Title: Fine-tuning ResNet50 for Image Classification

Abstract:

This code showcases the usage of the ResNet50 convolutional neural network (CNN) model as the base model for fine-tuning or transfer learning. By freezing the base model's layers and adding additional layers, we construct a new model capable of classifying images in a specific task. This report provides an explanation of the code and presents a formatted version for clear understanding and implementation.

Introduction What is Resnet50

ResNet-50 is a deep neural network architecture that has been widely used for various computer vision tasks, such as image recognition, object detection, and segmentation. The name "ResNet" stands for "Residual Network", which refers to the way the network is constructed using residual connections.

The idea behind residual connections is to enable the network to learn residual mappings, i.e., the difference between the input and output of a layer. This is achieved by adding a shortcut connection that bypasses one or more layers and merges the input directly with the output of the subsequent layer. By doing so, the network can learn to focus on the residual mapping, rather than trying to learn the full mapping from input to output.

ResNet-50 consists of 50 layers, including convolutional layers, pooling layers, fully connected layers, and residual connections. The architecture is composed of four blocks, each containing multiple convolutional layers and a shortcut connection. The first block has 64 filters, the second has 128 filters, the third has 256 filters, and the fourth has 512 filters.

The network is trained using backpropagation with a large dataset of labeled images. During training, the weights of the network are adjusted to minimize the error between the predicted output and the true label. Once the network is trained, it can be used for inference on new, unseen images.

Overall, ResNet-50 is a powerful deep learning model that has achieved state-of-the-art performance on various computer vision tasks. Its success can be attributed to the use of residual connections, which allow for deeper and more accurate learning, and the large amount of data used for training.

Code Explanation:

This code implements the architecture of the AlexNet convolutional neural network (CNN) model using the Keras API in TensorFlow. Let's go through the code and explain each section:

1. Importing the necessary libraries:

- `tensorflow` is imported as `tf` for building and training the neural network.

- `Sequential` from `tensorflow.keras.models` is imported to create a sequential model.

- Various layers and utilities are imported from `tensorflow.keras.layers`.

2. Instantiating the model:

- `AlexNet = Sequential()` creates an instance of the sequential model.

3. Adding the layers:

- The subsequent lines add the layers to the model in the order specified by the AlexNet architecture.

4. Convolutional layers:

- The code adds five convolutional layers (`Conv2D`) with different configurations of filters, kernel sizes, strides, and padding. After each convolutional layer, `BatchNormalization` is applied for normalization, followed by the ReLU activation function using `Activation('relu')`.

- Between the convolutional layers, max pooling (`MaxPooling2D`) is applied to reduce spatial dimensions.

5. Flattening:

- After the last convolutional layer, the model is flattened using `Flatten()` to transform the 4D tensor into a 2D tensor for the fully connected layers.

6. Fully connected layers:

- The code adds three fully connected layers (`Dense`) with different numbers of units. Each fully connected layer is followed by batch normalization, ReLU activation, and dropout (`Dropout`) for regularization to prevent overfitting.

7. Output layer:

- The final fully connected layer consists of 10 units, representing the number of classes in the classification task. The activation function used is softmax to obtain class probabilities.

8. Compilation:

- The model is compiled using `AlexNet.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])`. The optimizer is set to Adam, the loss function is categorical cross-entropy (suitable for multi-class classification), and the metric to evaluate the model is accuracy.

9. Summary:

- `AlexNet` is printed to display a summary of the model's architecture, including the layer types, output shapes, and the number of trainable parameters.

Overall, this code builds an AlexNet model architecture with convolutional and fully connected layers, applying batch normalization, dropout, and activation functions to create a powerful image classification model.

Formatted Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications.resnet50 import ResNet50

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Preprocess the data by scaling to values between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Define the ResNet50 base model
base_model = ResNet50(include_top=False, input_shape=(32, 32, 3),
pooling='avg')

# Add a dense layer and a classification layer on top of the base
model
x = base_model.output
x = Dense(256, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# Freeze the base model layers so they are not trained during the
first training phase
for layer in base_model.layers:
    layer.trainable = False

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

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
```

```
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32,
validation_data=(x_test,y_test))
```

Conclusion:

The utilization of the ResNet50 model as the base model, combined with additional layers for fine-tuning or transfer learning, enables powerful image classification capabilities. By freezing the base model's layers and adding custom layers on top, we can adapt the model to specific datasets and achieve accurate predictions.