DSAA 5002 - Data Mining and Knowledge Discovery in Data Science

Final Exam Report – Q2 Weather Recognition

50015940 Jiaxiang Gao

1. Design of data loader

I. Data Organization and Standardization:

The script begins by structuring the data into subdirectories named according to different weather conditions (Cloudy, Foggy, Rainy, Snowy, Sunny). This structure is integral for the automatic labeling of data during the loading process, as these folder names are used as class labels.

Each image file is sorted into an appropriate subfolder based on its filename. This organization ensures that the data is systematically arranged, facilitating easier label assignment by the data loader.

```
# Data directory
data_dir = '../Data_Q2/train_data'

# List of folder names corresponding to different weather conditions
folders = ['Cloudy', 'Foggy', 'Rainy', 'Snowy', 'Sunny']

# Create directories for each weather condition if they don't exist
for folder_name in folders:
    folder_path = os.path.join(data_dir, folder_name)
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

# Move files to corresponding weather condition folders
for filename in os.listdir(data_dir):
    # Check if the item in the data directory is a file
    if os.path.isfile(os.path.join(data_dir, filename)):
        # Loop through each folder name to find a matching condition in the filename
        for folder_name in folders:
            # If the folder name is part of the filename
            if folder_name in filename:
                # Define the source and destination paths for the file
                source_path = os.path.join(data_dir, filename)
                destination_path = os.path.join(data_dir, folder_name, filename)
                # Move the file from the source to the destination path
                shutil.move(source_path, destination_path)

在 2023.12.15 02:10:56 于 8ms内执行
```

II. Image Preprocessing and Rescaling:

An ImageDataGenerator is utilized for image preprocessing, with rescaling set to 1./255. This step is crucial for standardizing the input data. Rescaling the images to a range of 0-1 helps in normalizing the pixel values, making it easier for the model to process and learn from the data.

```
# Initialize ImageDataGenerator with rescaling
datagen = ImageDataGenerator(rescale=1./255)
```

III. Data Loader Configuration:

The flow_from_directory method of ImageDataGenerator is employed to load images from the organized directories. This method is beneficial as it automatically assigns labels based on the directory structure and handles image loading efficiently.

```
# Image size and batch size settings
image_size = (224, 224)
batch_size = 32
```

```
# Load images from directory
train_generator = datagen.flow_from_directory(
    data_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical')
```

The target size for each image is set to 224x224 pixels, ensuring uniformity in input dimensions. Standardizing the size of input images is important for consistent processing by the neural network.

A batch size of 32 is chosen, which determines the number of images processed at a time during training. This size is a balance between computational efficiency and memory constraints.

IV. Label Assignment and Categorization:

The class mode is set to 'categorical', which means the labels are one-hot encoded. This encoding is a standard approach for multi-class classification problems.

V. Shuffling and Additional Considerations:

Although not explicitly mentioned in the code, the flow_from_directory method typically shuffles the data by default for each epoch, which helps in reducing model overfitting and ensuring that the model generalizes well.

2. Model introduction

The defined model is a custom implementation of the VGG16 architecture, a widely recognized convolutional neural network (CNN) used for image classification tasks. Here's a brief introduction to this model:

```
def build_vgg16(input_shape=(224, 224, 3), num_classes=5):
    # Define the input
    input_tensor = Input(shape=input_shape)

    # First block
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_tensor)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)

    # Second block
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)

    # Third block
    x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)

    # Fourth block
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)

    # Fifth block
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2))(x)

    # Flatten and Dense layers
    x = Flatten()(x)
    x = Dense(4096, activation='relu')(x)
    x = Dropout(0.5)(x)
    x = Dense(4096, activation='relu')(x)
    x = Dropout(0.5)(x)
    x = Dense(num_classes, activation='softmax')(x)
```

VGG16 is known for its simplicity, using only 3x3 convolutional layers stacked on top of each other in increasing depth. This specific implementation has been adapted for a classification task with num_classes categories.

The model accepts input images of size 224x224 pixels with 3 channels (RGB).

Architecture:

The model consists of five main blocks of convolutional layers. Each block contains 2-3 convolutional layers with 'relu' activation and 'same' padding, followed by a max-pooling layer. The number of filters in the convolutional layers starts at 64 and doubles with each block, reaching up to 512 in the fourth and fifth blocks.

After the convolutional blocks, the model includes a flattening step to transform the 2D features into a 1D vector.

This is followed by three dense (fully connected) layers. The first two dense layers have 4096 units each and 'relu' activation, interspersed with dropout layers (with a dropout rate of 0.5) to reduce overfitting.

The final dense layer has a number of units equal to num_classes and uses a 'softmax' activation function for multi-class classification.

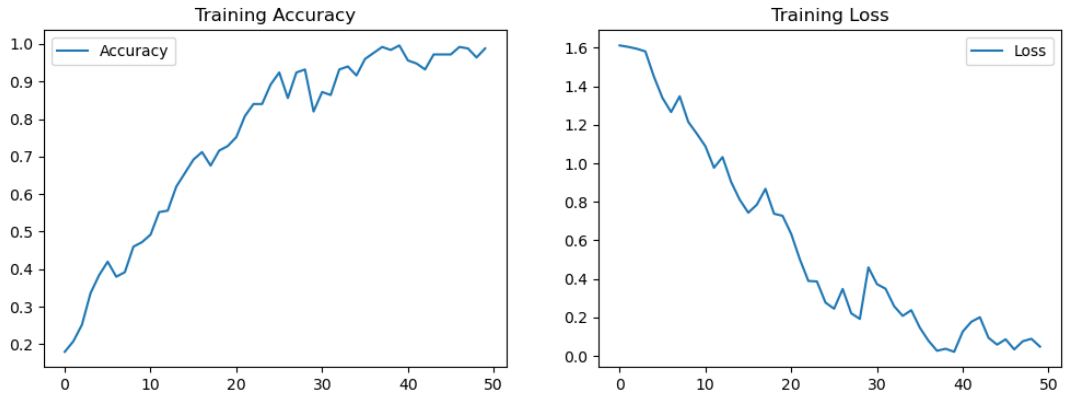
The output is a probability distribution across the num_classes categories, indicating the likelihood of the input image belonging to each category.

3. Training process

```
# Train the model
history = model.fit(train_generator, epochs=50)
```

在 2023.12.15 02:50:50 于 39m 46s 842ms内执行

```
8/8 [==============================] - 48s 6s/step - loss: 0.2007 - accuracy: 0.9320
Epoch 44/50
8/8 [==============================] - 52s 6s/step - loss: 0.0942 - accuracy: 0.9720
Epoch 45/50
8/8 [==============================] - 50s 6s/step - loss: 0.0585 - accuracy: 0.9720
Epoch 46/50
8/8 [==============================] - 49s 6s/step - loss: 0.0862 - accuracy: 0.9720
Epoch 47/50
8/8 [==============================] - 47s 6s/step - loss: 0.0336 - accuracy: 0.9920
Epoch 48/50
8/8 [==============================] - 47s 6s/step - loss: 0.0762 - accuracy: 0.9880
Epoch 49/50
8/8 [==============================] - 46s 6s/step - loss: 0.0889 - accuracy: 0.9640
Epoch 50/50
8/8 [==============================] - 48s 6s/step - loss: 0.0482 - accuracy: 0.9880
```



4. Accuracy on the training set

Accuracy on training set: 0.9919999837875366

```
# Evaluate the model on the training set
train_loss, train_accuracy = model.evaluate(train_generator)
print("Accuracy on training set:", train_accuracy)
```

在 2023.12.15 02:51:03 于 13s 529ms内执行

```
8/8 [==============================] - 13s 2s/step - loss: 0.0312 - accuracy: 0.9920
Accuracy on training set: 0.9919999837875366
```