6. Object Detection using Transfer Learning of CNN Architectures

Aim: To implement object detection using transfer learning by leveraging a pre-trained Convolutional Neural Network (CNN) architecture.

Theory: **Object detection** is a computer vision task that involves identifying and localizing objects within an image by drawing bounding boxes around them and assigning a class label. Training an object detection model from scratch is computationally expensive and requires massive datasets. **Transfer learning** offers a highly effective solution.

The process leverages a CNN pre-trained on a large-scale image classification dataset (like ImageNet). This pre-trained model has already learned to extract rich hierarchical features from images.

Steps:

- **a. Load in a Pre-trained CNN Model**: We start by loading a popular CNN architecture (e.g., VGG16, ResNet50, MobileNetV2) that has been pre-trained on ImageNet. Typically, we load the **convolutional base** of the model, excluding its final classification layers, as these are task-specific.
- **b. Freeze Parameters (Weights) in the Model's Lower Convolutional Layers**: To retain the general feature extraction capabilities learned from ImageNet, we freeze the weights of the pre-trained convolutional base. This prevents them from being updated during the initial stages of training on our specific dataset, avoiding catastrophic forgetting of learned features.
- **c. Add a Custom Classifier with Trainable Parameters**: On top of the frozen convolutional base, we add new layers designed for object detection. This often includes:
    - A few Dense layers for high-level feature processing.
    - Specific layers tailored for object detection, such as region proposal networks or bounding box regression and classification heads (depending on the object detection framework used, e.g., Faster R-CNN, YOLO, SSD). For simplicity here, we'll illustrate adding a classifier for bounding box prediction and class probabilities.
    - **Dropout** layers can be added for regularization.
- **d. Train Classifier Layers on Training Data**: The model is compiled and trained. During this phase, only the weights of the newly added custom layers are updated. The pre-trained base acts as a fixed feature extractor. We use a suitable optimizer (e.g., Adam) and loss function. For object detection, the loss is typically a combination of classification loss (e.g., categorical cross-entropy) and regression loss (e.g., smooth L1 loss for bounding box coordinates).

- **e. Fine-tune Hyperparameters and Unfreeze More Layers (Optional but Recommended)**: After the custom layers have been trained, we can optionally **fine-tune** the model. This involves:
    - **Unfreezing** some of the top layers of the pre-trained convolutional base.
    - Recompiling the model with a very **low learning rate**.
    - Training the model again. This allows the pre-trained features to adapt slightly to the specifics of the new task, potentially improving performance further. Hyperparameters like learning rate, batch size, and optimizer choice are critical here.

Code Example (Conceptual - Fine-tuning for classification as a proxy for detection steps):

Note: Implementing a full object detection model (like Faster R-CNN or YOLO) is complex and beyond a simple manual example. This code demonstrates the core transfer learning concept by fine-tuning a pre-trained model for classification on a dataset like CIFAR-10, which is a stepping stone. For actual object detection, you'd integrate specific detection heads.

Python

```
import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.optimizers import Adam

import matplotlib.pyplot as plt


# --- a. Load in a pre-trained CNN model trained on a large dataset ---


# Load VGG16 pre-trained on ImageNet, without the top classification layers

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# input_shape is set to 32x32 for CIFAR-10. VGG16 was originally trained on 224x224,

# but Keras allows adapting it. However, for best results, consider resizing CIFAR-10 to 224x224 or using models designed for smaller inputs.
```

```
# --- b. Freeze parameters (weights) in the model's lower convolutional layers ---
# Freeze all layers in the base model initially
for layer in base_model.layers:
    layer.trainable = False

# --- c. Add a custom classifier with several layers of trainable parameters ---
x = base_model.output
x = Flatten()(x) # Flatten the output of the convolutional base
x = Dense(512, activation='relu')(x) # Add a dense layer
x = Dropout(0.5)(x) # Add dropout for regularization
# Add the final output layer for classification (CIFAR-10 has 10 classes)
num_classes = 10
predictions = Dense(num_classes, activation='softmax')(x) # Softmax for multi-class
classification

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

model.summary()

# --- d. Train classifier layers on training data available for the task ---

# Load and preprocess CIFAR-10 data (similar to previous example)
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

```
# Compile the model
# Use Adam optimizer with a moderate learning rate for the initial training
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model (only the new layers are trained)
print("\nTraining the custom classifier layers...")
batch_size = 64
epochs = 5 # Reduced epochs for demonstration
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test))

# --- e. Fine-tune hyperparameters and unfreeze more layers as needed ---
print("\nStarting fine-tuning...")

# Unfreeze some top layers of the base model
# For VGG16, let's unfreeze the last convolutional block (e.g., conv5)
# This requires knowing the layer names/indices. VGG16 has blocks 'block5_conv1',
'block5_conv2', 'block5_conv3'.
# Let's unfreeze from 'block5_conv1' onwards.
for layer in base_model.layers[20:]: # Layers from index 20 onwards in VGG16 are part of
'block5'
    layer.trainable = True

# Re-compile the model with a very low learning rate for fine-tuning
# A low learning rate prevents large updates that could destroy learned features
```

```
model.compile(optimizer=Adam(learning_rate=0.00001), # Very low learning rate
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary() # Check which layers are now trainable

# Train again for fine-tuning
print("Fine-tuning the model with unfreezed layers...")
fine_tune_epochs = 5 # More epochs for fine-tuning might be needed
total_epochs = epochs + fine_tune_epochs

history_fine = model.fit(x_train, y_train,
                         batch_size=batch_size,
                         epochs=fine_tune_epochs, # Train for additional epochs
                         initial_epoch=history.epoch[-1] + 1, # Continue from where we left off
                         validation_data=(x_test, y_test))

# Combine histories for plotting
history.history['accuracy'].extend(history_fine.history['accuracy'])
history.history['val_accuracy'].extend(history_fine.history['val_accuracy'])
history.history['loss'].extend(history_fine.history['loss'])
history.history['val_loss'].extend(history_fine.history['val_loss'])

# Plot training history after fine-tuning
plt.figure(figsize=(14, 5))

# Plotting Accuracy
plt.subplot(1, 2, 1)
plt.plot(range(1, total_epochs + 1), history.history['accuracy'], label='Training Accuracy')
```

```
plt.plot(range(1, total_epochs + 1), history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy (After Fine-tuning)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plotting Loss
plt.subplot(1, 2, 2)
plt.plot(range(1, total_epochs + 1), history.history['loss'], label='Training Loss')
plt.plot(range(1, total_epochs + 1), history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss (After Fine-tuning)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Evaluate the final model
print("\nEvaluating the final fine-tuned model...")
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Final Test Loss: {loss:.4f}")
print(f"Final Test Accuracy: {accuracy:.4f}")
```

Conclusion: Transfer learning significantly accelerates the development of effective models, especially for tasks like object detection where data is scarce or training is resource-intensive. By leveraging pre-trained features, we can build powerful models with less data and computational effort. Fine-tuning further refines the model's performance by adapting pre-

trained features to the specific nuances of the target task. This approach is fundamental to many state-of-the-art computer vision applications.