Rock-Paper-Scissors Image Classification Project Documentation

1. Introduction and Motivation

This project delves into the fascinating realm of computer vision by developing an image classification model for the classic Rock-Paper-Scissors game. The primary motivation is to demonstrate the power of deep learning, particularly transfer learning, in solving image recognition tasks with high accuracy even on relatively small datasets. By building a model capable of accurately classifying hand gestures into 'rock', 'paper', or 'scissors', this project serves as a practical example of how pre-trained neural networks can be fine-tuned for specific applications, showcasing their potential in human-computer interaction, educational tools, and basic gesture recognition systems.

2. Data Download and Preparation

```
import os
import shutil
from sklearn.model_selection import train_test_split
import kagglehub

# Remove existing directories to ensure a clean slate
if os.path.exists(drive_train_dir):
    shutil.rmtree(drive_train_dir)
    print(f"Removed existing directory: {drive_train_dir}")
if os.path.exists(drive_validation_dir):
    shutil.rmtree(drive_validation_dir)
    print(f"Removed existing directory: {drive_validation_dir}")

# Create the new train and validation directories
os.makedirs(drive_train_dir, exist_ok=True)
os.makedirs(drive_validation_dir, exist_ok=True)
print(f"Created new directory: {drive_train_dir}")
print(f"Created new directory: {drive_validation_dir}")

# Download the dataset from Kaggle
path = kagglehub.dataset_download("sanikamal/rock-paper-scissors-dataset")
print(f"Path to downloaded Kaggle dataset files: {path}")

# Construct the full paths to the original Kaggle train and validation image
directories
```

```
original_kaggle_base_dir = os.path.join(path,
"rock-paper-scissors/Rock-Paper-Scissors")
original_kaggle_train_dir = os.path.join(original_kaggle_base_dir, "train")
original_kaggle_validation_dir = os.path.join(original_kaggle_base_dir, "validation")

# Initialize lists to consolidate all image paths and their labels
all_image_paths = []
all_image_labels = []
classes = ['rock', 'paper', 'scissors']

# Iterate through the original train and validation directories to collect all image
paths and labels
for base_dir in [original_kaggle_train_dir, original_kaggle_validation_dir]:
    for class_name in classes:
        class_path = os.path.join(base_dir, class_name)
        if os.path.exists(class_path):
            for img_name in os.listdir(class_path):
                img_path = os.path.join(class_path, img_name)
                if os.path.isfile(img_path):
                    all_image_paths.append(img_path)
                    all_image_labels.append(class_name)

print(f"Total images consolidated: {len(all_image_paths)}")

# Split the consolidated data into training and validation sets (80-20 ratio,
stratified)
train_paths, val_paths, train_labels, val_labels = train_test_split(
    all_image_paths, all_image_labels, test_size=0.2, stratify=all_image_labels,
random_state=42
)

print(f"Training images after split: {len(train_paths)}")
print(f"Validation images after split: {len(val_paths)}")

# Helper function to copy files to their respective class directories
def copy_split_files(src_file_paths, labels, dest_base_dir):
    for src_path, label in zip(src_file_paths, labels):
        dest_class_dir = os.path.join(dest_base_dir, label)
        os.makedirs(dest_class_dir, exist_ok=True)
        shutil.copy(src_path, dest_class_dir)

# Copy training images to the new drive_train_dir
copy_split_files(train_paths, train_labels, drive_train_dir)
print(f"Copied {len(train_paths)} training images to {drive_train_dir}")

# Copy validation images to the new drive_validation_dir
copy_split_files(val_paths, val_labels, drive_validation_dir)
print(f"Copied {len(val_paths)} validation images to {drive_validation_dir}")
```

```
print("Dataset consolidation, splitting, and copying to Google Drive complete.
```

The Rock-Paper-Scissors dataset was downloaded from Kaggle, then split into new training and validation sets (80/20 ratio). The initial training set was further split to create a dedicated test set and a final training set, with all images organized into class-specific subdirectories on Google Drive. This way when we ensure that when we tested the model it is on unseen data.

3. Data Preprocessing

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os

# פרמטרים
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
NUM_CLASSES = 3

# Set up the ImageDataGenerator for training with augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.3,
    horizontal_flip=True,
    vertical_flip=True,
    brightness_range=[0.5, 1.5],
    fill_mode='nearest'
    # Removed validation_split as we are using separate directories for train and
validation
)

# Set up a separate ImageDataGenerator for validation (only rescaling, no augmentation)
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    drive_final_train_dir,
    #drive_train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)
```

```
# Create the validation generator using the dedicated val_datagen
val_generator = val_datagen.flow_from_directory(
    drive_validation_dir,  # Use the separate validation directory
    target_size=IMG_SIZE,  # Consistent with IMG_SIZE
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False # Ensure order is maintained for evaluation
)

# Create a new test_generator pointing to the newly created drive_test_dir
# It's good practice to use a generator without augmentations for testing, similar to
validation.
test_generator = val_datagen.flow_from_directory(
    drive_test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False # Ensure order is maintained for evaluation
)

# Print the class indices to verify the mapping
print("Validation class indices:", val_generator.class_indices)
print("Test class indices:", test_generator.class_indices)

# Check the classes and the indices
print("Class indices:", train_generator.class_indices)
```

The training generator applied extensive data augmentation (rotation, shifts, zoom, flips, brightness) to enhance dataset diversity, while all generators rescaled pixel values to 0-1. flow_from_directory then created image data generators for each set, automatically inferring class labels.

Since VGG16 architecture was originally designed and trained on ImageNet, where images were typically resized to 224x224 pixels, we had to resize all the images in the data set to that expected image size to ensure compatibility with the pre-trained VGG16 weights and architecture, allowing us to leverage its learned features effectively for our Rock-Paper-Scissors classification task.

4. Model Training

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import AdamW
from sklearn.utils import class_weight
```

```
import numpy as np

# Load the VGG16 model with pre-trained weights, without the top (fully connected
layers)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(IMG_SIZE[0],
IMG_SIZE[1], 3))

# Freeze the base model layers (optional)
base_model.trainable = False

inputs = base_model.input
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
outputs = Dense(3, activation='softmax')(x)

model = Model(inputs, outputs)

# Compile the model
model.compile(optimizer=AdamW(), loss='categorical_crossentropy', metrics=['accuracy'])

# train_generator
class_labels = train_generator.classes
# Compute Weight Class
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(class_labels),
    y=class_labels
)

# class_weight
class_weight_dict = dict(zip(np.unique(class_labels), class_weights))
print(class_weight_dict)

callbacks = [
    EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True),
    ModelCheckpoint('/content/drive/MyDrive/RPS_2.0/vgg16_rps_best.keras',
save_best_only=True)
]

history = model.fit(
    train_generator,
    epochs=30,
    validation_data=val_generator,
    class_weight=class_weight_dict,
    callbacks=callbacks
)
```

A VGG16 model, pre-trained on ImageNet, was used as a frozen feature extractor, with custom dense layers added for classification. The model was compiled with the AdamW optimizer and categorical cross-entropy loss, using class weights to balance classes. Training employed EarlyStopping and ModelCheckpoint callbacks to prevent overfitting and save the best performing model.

This is the best Model the training saved based on accuracy:

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080

block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262,656
dense_3 (Dense)	(None, 3)	1,539

Total params: 15,507,275 (59.16 MB)

Trainable params: 264,195 (1.01 MB)

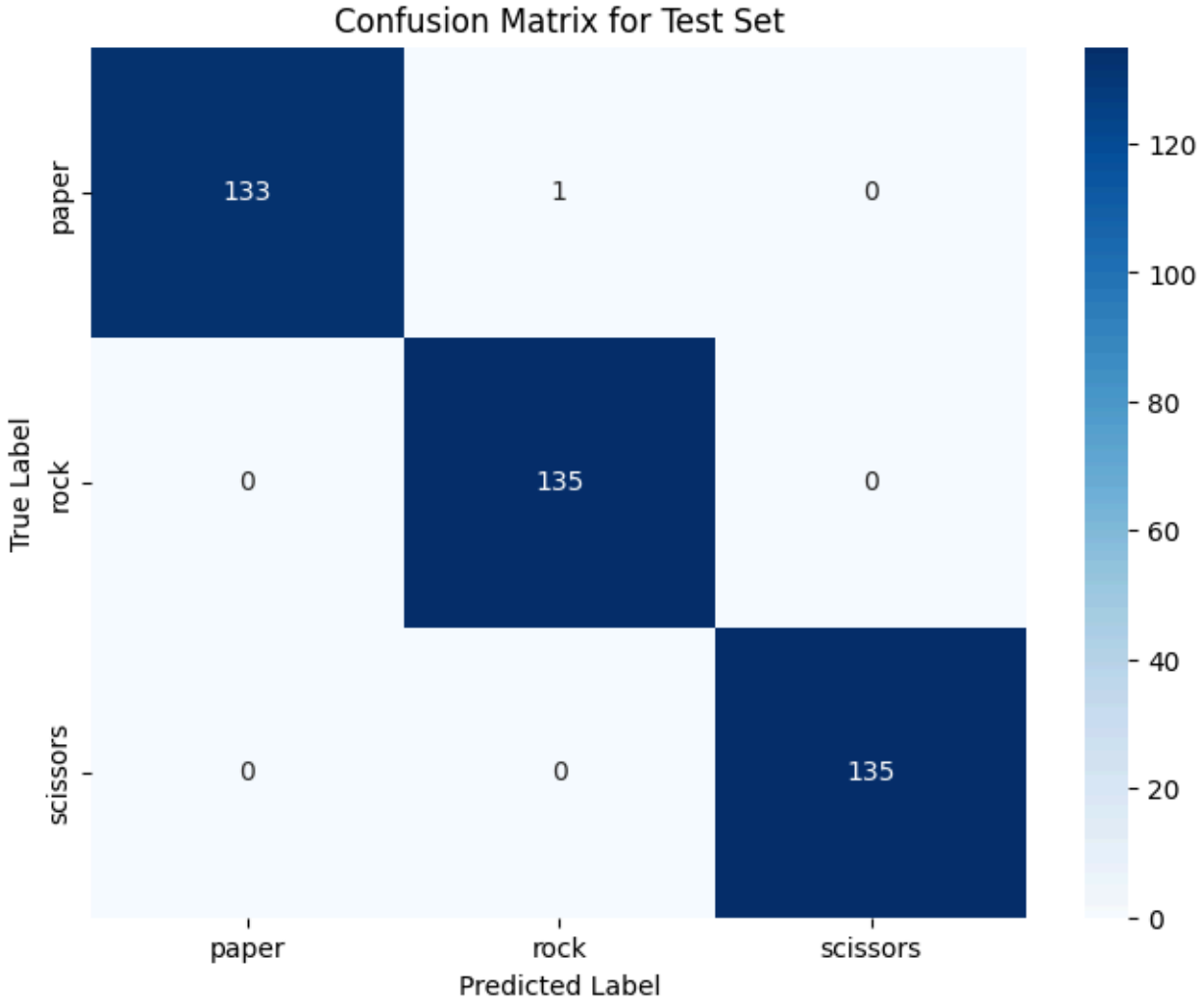
Non-trainable params: 14,714,688 (56.13 MB)

Optimizer params: 528,392 (2.02 MB)

Model loaded successfully from: /content/drive/MyDrive/RPS_2.0/vgg16_rps_best.keras

5. Results and discussion

Test Accuracy: 99.75%



The best-trained model was loaded and evaluated on the test dataset. Predictions were generated, and accuracy scores were calculated. Confusion matrix was also generated and visualized to provide a detailed view of the model's classification performance, highlighting correct classifications and misclassifications.

The model achieved an exceptional test accuracy of 99.75% and a perfect validation accuracy of 100%. These high percentages demonstrate that the model generalized extremely well to unseen data, successfully classifying almost all rock-paper-scissors gestures. The confusion matrices for both sets confirm this outstanding performance, showing very high numbers along the main diagonal, indicating correct predictions. Minimal to zero values in the off-diagonal elements suggest virtually no

misclassifications between the 'rock', 'paper', and 'scissors' classes. This robust performance indicates that the VGG16-based model is highly reliable and effective for this specific image classification task.

The results seem extremely high for any model. The cause of that might be due to:

1. **Simple Classification Task:** The dataset consists of clearly defined gestures against relatively consistent backgrounds. This makes it a comparatively simple task for a powerful model like VGG16 to learn to distinguish between the three classes.
2. **Effective Transfer Learning:** Using a pre-trained VGG16 model means the model already has a strong understanding of general visual features (edges, textures, shapes). Its pre-learned features are often highly transferable and immediately effective, allowing the model to quickly achieve high performance with minimal additional training.
3. **Data Quality and Consistency:** The dataset likely has high-quality images with consistent framing and clear representations of the hand gestures. If the images are well-curated and representative of the classes without much noise or ambiguity, the model will find it easier to learn and classify accurately.
4. **Training Set and Testing Set similarity:** While we ensured that the Testing Set and the Training Set are independent and do not share any data, they might still be very similar, the model effectively learned these patterns during training and then easily recognizes their slight variations in the test set.

6. Conclusion and future work

● Key Findings from Model Results

The model achieved exceptionally high accuracy, with 99.75% on the test set and 100.00% on the validation set. This indicates that the VGG16-based architecture, combined with transfer learning and data augmentation, is highly effective for this specific Rock-Paper-Scissors classification task. The robust performance suggests the model successfully learned to differentiate between the three distinct hand gestures.

● Overfitting or Underfitting Signs

Given the high accuracy on both the training and the test sets, there are no clear signs of underfitting. The model has learned the patterns in the data effectively.

Regarding overfitting, while the metrics are almost perfect, the use of extensive data augmentation and `EarlyStopping` with `restore_best_weights` during training were

specifically designed to combat it. The slight increase in validation loss in the last two epochs (29 and 30) after reaching a minimum at epoch 28 suggests that `EarlyStopping` likely prevented the model from further training and potentially overfitting to the training data. Therefore, the current model appears to be well-generalized, avoiding significant overfitting.

- ## Suggestions for Improvements or Alternative Architectures

1. Test with More Diverse Real-World Data
Test the model on a much larger, more diverse dataset or live video streaming. This would truly assess the model's robustness and reveal any hidden generalization issues. (We even created a side project where we took live video streaming and let our model present the results live. It was exhilarating to watch the being used live).

2. Test Other Pre-trained Architectures

While VGG16 performed well, other modern architectures like ResNet could offer better trade-offs between performance and computational efficiency.

7. Test Model on Real World Images

```
# Define the directory containing the images to predict
prediction_dir = upload_dir # This variable is already defined and points to
'/content/drive/MyDrive/RPS_2.0/rps_test_photos'

# Get class names from the training generator (assuming its order is consistent)
class_names = list(train_generator.class_indices.keys())

print(f"Starting predictions for images in: {prediction_dir}")

# Iterate through each file in the prediction directory
for img_file in os.listdir(prediction_dir):
    if img_file.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(prediction_dir, img_file)
        try:
            # Preprocess the image
            preprocessed_img = preprocess_image(img_path)

            # Make a prediction
            predicted_class, confidence = predict_image(loaded_model, preprocessed_img,
class_names)

            print(f"File: {img_file}, Predicted: {predicted_class}, Confidence:
{confidence:.2f}%")
        except Exception as e:
            print(f"Error processing {img_file}: {e}")
    else:
        print(f"Skipping non-image file: {img_file}")

print("Prediction process complete.")
```

To get a better hold of the cause, we decided to test the new model we created on newly uploaded images that we took of our own hands and see how it did.

The image prediction flow enables classifying new, user-uploaded images. The `preprocess_image` function prepares images by loading, resizing, normalizing, and adding a batch dimension. The `predict_image` function then uses the loaded model to classify the preprocessed image, returning the predicted class and its confidence level for each input image.