Deep Learning Project Documentation: **Image Classification Using Transfer Learning**

Team 4:

Mihajlo Pasic

Mohammed Dele

Ahmed Abdinur Ahmed

Contents

1. Introduction

In the era of artificial intelligence and deep learning, image classification is a fundamental problem with applications spanning various domains, from healthcare to autonomous vehicles. This document presents an in-depth overview of a deep

learning project conducted by Team 4, focusing on image classification using transfer learning techniques. The project leveraged pre-trained convolutional neural network (CNN) models to classify images into multiple classes.

2. Project Overview

2.1 Objective

The primary objective of this project was to design and implement an image classification system capable of accurately categorizing images into predefined classes. The specific goals included:

- Exploring transfer learning with pre-trained CNN architectures.
- Customizing and fine-tuning pre-trained models for a specific classification task.
- Evaluating model performance using validation metrics.
- Generating predictions for new images and submitting results to Kaggle for evaluation.

2.2 Dataset

The project utilized a custom dataset curated specifically for image classification. The dataset comprised images categorized into multiple classes, with a sufficient number of samples for training and validation. The dataset was preprocessed and split into training and validation sets for model development and evaluation.

3. Methodology

3.1 Data Preprocessing

Data preprocessing played a crucial role in ensuring the quality and diversity of the dataset. The following preprocessing techniques were applied:

- Image Resizing: All images were resized to a uniform size (e.g., 224x224 pixels) to ensure consistency across the dataset.
- Data Augmentation: Augmentation techniques such as rotation, shifting, and flipping were employed to increase the variability of the training data and enhance model generalization.
- Normalization: Pixel values of the images were normalized to a range of [0,1] to facilitate model training.

3.2 Model Selection

Two state-of-the-art pre-trained CNN models were selected for transfer learning:

- VGG16: A classic deep CNN architecture known for its simplicity and effectiveness in feature extraction.

- DenseNet121: A more recent CNN architecture that emphasizes feature reuse through dense connections between layers.

These models were chosen for their proven performance in various computer vision tasks and availability of pre-trained weights on large-scale image datasets.

3.3 Model Development

Custom classifier layers were added on top of the pre-trained CNN architectures to adapt them to the specific image classification task. The classifier typically consisted of dense (fully connected) layers followed by activation functions (e.g., ReLU) and dropout layers to prevent overfitting.

3.4 Model Training

The models were trained using the augmented training dataset with specified parameters:

- Batch Size: The number of samples processed per iteration during training (e.g., 32 or 64).
- Epochs: The number of complete passes through the entire training dataset during training.
- Optimizer: The optimization algorithm used to update the model weights based on the training data (e.g., Adam).

During training, model performance was monitored using validation metrics to assess convergence and prevent overfitting.

3.5 Model Evaluation

Upon completion of training, the models were evaluated using the validation dataset to assess their performance in terms of accuracy and loss. Evaluation metrics such as precision, recall, and F1-score were computed to analyze class-wise performance.

3.6 Prediction Generation

The trained models were utilized to generate predictions for new images. Each image was preprocessed using the same techniques applied during training, and the predicted class labels were stored alongside image IDs for submission.

4. Results

4.1 Model Performance

The trained models exhibited strong performance on the validation dataset, achieving high accuracy and demonstrating robustness in classifying images into the predefined categories. Performance metrics were computed to quantify the effectiveness of the models in differentiating between classes.

4.2 Kaggle Submission

To evaluate the real-world performance of the models, predictions were submitted to Kaggle for scoring. The project achieved a notable score of 0.93886, indicating the efficacy of the developed models in classifying unseen images.

5. Conclusion

In conclusion, the deep learning project conducted by Team 4 successfully implemented transfer learning techniques to build an image classification system using pre-trained CNN models. The project showcased effective model training, evaluation, and prediction generation, highlighting the potential of deep learning in image analysis tasks.

The achieved Kaggle score of 0.93886 reflects the robustness and accuracy of the models, underscoring their practical utility in real-world applications. The project outcomes provide valuable insights for future research and development in the field of computer vision and deep learning.

---

6. Important Code Snippets and Their Explanations

Although the code snippets of this project were briefly explained in the Colab notebook, this section provides a short explanation of some of the key code segments. These code blocks encompass various aspects of the workflow, including data preprocessing, model training, evaluation, and prediction. Each code block will also include short comments to explain the code does.

1. Necessary libraries for building and training the model

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorflow.keras.applications import VGG16
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import csv
from PIL import Image
import cv2
```

Explanation:

This snippet imports the necessary libraries for building and training the model, performing image preprocessing, evaluating the model, and visualizing the results. TensorFlow and Keras are used for deep learning, while Matplotlib and Seaborn are used for plotting and visualization. Additional libraries like PIL and OpenCV are used for image processing tasks.

2. Setting Constants and Hyperparameters

```
# Define constants for the project
NUM_CLASSES = 5  # Number of classes for classification
IMG_SIZE = (224, 224)  # Image size expected by VGG16
BATCH_SIZE = 32  # Batch size for training
epochs = 20  # Number of epochs for training
```

Explanation:

These constants define the number of classes for classification, the image size expected by the VGG16 model, the batch size for training, and the number of epochs for which the model will be trained. Adjusting these parameters can significantly affect the training process and model performance.

3. Data Loading and Augmentation

```
# Set up directories for training and validation data
train_dir = '/path/to/train_data'
validation_dir = '/path/to/validation_data'

# Initialize ImageDataGenerator for data augmentation and preprocessing
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

validation_datagen = ImageDataGenerator(rescale=1./255)
```

```
# Create data generators for training and validation
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)
```

Explanation:

This snippet sets up data generators for training and validation datasets. The ImageDataGenerator class is used for real-time data augmentation, which helps in creating a diverse set of training images by applying random transformations. This improves the model's generalization capabilities.

4. Loading Pre-Trained VGG16 Model

```
# Load the VGG16 model pre-trained on ImageNet, excluding the top layers
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3))
base_model.trainable = False  # Freeze the convolutional base
```

Explanation:

The pre-trained VGG16 model is loaded with weights trained on the ImageNet dataset. The top layers of the model are excluded (include_top=False), and the convolutional base is frozen (trainable=False) to retain the learned features and prevent them from being updated during our training process.

5. Custom Classifier Addition

```
# Add custom classifier on top of the VGG16 base
x = base_model.output
x = Flatten()(x)  # Flatten the output of the convolutional base
```

```
x = Dense(512, activation='relu')(x)  # Fully connected layer with 512 units
and ReLU activation
x = Dropout(0.4)(x)  # Dropout layer with 40% dropout rate for regularization
predictions = Dense(NUM_CLASSES, activation='softmax')(x)  # Output layer with
softmax activation for classification

# Create the final model
model = Model(inputs=base_model.input, outputs=predictions)
```

Explanation:

A custom classifier is added on top of the VGG16 base model. It consists of a Flatten layer to convert 3D feature maps to 1D feature vectors, a Dense layer with 512 units and ReLU activation, a Dropout layer for regularization, and a final output layer with softmax activation for multi-class classification.

6. Model Compilation

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

Explanation:

The model is compiled using the Adam optimizer, which adjusts the learning rate dynamically during training. The loss function used is categorical cross-entropy, suitable for multi-class classification tasks, and accuracy is set as the evaluation metric.

7. Callback Definitions

```
# Define callbacks for early stopping and learning rate reduction
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3,
min_lr=0.0001)
```

Explanation:

Two callbacks are defined: EarlyStopping, which stops training when the validation loss stops improving, and ReduceLROnPlateau, which reduces the learning rate

when the validation loss plateaus. These callbacks help in preventing overfitting and ensuring better convergence.

8. Model Training

```
# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // BATCH_SIZE,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // BATCH_SIZE,
    callbacks=[early_stopping, reduce_lr]
)
```

Explanation:

The model is trained using the fit method. The training and validation data generators are passed along with the defined callbacks. The number of steps per epoch and validation steps are calculated based on the number of samples and batch size.

9. Model Evaluation

```
# Evaluate the model on the validation set
test_loss, test_accuracy = model.evaluate(validation_generator,
steps=validation_generator.samples // BATCH_SIZE)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

Explanation:

The model is evaluated on the validation set to compute the test loss and accuracy. This provides an indication of the model's performance on unseen data.

10. Generating Predictions

```
# Generate predictions on the validation set
predictions = model.predict(validation_generator)
predicted_classes = np.argmax(predictions, axis=1)  # Get the predicted
classes
true_classes = validation_generator.classes  # Get the true classes
```

```
# Print classification report
print("Classification Report:")
print(classification_report(true_classes, predicted_classes))

# Compute and plot confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Explanation:

This snippet generates predictions for the validation set and computes the predicted classes. It then prints a classification report showing precision, recall, F1-score, and support for each class. The confusion matrix is calculated and visualized as a heatmap using Matplotlib and Seaborn, providing a clear view of the model's performance across different classes.

11. Saving and Loading the Model

```
# Save the trained model
model.save('path/to/save_model.h5')

# Load the trained model
loaded_model = tf.keras.models.load_model('path/to/save_model.h5')
```

Explanation:

This snippet demonstrates how to save a trained model to a file using model.save() and load it back using tf.keras.models.load_model(). This is useful for preserving the trained model for future use or deployment.

12. Preprocessing New Images for Prediction

```
# Function to preprocess new images for prediction
def preprocess_image(img_path):
    img = image.load_img(img_path, target_size=(IMG_SIZE[0], IMG_SIZE[1]))  # 
Load and resize the image
    img_array = image.img_to_array(img)  # Convert image to array
```

```
    img_array = np.expand_dims(img_array, axis=0) / 255.0  # Expand dimensions
and normalize pixel values
    return img_array

# Path to directory containing new images
new_images_dir = 'path/to/new_images'

# Lists to store predictions and image IDs
predictions = []
image_ids = []

# Iterate through the images in the directory
for filename in os.listdir(new_images_dir):
    if filename.endswith(".jpg") or filename.endswith(".png"):
        img_path = os.path.join(new_images_dir, filename)
        img_array = preprocess_image(img_path)  # Preprocess the image
        prediction = model.predict(img_array)  # Make prediction
        predicted_class = np.argmax(prediction)
        predictions.append(predicted_class)  # Store the predicted class
        image_ids.append(filename)  # Store the image ID

# Save predictions to CSV
output_csv_file = 'predictions.csv'
with open(output_csv_file, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["ImageID", "PredictedClass"])
    writer.writerows(zip(image_ids, predictions))

print(f"Predictions saved to {output_csv_file}")
```

Explanation:

This snippet defines a function preprocess_image() to load and preprocess images for prediction. It then iterates through a specified directory of new images, preprocesses each image, and uses the trained model to make predictions. The predicted class labels and corresponding image IDs are stored in lists and saved to a CSV file.

Code snippets Conclusion

These code snippets collectively illustrate the process of setting up a deep learning model using transfer learning with VGG16, performing data augmentation, compiling and training the model, evaluating its performance, making predictions on new data, and visualizing the results. Each step is crucial in building an effective image classification model capable of handling multi-class tasks. By leveraging the power

of pre-trained models and applying appropriate training techniques, we can achieve high accuracy and robust performance even with limited data.