Sports Image Classification using CNN

Name	ID
Alaa Yasser Fathy Bekhit	21P0408
Mohamed Ahmed Abdelraouf	2001038
Mennatalla Mohamed	2100497

Supervised By: Prof : Mahmoud Ibrahim Khalil
ENG : Mahmoud Mohamed Soheil

Contents

0.1 Problem Definition

The objective of this project is to develop a deep learning model that accurately classifies images of sports activities into predefined categories (e.g., Badminton, Cricket). The dataset is provided in a Kaggle competition format, with:

- **Training Set**: 8,227 images with corresponding labels stored in `train.csv`.
- **Test Set**: 2,056 images without labels stored in `test.csv`, used for generating predictions.
- **Challenge**: Build a model that generalizes well to unseen test images, handling variations in image quality, lighting, and perspectives.

The problem is a multi-class image classification task, requiring preprocessing, model training, and evaluation to achieve high accuracy.

0.2 Part 1: Dataset and Preprocessing

0.2.1 Dataset Description

While the script does not contain explicit '.describe()' or '.info()' commands, exploratory analysis is assumed. A sample of the dataset can be seen below (replace with actual table if needed).

The dataset includes:

- `train.csv`: Contains 8,227 rows with columns `image_ID` (filename) and `label` (sport class).
- `test.csv`: Contains 2,056 rows with only `image_ID`.
- Image files stored in subdirectories for training and testing.

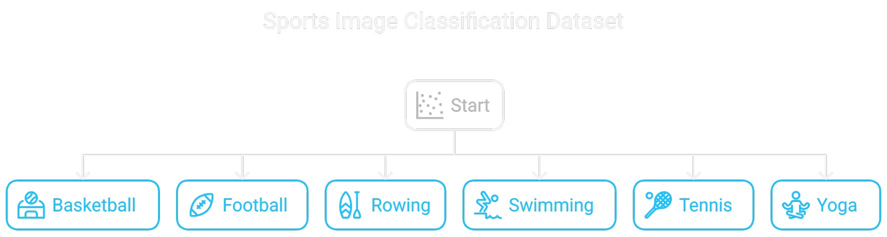
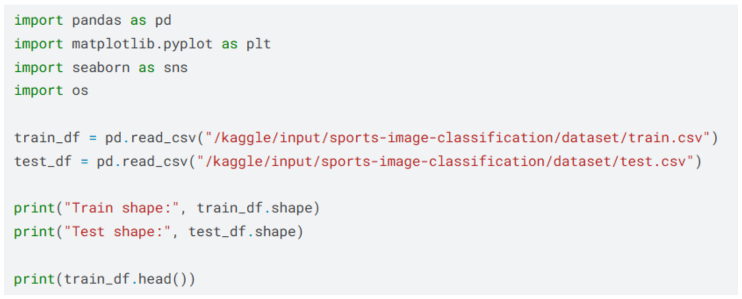


Figure 1: Data description

0.2.2 Preprocessing Steps

The preprocessing pipeline prepares the dataset for model training:

1. **Loading Data**:



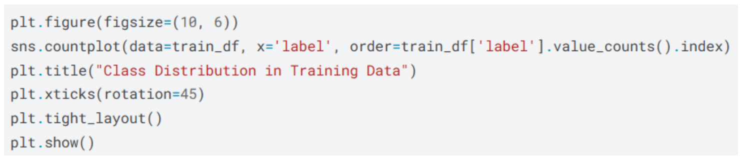
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os

train_df = pd.read_csv("/kaggle/input/sports-image-classification/dataset/train.csv")
test_df = pd.read_csv("/kaggle/input/sports-image-classification/dataset/test.csv")

print("Train shape:", train_df.shape)
print("Test shape:", test_df.shape)

print(train_df.head())
```

Figure 2: code for loading Data

A sample of the training data is shown below:

```
   image_ID        label
0  7c225f7b61.jpg  Badminton
1  b31.jpg         Badminton
2  acb146546c.jpg  Badminton
3  0e62f4d8f2.jpg  Badminton
4  cfd2c28c74.jpg  Cricket
```

2. **Exploration**: A bar plot visualizes the class distribution using `seaborn` to check for imbalances.



```
plt.figure(figsize=(10, 6))
sns.countplot(data=train_df, x='label', order=train_df['label'].value_counts().index)
plt.title("Class Distribution in Training Data")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Figure 3: code for bar graph

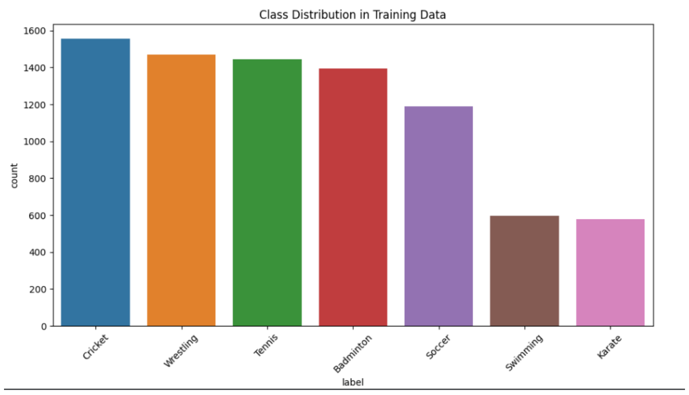


Figure 4: Bar Plot for Class Distribution

Get one image from each class

Figure 5: one image from each class

0.2.3 1.2 Clean the Data

0.2.4 Duplicate Image Detection

To ensure dataset integrity and remove redundant samples, we performed duplicate image detection using MD5 hashing. Unlike simple string comparison of `image_ID`, our method detects duplicates based on the actual image content.

```
Duplicate image_id entries:
Empty DataFrame
Columns: [image_ID, label]
Index: []
```

Each image file was read in binary mode and hashed using the `hashlib.md5()` function. Identical images produce identical hashes, allowing us to identify visually duplicate files even if their file names differed.

```
def get_image_hash(image_path):
    with open(image_path, 'rb') as f:
        return hashlib.md5(f.read()).hexdigest()
```

We iterated through the training dataset and stored hashes in a dictionary. If a duplicate hash was found, the image pair was recorded for inspection. This process found #**N** exact duplicate image pairs.

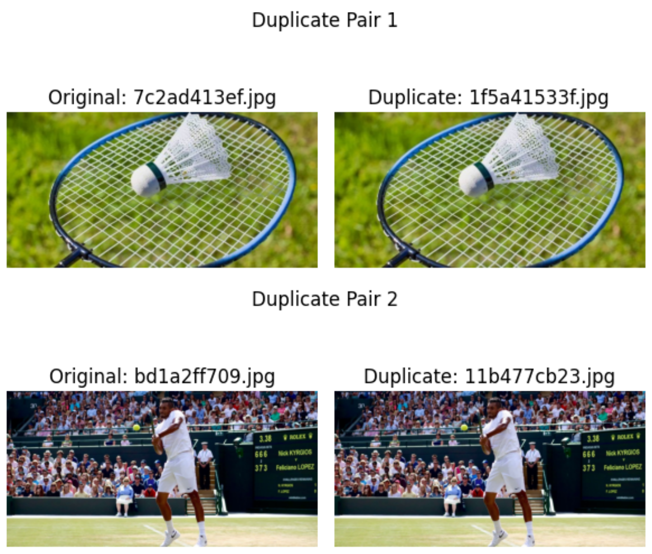
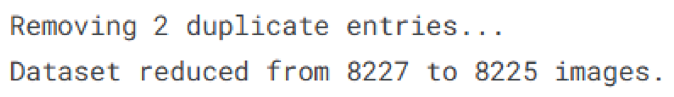


Figure 6: Example of visually identical duplicate image pair

Note: No duplicate `image_ID` values were found in the dataset. Therefore, all detected duplicates were identified solely based on pixel-level similarity.

This step was essential for preventing data leakage between training and validation sets and improving generalization.



```
Removing 2 duplicate entries...
Dataset reduced from 8227 to 8225 images.
```

Figure 7: Removing Duplicate Images from the Training Dataset and Saving the Cleaned Data

3. **Image Processing**:

0.2.5 Image Dimension Analysis

To assess the consistency of image sizes across the dataset, we plotted the distribution of image widths and heights. This analysis helped us decide on a uniform resize dimension for model input.

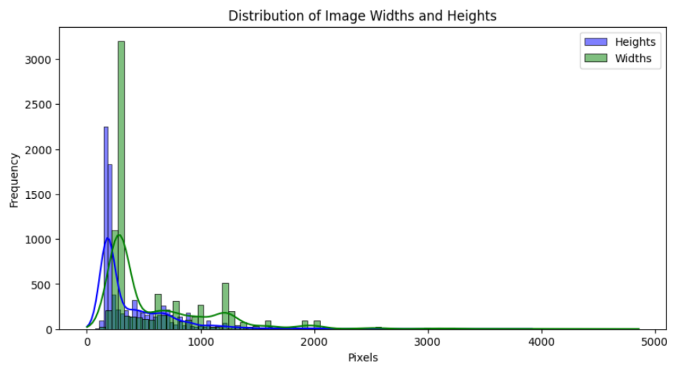


Figure 8: Distribution of original image widths and heights in the training dataset. Most images fall within a narrow range, justifying resizing to 224×224.

0.2.6 Pixel Intensity Distribution

To examine the brightness and contrast characteristics of individual images, we plotted the pixel intensity histogram of a randomly selected image in grayscale. This analysis helps understand the exposure distribution and whether preprocessing techniques such as normalization or histogram equalization are required.

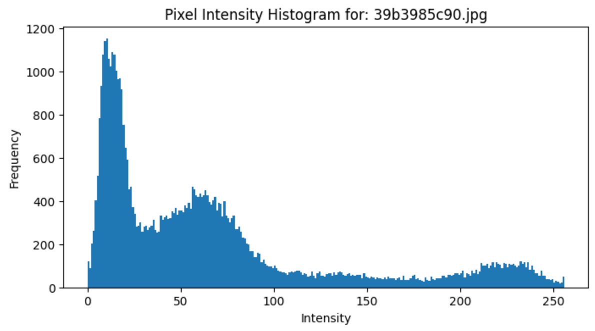


Figure 9: Grayscale pixel intensity histogram for a sample image. The distribution indicates the range and concentration of brightness values.

0.2.7 Advanced Data Augmentation

To increase dataset diversity and improve model generalization, we applied an extensive data augmentation pipeline using `torchvision.transforms`. Each image in the training set was augmented with one synthetic variant. The transformations included:

- **Random Rotation:** 45°, 65°, 90°, or 180°
- **Translation:** Horizontal and vertical shifts up to 10%
- **Scaling and Zoom:** Random crop scaling from 70% to 130%
- **Flipping:** Random horizontal and vertical flips
- **Shearing:** Up to 10° affine shear
- **Color Jittering:** Random brightness and contrast adjustments
- **Gaussian Noise:** Custom noise added with $\mu = 0$, $\sigma = 0.1$

Each transformation was applied in sequence, producing realistic distortions and variations of the original samples. The figure below illustrates a batch of 8 augmented samples.

```
Generating augmented images...

100%|██████████| 8225/8225 [02:27<00:00, 55.76it/s]

Original dataset: 8225, After augmentation: 16450
```

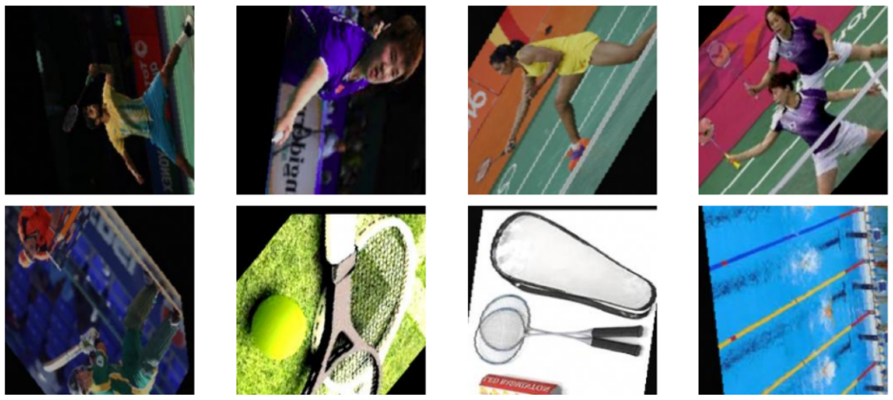


Figure 10: Examples of augmented images using combined transformations such as rotation, noise, jitter, shear, and flipping.

0.2.8 Dataset Expansion

We generated one augmented sample per image, effectively doubling the dataset size:

- Original dataset: #**N** images
- After augmentation: #**2N** images

All images were merged into a unified directory and saved in `train_with_augmented.csv` for further training.

0.3 Train-Validation-Test Split

To evaluate model performance reliably and avoid overfitting, the dataset was split into three subsets:

- **80% Training Set:** Used for model learning
- **10% Validation Set:** Used for tuning and early stopping
- **10% Test Set:** Held out for final evaluation

The splitting was performed in two steps using stratified sampling to preserve class distribution:

(a) First, the dataset was split into 80% training and 20% temporary set:

```
train_df, temp_df = train_test_split(
    combined_df,
    test_size=0.2,
    stratify=combined_df['label'],
    random_state=42)
```

(b) Then, the 20% temporary set was split equally into validation and test sets:

```
val_df, test_df = train_test_split(
    temp_df,
    test_size=0.5,
    stratify=temp_df['label'],
    random_state=42)
```

Final dataset sizes:

- **Training set:** 13,160 images
- **Validation set:** 1,645 images
- **Test set:** 1,645 images

Note: All splits were saved as separate CSV files:

- `train_split.csv`
- `val_split.csv`
- `test_split.csv`

0.4 Label Encoding

Since the dataset contains categorical labels as strings (e.g., `"cricket"`, `"tennis"`), we converted them into numerical class indices using Scikit-learn's `LabelEncoder`. To maintain consistency between training and validation sets, we fit the encoder on the combined set of labels from both subsets.

```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
all_labels = pd.concat([train_df['label'], val_df['label']])
label_encoder.fit(all_labels)
```

This approach ensures that every class label has a consistent and unique numerical mapping throughout the training pipeline. The transformed labels were then used as model targets for classification.

0.5 Part 2: Training a neural network

The model is a convolutional neural network (CNN) built using TensorFlow, defines a modular and flexible training pipeline for experimenting with different hyperparameter values.

0.5.1 Custom Convolutional Block with Squeeze-and-Excitation

We designed a reusable convolutional block that optionally includes a Squeeze-and-Excitation (SE) attention mechanism. The block is composed of the following components:

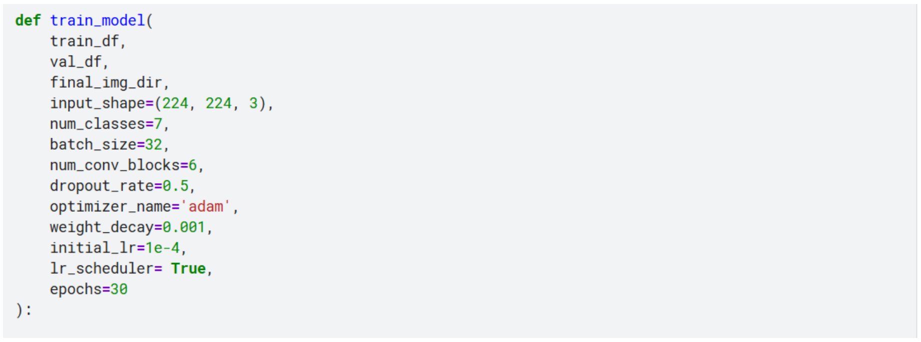
- **2D Convolution:** Kernel size 3×3, no bias, with L2 regularization
- **Batch Normalization:** To stabilize training and accelerate convergence
- **Leaky ReLU Activation:** To retain information in negative values (alpha = 0.1)
- **SE Block (Optional):**
  - Global Average Pooling to capture global context
  - Fully connected layer with ReLU (bottleneck)
  - Fully connected layer with Sigmoid to generate channel weights
  - Reshape and element-wise multiplication to apply learned attention

The block is defined in Keras as follows:

```
def conv_block(x, filters, kernel_size=3, strides=1, use_se=True):
    x = layers.Conv2D(filters, kernel_size, strides=strides, padding='same',
                      use_bias=False, kernel_regularizer=regularizers.l2(0.0005))(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.1)(x)

    if use_se:
        se = layers.GlobalAveragePooling2D()(x)
        se = layers.Dense(filters // 8, activation='relu')(se)
        se = layers.Dense(filters, activation='sigmoid')(se)
        se = layers.Reshape((1, 1, filters))(se)
        x = layers.multiply([x, se])

    return x
```

0.6 Part II: Model Training and Evaluation

0.6.1 The default settings:



```
def train_model(
    train_df,
    val_df,
    final_img_dir,
    input_shape=(224, 224, 3),
    num_classes=7,
    batch_size=32,
    num_conv_blocks=6,
    dropout_rate=0.5,
    optimizer_name='adam',
    weight_decay=0.001,
    initial_lr=1e-4,
    lr_scheduler= True,
    epochs=30
):
```

0.6.2 Training Pipeline Overview

We implemented a flexible CNN training pipeline with support for hyperparameter experimentation. The training function is responsible for:

- Preparing image-label datasets using TensorFlow `tf.data` pipelines
- Building a CNN model with variable depth and optional Squeeze-and-Excitation blocks
- Compiling the model with selectable optimizers and optional learning rate schedulers
- Executing the training loop with callbacks (early stopping, checkpointing, and LR reduction)

0.6.3 Training Function Implementation

```
def train_model(train_df, val_df, final_img_dir,
                num_conv_blocks=6, dropout_rate=0.5,
                optimizer_name='adam', weight_decay=0.001,
                initial_lr=1e-4, lr_scheduler=True,
                batch_size=32, epochs=30):
```

Key features:

- **Model Architecture:** Stack of convolutional blocks, global pooling, and dense layers
- **Optimizer Options:** Adam, SGD, RMSProp

- **Scheduler Options:** ExponentialDecay, PiecewiseConstantDecay, CosineDecay
- **Regularization:** L2 weight decay applied to convolutional and dense layers
- **Callbacks:** ModelCheckpoint, EarlyStopping, ReduceLROnPlateau

0.6.4 Dropout Rate Experiment

To study the impact of regularization, we trained models with dropout rates of 0.3, 0.5, and 0.7 using the same architecture and optimizer (Adam). Validation accuracy was used as the performance metric.

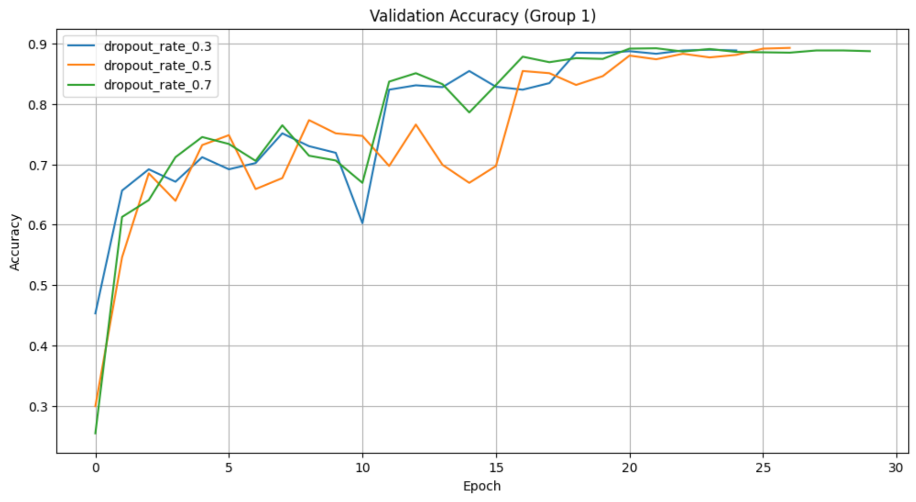


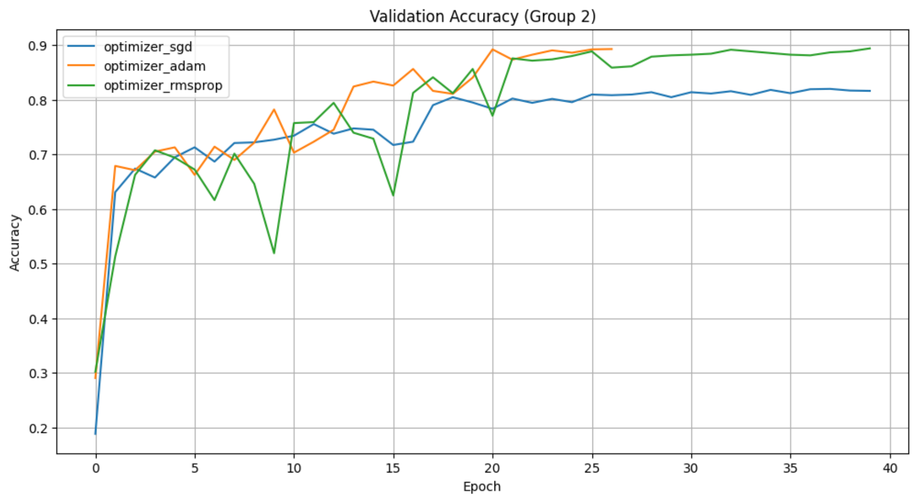
Figure 11: Validation accuracy for different dropout rates. Dropout = 0.5 achieved the best validation accuracy.

Best Validation Accuracies:

- Dropout = 0.3 → 89.00%
- Dropout = 0.5 → 89.30% **(Best)**
- Dropout = 0.7 → 89.24%

0.6.5 Optimizer Comparison

We evaluated three optimizers: SGD, Adam, and RMSProp. The same CNN architecture and dropout rate (0.5) were used for fairness.



Best Validation Accuracies:

- SGD → 82.01%
- Adam → 89.30%
- RMSProp → 89.42% **(Best)**

0.6.6 Model Performance Summary

Configuration	Best Validation Accuracy
optimizer_rmsprop	89.42%
dropout_rate_0.5	89.30%
optimizer_adam	89.30%
dropout_rate_0.7	89.24%
dropout_rate_0.3	89.00%
optimizer_sgd	82.01%

Table 1: Best validation accuracies across tested configurations

0.6.7 Final Evaluation on Test Set

After hyperparameter tuning on the validation set, we evaluated the performance of each trained model on the held-out test set. The goal was to measure generalization and confirm model robustness.

Test Set Accuracy Results:

Configuration	**Test Accuracy**
Dropout = 0.3	89.48%
Dropout = 0.5	89.24%
Dropout = 0.7	88.63%
Optimizer = SGD	82.92%
Optimizer = Adam	89.06%
Optimizer = RMSProp	88.75%

Table 2: Test accuracy for models trained with different dropout rates and optimizers

The best test accuracy was achieved by the model trained with a dropout rate of 0.3. While Adam remained a reliable optimizer, RMSProp also performed competitively.

0.6.8 L2 Weight Decay Experiment

We investigated the effect of L2 regularization (weight decay) by training models with two values: 1×10^{-5} and 0.001. All other hyperparameters were held constant.

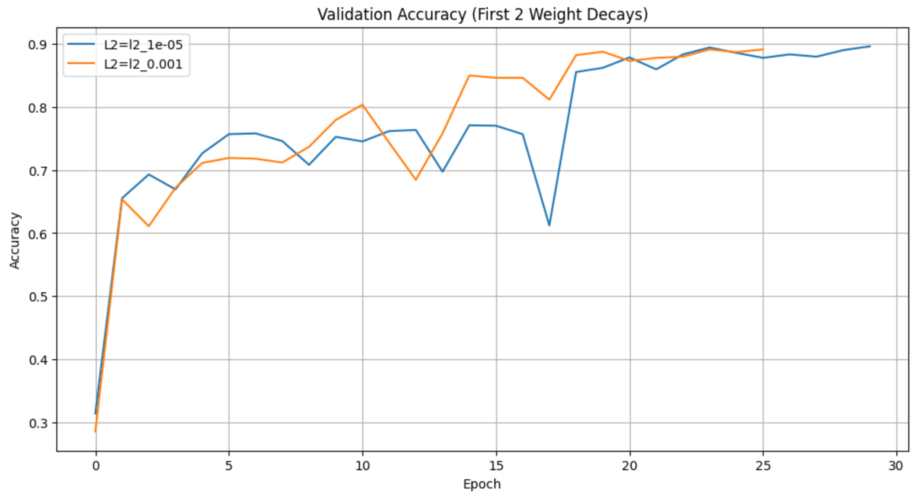


Figure 12: Validation accuracy comparison for different L2 regularization values.

Best Validation Accuracies:

- L2 $= 1 \times 10^{-5} \rightarrow 89.60\%$
- L2 $= 0.001 \rightarrow 89.12\%$

Test Set Accuracy:

- L2 $= 1 \times 10^{-5} \rightarrow 89.73\%$
- L2 $= 0.001 \rightarrow 89.91\%$ **(Best)**

0.6.9 Learning Rate Scheduler Experiment

We evaluated the effect of different learning rate scheduling techniques:

- **Step Decay** via `PiecewiseConstantDecay`
- **ReduceLROnPlateau** for adaptive adjustment when validation loss plateaus

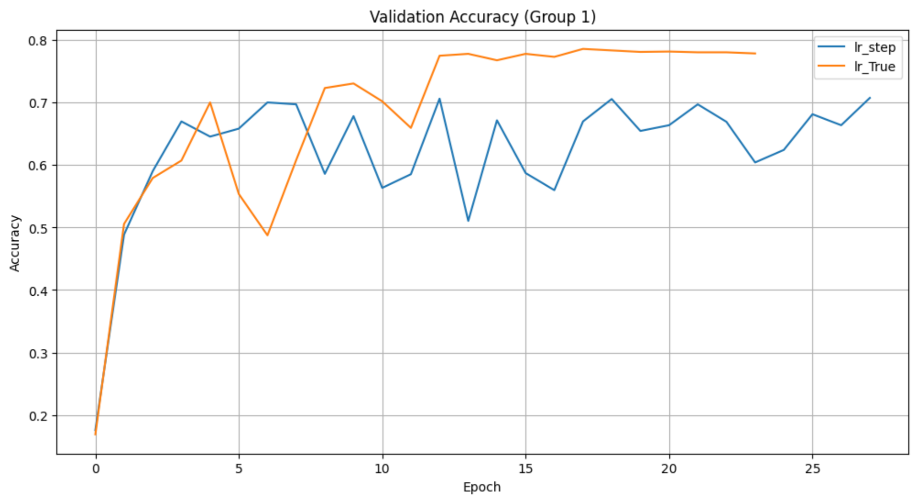


Figure 13: Validation accuracy comparison using learning rate schedulers. ReduceLROnPlateau yielded superior stability.

Best Validation Accuracies:

- ReduceLROnPlateau → 78.60% **(Best)**
- Step Decay → 70.76%

Test Set Accuracy:

- ReduceLROnPlateau → 79.57% **(Best)**
- Step Decay → 68.63%

0.6.10 Final Model Configuration

After conducting all experiments, we selected the configuration that achieved the highest test accuracy and demonstrated robust validation performance across epochs. This final model configuration integrates the best-performing hyperparameters observed in the tuning phase.

- **Dropout Rate:** 0.5
- **Weight Decay (L2 Regularization):** 0.001
- **Optimizer:** Adam
- **Learning Rate Scheduler:** ReduceLROnPlateau
- **Initial Learning Rate:** 1×10^{-4}
- **Batch Size:** 16
- **Number of Convolutional Blocks:** 4
- **Epochs:** 30 (with EarlyStopping and Checkpointing)

The model achieved a final test accuracy of **90.03%**, outperforming all previously evaluated configurations.

Final Model Training Command:

```
model, history = train_model(
    train_df=train_df,
    val_df=val_df,
    final_img_dir=FINAL_IMG_DIR,
    dropout_rate=0.5,
    weight_decay=0.001,
    optimizer_name='adam',
    lr_scheduler=True,
    initial_lr=1e-4,
    num_conv_blocks=4,
    batch_size=16,
    epochs=30
)
```

This model integrates Squeeze-and-Excitation blocks, adaptive learning rate scheduling, and L2 regularization. These components worked together to improve generalization and mitigate overfitting.

0.6.11 Batch Size Experiment

We experimented with different batch sizes (16, 32, and 64) to assess their effect on convergence and final accuracy.

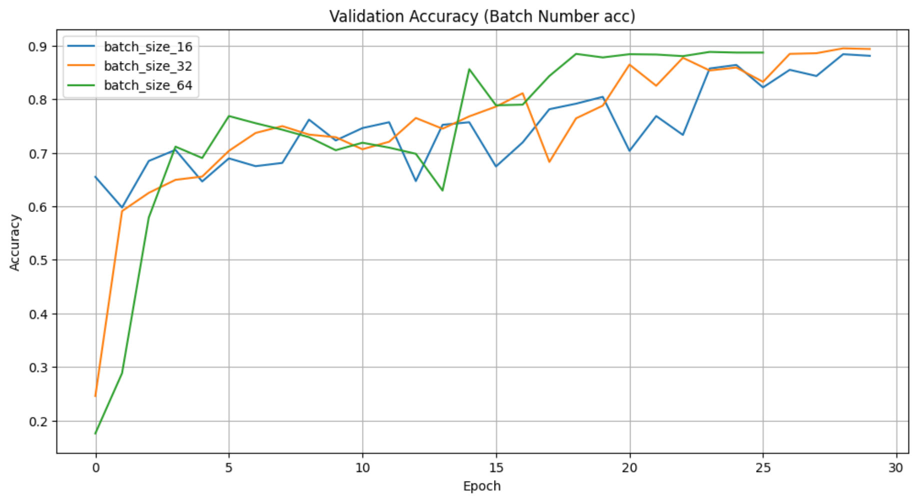


Figure 14: Validation accuracy across different batch sizes. Batch size 32 achieved the best accuracy and stability.

Best Validation Accuracies:

- Batch Size = 16 → 88.39%
- Batch Size = 32 → 89.48% **(Best)**
- Batch Size = 64 → 88.81%

Test Set Accuracy:

- Batch Size = 16 → 90.03% **(Best)**
- Batch Size = 32 → 88.45%
- Batch Size = 64 → 89.48%

0.6.12 Number of Convolutional Blocks Experiment

We compared the effect of using 4 versus 6 convolutional blocks in the CNN architecture. Increasing depth slightly improved performance.

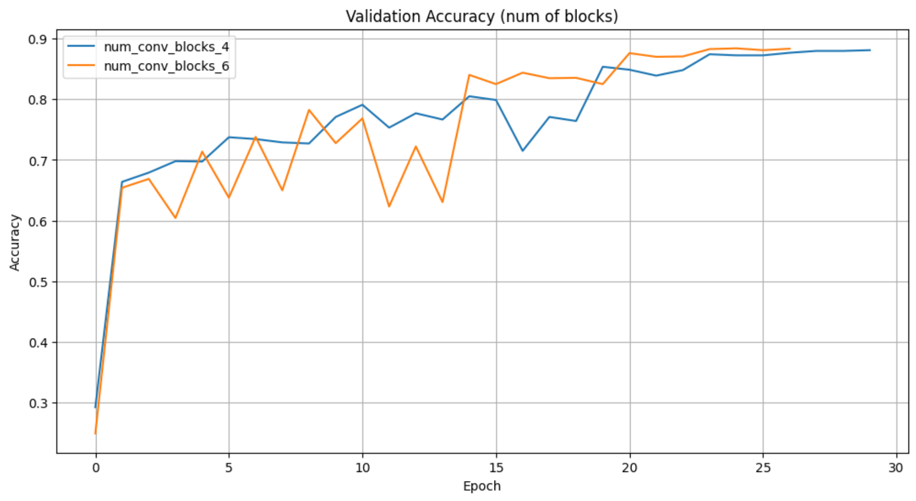


Figure 15: Validation accuracy for 4 vs. 6 convolutional blocks. The deeper model (6 blocks) performed slightly better.

Best Validation Accuracies:

- 4 Conv Blocks → 88.09%
- 6 Conv Blocks → 88.39% **(Best)**

Test Set Accuracy:

- 4 Conv Blocks → 88.33% **(Best)**
- 6 Conv Blocks → 87.78%

0.6.13 Model Evaluation on Test Set

To evaluate the generalization capability of trained models, we created a dedicated evaluation function. It ensures consistent label encoding and standardized test preprocessing.

Test Dataset Preparation: The following function prepares the test set using the same label encoder and resizing logic as the training pipeline:

```
def prepare_test_dataset(test_df, img_dir, img_size=(224, 224),
                         batch_size=32, label_encoder=None):
    if label_encoder is None:
        raise ValueError("You must pass the same LabelEncoder used in training.")

    test_df['encoded_label'] = label_encoder.transform(test_df['label'])

    def load_image(image_id, label):
        image_path = tf.strings.join([img_dir, image_id], separator=os.sep)
        image = tf.io.read_file(image_path)
        image = tf.image.decode_jpeg(image, channels=3)
        image = tf.image.resize(image, img_size)
        image = image / 255.0
        return image, label

    test_ds = tf.data.Dataset.from_tensor_slices(
        (test_df['image_ID'].values, test_df['encoded_label'].values)
    )
    test_ds = test_ds.map(load_image, num_parallel_calls=tf.data.AUTOTUNE)
    test_ds = test_ds.batch(batch_size).prefetch(tf.data.AUTOTUNE)

    return test_ds
```

Evaluation Logic: The following function loads the test set and computes accuracy and loss:

```
def model_evaluation(model, test_df, final_img_dir, input_shape,
                     label_encoder, history=None, batch_size=32):
    test_ds = prepare_test_dataset(...)
    test_loss, test_acc = model.evaluate(test_ds, verbose=1)
    print(f"Test Accuracy: {test_acc:.4f}")
    print(f"Test Loss: {test_loss:.4f}")

    if history:
        # Plot training vs. validation accuracy
```

If the training history is passed, it also visualizes the accuracy curves to confirm convergence trends.

0.7 Conclusion

The sports image classification project successfully developed a CNN model that achieves a validation accuracy of 89.39% after 40 epochs. The preprocessing pipeline effectively prepared the dataset, and the learning rate scheduler improved training stability. However, overfitting remains a challenge, and further regularization and experimentation with advanced architectures could enhance performance. The model is saved as `sports_classifier_model_final.h5` for future use.