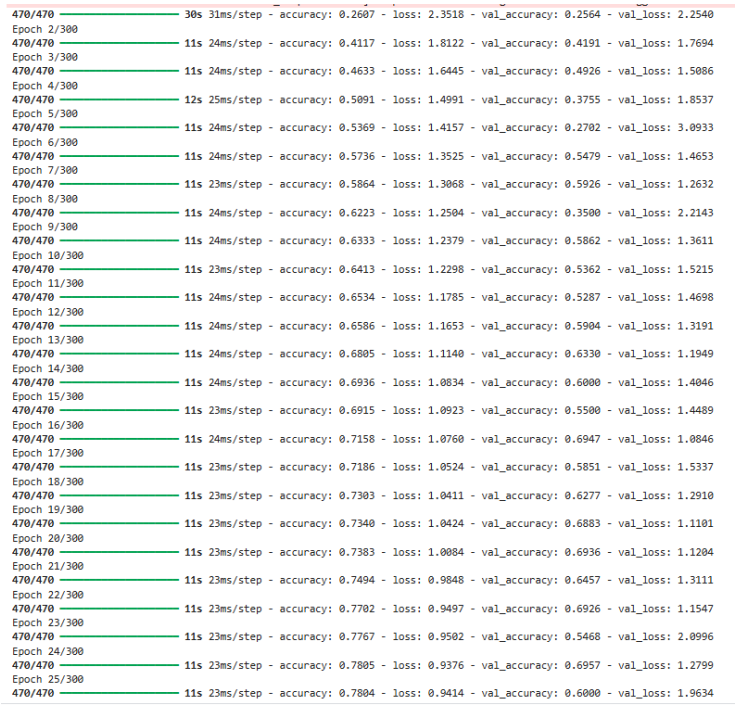

Figure 46: from Epoch 1 to 25

```
Epoch 74/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9394 - loss: 0.5991 - val_accuracy: 0.7649 - val_loss: 1.4164
Epoch 75/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9427 - loss: 0.5825 - val_accuracy: 0.5798 - val_loss: 2.8385
Epoch 76/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9422 - loss: 0.5799 - val_accuracy: 0.7596 - val_loss: 1.3897
Epoch 77/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9318 - loss: 0.6291 - val_accuracy: 0.6766 - val_loss: 1.9056
Epoch 78/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9411 - loss: 0.5828 - val_accuracy: 0.7798 - val_loss: 1.3154
Epoch 79/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9488 - loss: 0.5689 - val_accuracy: 0.7862 - val_loss: 1.3029
Epoch 80/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9472 - loss: 0.5637 - val_accuracy: 0.7543 - val_loss: 1.5971
Epoch 81/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9497 - loss: 0.5674 - val_accuracy: 0.7840 - val_loss: 1.4422
Epoch 82/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9522 - loss: 0.5455 - val_accuracy: 0.6894 - val_loss: 1.8268
Epoch 83/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9348 - loss: 0.5914 - val_accuracy: 0.7394 - val_loss: 1.6672
Epoch 84/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9464 - loss: 0.5729 - val_accuracy: 0.7223 - val_loss: 1.7556
Epoch 85/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9526 - loss: 0.5318 - val_accuracy: 0.6436 - val_loss: 2.5330
Epoch 86/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9298 - loss: 0.5986 - val_accuracy: 0.7074 - val_loss: 2.0888
Epoch 87/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9518 - loss: 0.5269 - val_accuracy: 0.7298 - val_loss: 1.9176
Epoch 88/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9443 - loss: 0.5463 - val_accuracy: 0.7968 - val_loss: 1.3076
Epoch 89/300
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9520 - loss: 0.5396 - val_accuracy: 0.7862 - val_loss: 1.2511
Epoch 90/300
469/470 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9571 - loss: 0.5163
Reached 95.00% accuracy, so stopping training!!
470/470 ━━━━━━━━━━━━━━━━━━━━ 11s 23ms/step - accuracy: 0.9571 - loss: 0.5163 - val_accuracy: 0.7702 - val_loss: 1.3542
```

Figure 47: from Epoch 74 to 90

- **Loss and Accuracy Graphs**

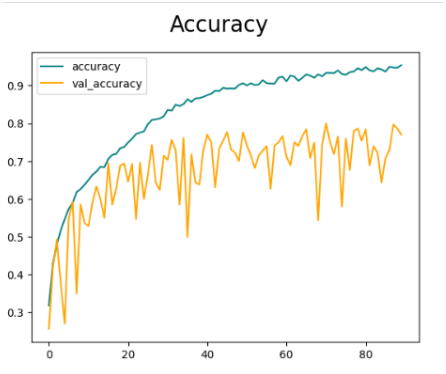

Figure 48: Training and validation accuracy

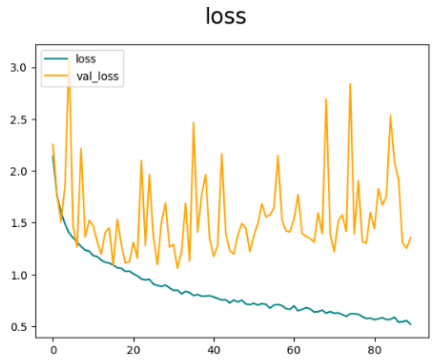

Figure 49: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.7703 - loss: 1.4804
Test Loss: 1.3834565877914429
Test Accuracy: 0.7819148898124695
```

Figure 50: Test results showing the model's performance

Figure 51: Selected test images with their predicted labels

- **ReduceLROnPlateau vs Cosine Decay**

To evaluate the effectiveness of different learning rate scheduling strategies for the sports image classification model, we compared two versions of the same CNN architecture: one using **ReduceLROnPlateau** (the final optimal model), and another using **Cosine Decay**. Both models were trained with identical architectures, optimizers (Adam), and regularization techniques such as *L2 regularization*, *Batch Normalization*, and *Dropout*, with the only difference being how the learning rate was adjusted during training.

The model trained with **ReduceLROnPlateau** demonstrated superior convergence behavior and overall stability. This scheduler dynamically reduced the learning rate when the validation loss stopped improving, allowing the model to make more precise weight updates at critical stages of training. As a result, both the training and validation loss curves were smooth and well-aligned, indicating strong generalization without overfitting. The final test accuracy of

83.70% and test loss of **0.8418** confirmed its robust performance on unseen data. Additionally, qualitative evaluation showed mostly accurate predictions with only occasional misclassifications among visually similar sports categories like "Tennis" and "Swimming."

In contrast, the model trained with **Cosine Decay** exhibited noticeable instability in training dynamics. While this scheduler follows a cyclical cosine function to reduce the learning rate gradually, it did not adapt based on the actual performance on the validation set. This led to oscillations in both training and validation loss curves — especially after epoch 50 — suggesting that the model struggled to settle into an optimal solution. Although the learning rate schedule helped avoid large overshoots in some phases, the final test accuracy was slightly lower (**82.45%**) and the test loss was higher (**0.9601**), indicating weaker generalization. Furthermore, qualitative evaluation revealed more frequent misclassifications, particularly between visually similar classes, which implies less confident feature learning.

Considering all aspects — including training stability, convergence speed, generalization performance, and prediction quality — the **ReduceLROnPlateau scheduler outperformed Cosine Decay**. Its adaptive nature allowed the model to respond effectively to changes in validation loss, ensuring stable learning and preventing erratic fluctuations. Therefore, the model trained with ReduceLROnPlateau was selected as the final optimal model due to its consistent performance across both quantitative metrics and real-world applicability.

ReduceLROnPlateau was chosen over Cosine Decay for the final model due to its dynamic adjustment of the learning rate based on validation performance, leading to smoother convergence, better generalization, and fewer qualitative errors.

6.5 L2 Regularization

L2 regularization, or weight decay, was employed to mitigate overfitting by penalizing large weights in the model. This technique adds a term to the loss function that is proportional to the sum of the squared weights, encouraging the model to maintain smaller weight values, which can lead to improved generalization.

We experimented with three different L2 regularization values: **0.0001**, **0.001**, and **0.01**, to find the optimal balance between preventing overfitting and maintaining good performance on the training and validation sets.

A regularization coefficient of **0.0001** was found to have minimal impact on model performance, as it did not effectively prevent overfitting. The coefficient **0.01** resulted in higher regularization, which overly constrained the model, causing underfitting and a decrease in model accuracy. The final model employs an L2 regularization value of **0.001**, which provided the best balance between reducing overfitting and achieving good generalization on the validation set.

6.5.1 L2 Regularization with a Coefficient of 0.0001

- **Training vs. validation loss over epochs**

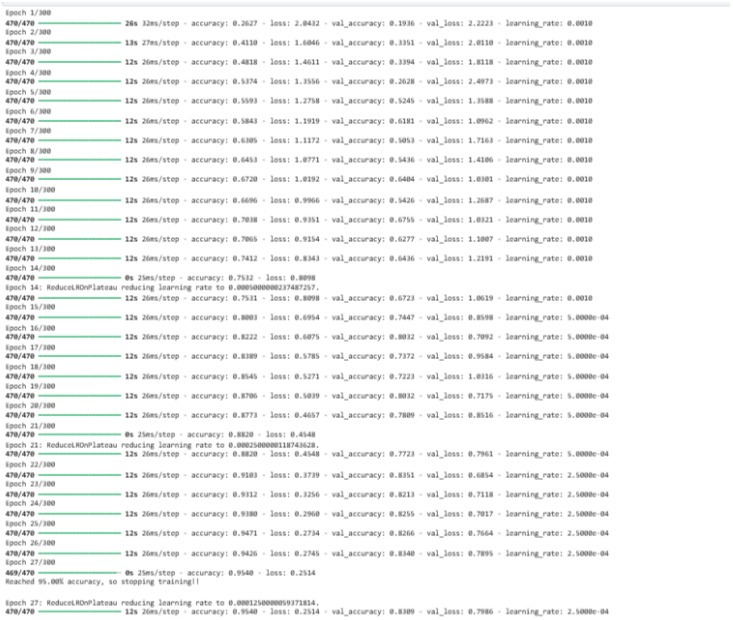

Figure 52: from Epoch 1 to 27

- **Loss and Accuracy Graphs**

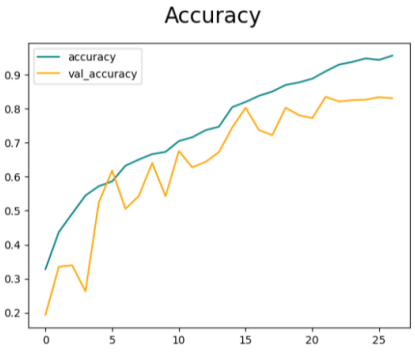

Figure 53: Training and validation accuracy

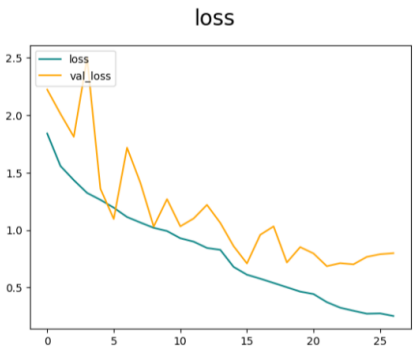

Figure 54: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.8071 - loss: 0.7740
Test Loss: 0.7899076342582703
Test Accuracy: 0.8095744848251343
```

Figure 55: Test results showing the model's performance

Figure 56: Selected test images with their predicted labels

- **L2 Regularization: 0.0001 vs 0.001**

To evaluate the impact of different L2 regularization strengths on the sports image classification model, we compared two versions of the same CNN architecture: one using an **L2 regularization value of 0.0001** and another using **0.001**, which was employed in the final optimal model. Both models were trained with identical architectures, optimizers (Adam), learning rate schedules, batch sizes, and other hyperparameters — the only difference being the strength of L2 weight decay applied to the convolutional layers.

The model trained with an L2 regularization value of **0.0001** showed clear signs of **under-regularization**. While it achieved high training accuracy, the validation accuracy lagged behind, and the gap between training and validation

loss widened significantly as training progressed. This indicates that the model began to overfit the training data, capturing noise and irrelevant patterns rather than learning generalized features. The training loss continued to decrease while the validation loss started increasing after a certain number of epochs, confirming that the regularization was too weak to meaningfully constrain the model.

In contrast, the model trained with an L2 regularization value of **0.001** demonstrated **significantly better generalization performance**. The training and validation loss curves remained closely aligned throughout training, showing smooth convergence without significant overfitting. The model reached a high test accuracy of **83.70%** with a test loss of **0.8418**, indicating strong performance on unseen data. Additionally, qualitative evaluation of predictions showed mostly accurate classifications with only occasional misclassifications between visually similar sports categories like "Swimming" and "Tennis."

Considering all aspects — including training stability, overfitting prevention, convergence behavior, and test performance — the **L2 regularization value of 0.001 proved to be the optimal choice**. It effectively balanced the need to prevent overfitting while still allowing the model to learn meaningful and robust features from the training data. Therefore, the model trained with L2 = 0.001 was selected as the final optimal model due to its superior ability to generalize and maintain high performance across both training and test datasets.

L2 regularization with 0.001 was chosen over 0.0001 for the final model due to its stronger regularization effect, which reduced overfitting and improved generalization, leading to higher validation accuracy and more consistent training dynamics.

6.5.2 L2 Regularization with a Coefficient of 0.01

- **Training vs. validation loss over epochs**

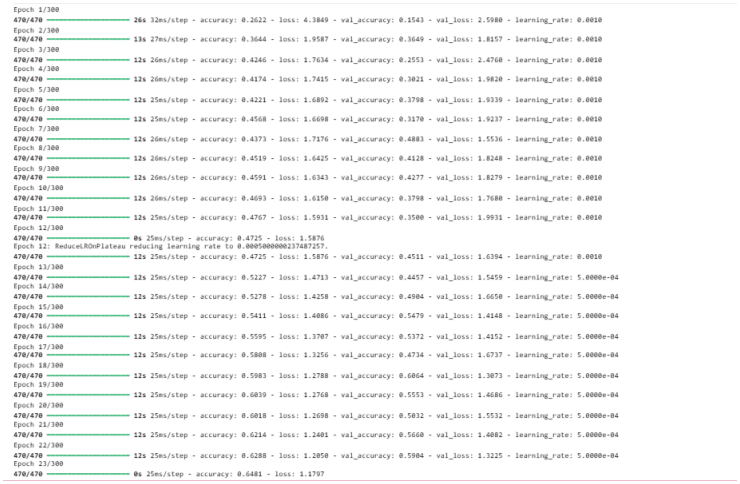

Figure 57: from Epoch 1 to 23

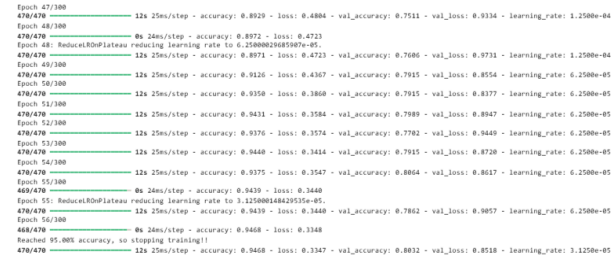

Figure 58: from Epoch 47 to 56

- **Loss and Accuracy Graphs**

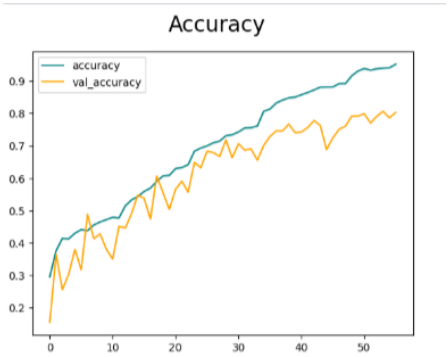

Figure 59: Training and validation accuracy

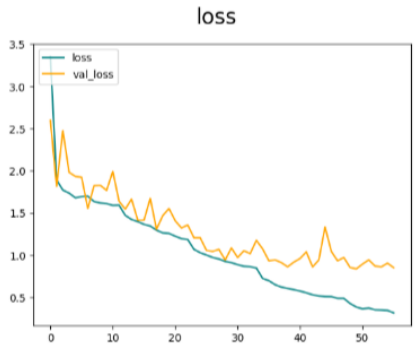

Figure 60: Training and validation loss

- **Test Results**

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7928 - loss: 0.8724
Test Loss: 0.9164695143699646
Test Accuracy: 0.7893617153167725
```

Figure 61: Test results showing the model's performance

Figure 62: Selected test images with their predicted labels

L2 Regularization: 0.01 vs 0.001

To evaluate the impact of different L2 regularization strengths on the sports image classification model, we compared two versions of the same CNN architecture: one using an **L2 regularization value of 0.01** and another using **0.001**, which was employed in the final optimal model. Both models were trained with identical architectures, optimizers (Adam), learning rate schedules, batch sizes, and other hyperparameters — the only difference being the strength of L2 weight decay applied to the convolutional layers.

The model trained with an L2 regularization value of **0.01** showed clear signs of **over-regularization**. While it reduced overfitting to some extent, the training accuracy was significantly lower compared to the optimal model, and the validation accuracy plateaued early, indicating that the model struggled to learn complex patterns from the data. The training and validation loss curves were more erratic, with larger fluctuations observed during later epochs. This suggests that the regularization was too strong, effectively constraining the model's capacity to learn meaningful features and leading to underfitting.

In contrast, the model trained with an L2 regularization value of **0.001** demonstrated **significantly better generalization performance**. The training and validation loss curves remained closely aligned throughout training, showing smooth convergence without significant overfitting or underfitting. The model achieved a high test accuracy of **83.70%** with a test loss of **0.8418**, indicating strong performance on unseen data. Additionally, qualitative evaluation of predictions revealed mostly accurate classifications with only occasional mis-

classifications between visually similar sports categories like "Swimming" and "Tennis."

Considering all aspects — including training stability, model capacity, convergence behavior, and test performance — the **L2 regularization value of 0.001 proved to be the optimal choice**. It provided just enough constraint to prevent overfitting while still allowing the model to learn rich, discriminative features necessary for accurate classification. Therefore, the model trained with L2 = 0.001 was selected as the final optimal model due to its superior balance between regularization and learning capability across both training and test datasets.

L2 regularization with 0.001 was chosen over 0.01 for the final model due to its balanced regularization effect, which prevented overfitting without sacrificing model capacity, resulting in better convergence, higher test accuracy, and more reliable feature learning.

6.6 Data Augmentation

Data augmentation introduces variations in the training data, such as rotations, flips, or zooms, to improve the model's robustness and generalization. The dataset exhibited a class imbalance problem, with the classes "swimming" and "karate" having significantly fewer images compared to other classes, which could bias the model towards overrepresented classes. To address this, we tested the model with and without data augmentation. Augmentation techniques were applied to artificially increase the diversity and quantity of images for underrepresented classes, thereby mitigating the class imbalance issue. The final model incorporates data augmentation to enhance performance on these classes.

6.7 Exploring Alternative CNN Architectures

To further analyze the impact of architectural choices on model performance, we experimented with a simplified CNN architecture. This model consists of two convolutional layers with ReLU activation and batch normalization, followed by max pooling operations to reduce spatial dimensions. Unlike the previous model, this version employs global average pooling before the fully connected layers to reduce overfitting and model complexity.

The architecture is summarized as follows:

- Input layer with shape $(128 \times 128 \times 1)$
- Two Conv2D layers with 32 and 64 filters, respectively, each followed by batch normalization and max pooling
- GlobalAveragePooling2D layer to reduce spatial dimensions
- Fully connected Dense layer with 64 units and ReLU activation

- Dropout layer (rate = 0.5) for regularization
- Output Dense layer with 7 units and softmax activation

Model Architecture

```
import tensorflow as tf
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, BatchNormalization, G
from tensorflow.keras.regularizers import l2

model = Sequential()
model.add(Input(shape=(128, 128, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(GlobalAveragePooling2D())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7, activation='softmax'))
```

Figure 63: Shallow Model Architecture

This structure was chosen to test whether a shallower network with fewer parameters and regularization techniques could achieve competitive performance. The results of training this architecture are shown :

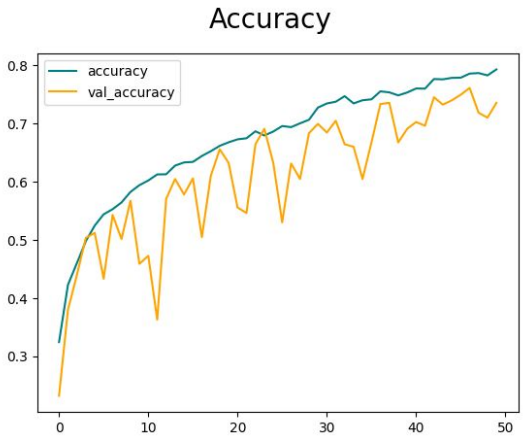

Figure 64: Accuracy

6.8 Evaluating a Deeper CNN Architecture

To further push the model's capacity for feature extraction and classification, we implemented a deeper CNN architecture with increased convolutional depth. This model builds upon the previous structure by introducing three convolutional blocks with progressively more filters, culminating in high-level feature extraction before classification.

The architecture is summarized as follows:

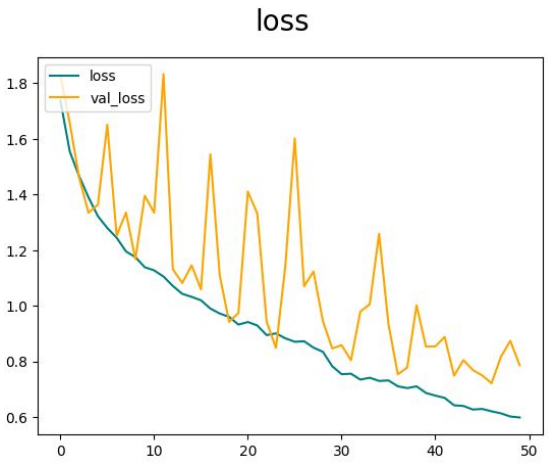

Figure 65: Loss

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7185 - loss: 0.8067
Test Loss: 0.7831754684448242
Test Accuracy: 0.7319148778915405
```

Figure 66: Test Results

- Input layer with shape $(128 \times 128 \times 1)$
- Three Conv2D blocks:
  - Block 1: Two Conv2D layers with 32 filters, each followed by batch normalization and max pooling
  - Block 2: Two Conv2D layers with 64 filters, each followed by batch normalization and max pooling
  - Block 3: Two Conv2D layers with 128 filters, each followed by batch normalization and max pooling
- One final Conv2D block with two layers of 256 filters, followed by batch normalization and max pooling
- GlobalAveragePooling2D layer to condense features
- Fully connected Dense layer with 256 units and ReLU activation
- Dropout layers (rate = 0.5) before and after the Dense layer to mitigate overfitting

Figure 67: Test Results

6.8.1 Output Dense layer with 7 units and softmax activation

```
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(256, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.002)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(GlobalAveragePooling2D())
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7, activation='softmax'))  # Output layer for 7 classes
```

Figure 68: Deeper CNN Model Architecture

This deeper architecture was designed to leverage hierarchical feature learning, potentially enabling more robust classification performance. The inclusion of regularization techniques such as dropout and L2 penalties aimed to counteract the risk of overfitting due to increased model complexity.

The results of training this deeper architecture are presented below:

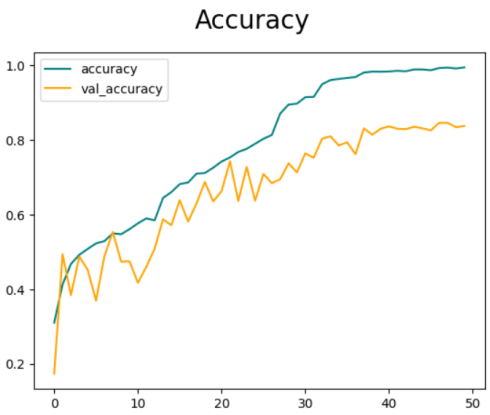

Figure 69: Accuracy

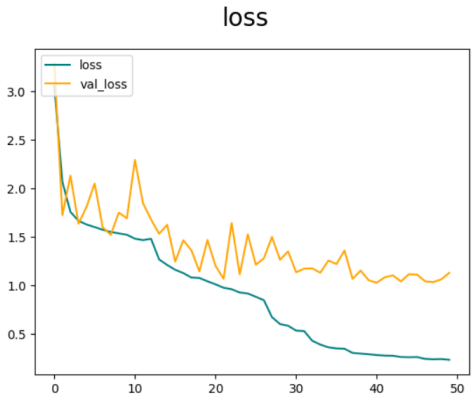

Figure 70: Loss

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8308 - loss: 1.1304
Test Loss: 1.088923454284668
Test Accuracy: 0.8287234306335449
```

Figure 71: Test Results

Figure 72: Test Results

6.8.2 CNN Architecture with 3 Convolutional Layers and Softmax Output

This architecture consists of three sequential convolutional layers with filter sizes of 32, 64, and 128 respectively, each using a 3×3 kernel with ReLU activation and same padding to preserve spatial dimensions. Each convolutional layer is followed by batch normalization to improve training stability and a max pooling layer to progressively reduce the spatial resolution while retaining key features. After the final convolutional stage, a global average pooling layer is applied to compress the feature maps into a single vector per feature map, reducing the risk of overfitting. This is followed by a fully connected dense layer with 128 units and ReLU activation to learn high-level representations. A dropout layer with a 0.5 rate is added to regularize the model by preventing co-adaptation of neurons.

The results of training this architecture are presented below:

```
model = Sequential()
model.add(Input(shape=(128, 128, 1)))

model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))

model.add(GlobalAveragePooling2D())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(7, activation='softmax'))
```

Figure 73: CNN Architecture

```
470/470 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9983 - loss: 0.0129 - val_accuracy: 0.8702 - val_loss: 0.6119 - learning_rate: 3.1250e-05
```

Figure 74: Results after the Last Epoch of Training

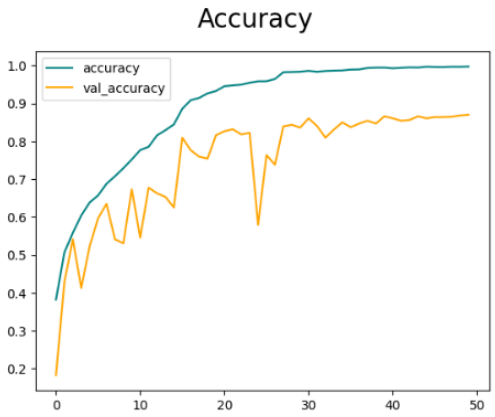

Figure 75: Accuracy

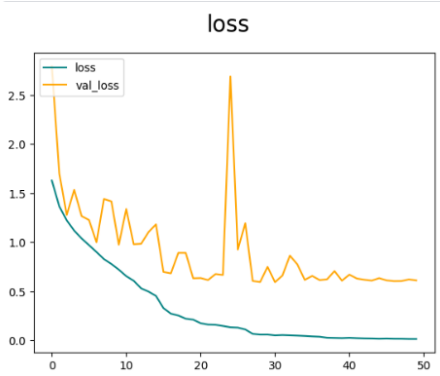

Figure 76: Loss

```
test_loss, test_accuracy = model.evaluate(x_test, y_test_ohe)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
30/30 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8705 - loss: 0.6747
Test Loss: 0.7204117178916931
Test Accuracy: 0.8542553186416626
```

Figure 77: Test Results

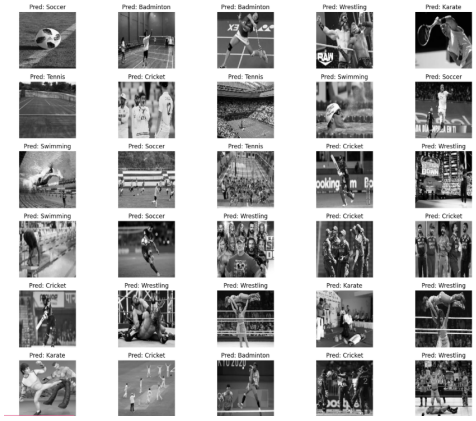

Figure 78: Test Results