Experiment-4: Cat Vs Dog

Aim: To implement a Convolution Neural Network (CNN) for dog/cat classification problem using keras.

Run the code to unzip the dataset.zip file after uploding

```
#@title Run the code to unzip the dataset.zip file after uploding
import zipfile
import io
# Replace 'your_zip_file.zip' with the name of your uploaded zip file
with zipfile.ZipFile('dataset.zip', 'r') as zip_ref:
    zip_ref.extractall()
```

Complete code

```
#@title Complete code
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, BatchNormalization, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

# Constants
TRAIN_DIR = "dataset/train"
TEST_DIR = "dataset/test"
IMAGE_SIZE = (128, 128)
BATCH_SIZE = 2
IMAGE_CHANNELS=3

# Create ImageDataGenerator
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
)

test_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
)

train_generator = train_datagen.flow_from_directory(
    TRAIN_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

test_generator = test_datagen.flow_from_directory(
    TEST_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, IMAGE_CHANNELS)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),
```

```
    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Flatten(),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(1, activation='sigmoid'), # 2 because we have cat and dog classes
])

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

model.summary()

# Train the model
history = model.fit(train_generator, steps_per_epoch=5, epochs=11, validation_data=test_generator, validation_steps=2)

# Retrieve training and validation accuracy
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

# Plot training and validation accuracy
plt.plot(train_accuracy, label='Train Accuracy')
plt.plot(val_accuracy, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator)

print("Test Accuracy:", test_accuracy)
print("Test Loss:", test_loss)
```

Code Explanation

Importing Nessary Library and Module

```
#@title Importing Nessary Library and Module
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, BatchNormalization, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

Constants

```
#@title Constants
TRAIN_DIR = "dataset/train"
TEST_DIR = "dataset/test"
IMAGE_SIZE = (128, 128)
BATCH_SIZE = 2
IMAGE_CHANNELS=3
```

Create ImageDataGenerator for Data Augmentation

```
#@title Create ImageDataGenerator for Data Augmentation
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
)
```

The provided code creates an instance of `ImageDataGenerator` in TensorFlow/Keras for data augmentation. Let's break down each parameter:

1. `rescale=1./255` : This parameter scales the pixel values of the images to the range [0, 1]. It's a common practice to rescale pixel values to facilitate model training.

2. `rotation_range=40` : Randomly rotates the images by degrees within the range [-40, 40]. This augmentation helps the model to generalize better by learning from rotated versions of the images.
3. `width_shift_range=0.2` : Randomly shifts the width of the images by a fraction of their total width. Here, it's set to 0.2, meaning the images can be horizontally shifted by up to 20% of their width. This augmentation helps the model learn robustness to changes in object position within the image.
4. `height_shift_range=0.2` : Similar to `width_shift_range` , but it applies vertical shifts to the images.
5. `shear_range=0.2` : Applies shear transformations to the images. Shear transformations slant the shapes of objects within the images. This parameter defines the range within which the shear angle will be randomly selected.
6. `zoom_range=0.2` : Randomly zooms into or out of the images. A value of 0.2 means the images can be zoomed in or out by up to 20%. This augmentation helps the model learn to recognize objects at different scales.
7. `horizontal_flip=True` : Randomly flips images horizontally. This augmentation is useful for tasks where horizontal orientation doesn't affect the semantics of the image, such as object recognition.

These augmentation techniques help in artificially increasing the diversity of the training dataset, which can improve the model's ability to generalize to unseen data and reduce overfitting.

Create Train Data Generator

```
#@title Create Train Data Generator
train_generator = train_datagen.flow_from_directory(
    TRAIN_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)
```

This code snippet creates a generator for yielding augmented training data from a directory using the `train_datagen` object created earlier. Let's break down the parameters used in the `flow_from_directory` method:

1. `TRAIN_DIR` : The directory path containing the training images.
2. `target_size=IMAGE_SIZE` : Specifies the dimensions to which all images will be resized during data loading. Here, it's set to `(128, 128)` , meaning all images will be resized to have a width and height of 128 pixels.
3. `batch_size=BATCH_SIZE` : Specifies the batch size of the data generator. It determines the number of samples in each batch of data yielded by the generator during training. In this case, `BATCH_SIZE` is used.
4. `class_mode='binary'` : Specifies the type of label arrays returned by the generator. In a binary classification problem, where there are only two classes (e.g., cat and dog), this parameter is set to `'binary'` . Each image in the directory is assumed to belong to one of the two classes, and the generator will yield binary labels (0 or 1) accordingly.

Overall, this code sets up a generator to load training images from the specified directory, apply data augmentation transformations defined in `train_datagen` , resize them to the specified target size, and yield batches of augmented images along with their corresponding binary labels during model training.

Create Test Data Generator

```
#@title Create Test Data Generator
test_generator = test_datagen.flow_from_directory(
    TEST_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)
```

This code snippet creates a data generator for the test dataset using the `flow_from_directory` method provided by the `ImageDataGenerator` class in TensorFlow/Keras. Here's an explanation of each parameter:

1. `TEST_DIR` : This is the path to the directory containing the test images.
2. `target_size=IMAGE_SIZE` : Specifies the dimensions to which all images will be resized during preprocessing. `IMAGE_SIZE` is a tuple containing the target height and width of the images.
3. `batch_size=BATCH_SIZE` : Determines the number of images processed in each batch during testing. `BATCH_SIZE` specifies the size of each batch.
4. `class_mode='binary'` : Specifies the type of label array generated by the generator. In this case, since it's set to `'binary'` , the generator will return 1D numpy arrays containing binary labels (0 or 1) indicating the class of the images.

Similar to the training data generator, the `flow_from_directory` method generates batches of data based on the images found in the specified directory. It automatically infers the class labels from the directory structure, where each subdirectory is considered to contain images belonging to a particular class.

Define Convolutional Neural Network Architecture

```
#@title Define Convolutional Neural Network Architecture
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, IMAGE_CHANNELS)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    #Dropout(0.25),

    Flatten(),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(1, activation='sigmoid'), # 2 because we have cat and dog classes
])
```

This code defines a Convolutional Neural Network (CNN) architecture using the Sequential model API in Keras. Here's an explanation of each layer:

1. **Conv2D**: The first convolutional layer with 32 filters, each having a 3x3 kernel size. It uses the ReLU activation function and expects input images of size 128x128 pixels with 3 color channels (RGB).
2. **BatchNormalization**: Normalizes the activations of the previous layer.
3. **MaxPooling2D**: Performs max pooling with a 2x2 pool size to reduce the spatial dimensions of the feature maps.
4. **Conv2D**: The second convolutional layer with 64 filters, followed by batch normalization and max pooling.
5. **Conv2D**: The third convolutional layer with 128 filters, followed by batch normalization and max pooling.
6. **Flatten**: Flattens the output of the convolutional layers into a 1D array to be fed into the dense layers.
7. **Dense**: Fully connected layer with 512 neurons and ReLU activation function.
8. **BatchNormalization**: Normalizes the activations of the previous dense layer.
9. **Dropout**: Regularization layer that randomly sets a fraction of input units to 0 during training to prevent overfitting.
10. **Dense**: Output layer with a single neuron and sigmoid activation function, which is suitable for binary classification tasks like the one at hand. The output value will be in the range [0, 1], representing the probability of the input image belonging to one of the classes (e.g., cat or dog).

This model is designed for binary classification, where the goal is to classify images into one of two categories.

Compile the Model

```
#@title Compile the Model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

This code compiles the defined model with the following configurations:

- **Loss Function**: Binary cross-entropy. It's commonly used for binary classification problems.
- **Optimizer**: Adam optimizer. It's an adaptive learning rate optimization algorithm that's well-suited for most problems.
- **Metrics**: Accuracy. It measures the fraction of correctly classified images, which is a common metric for classification tasks.

Print Model Summary

```
#@title Print Model Summary
model.summary()
```

This code prints a summary of the defined model, including:

- The layer type, output shape, and number of parameters for each layer.
- The total number of parameters in the model.
- The output shape of each layer, which helps in understanding the flow of data through the network.

Train the Model

```
#@title Train the Model
#history = model.fit(train_generator, steps_per_epoch=5, epochs=11, validation_data=test_generator, validation_steps=2)

# Train the model with Early Stopping and Reduce Learning Rate on Plateau callbacks
history = model.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=30,
    validation_data=test_generator,
    validation_steps=len(test_generator),
    callbacks=[
        tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',  # Monitor validation loss for early stopping
            patience=5,          # Stop training if no improvement in validation loss for 5 epochs
            restore_best_weights=True  # Restore weights from the epoch with the best validation loss
        ),
        tf.keras.callbacks.ReduceLROnPlateau(
            monitor='val_loss',  # Monitor validation loss to adjust learning rate
            factor=0.2,          # Reduce learning rate by 20% when validation loss doesn't improve
            patience=3,          # Wait for 3 epochs before reducing learning rate
            min_lr=0.0001        # Lower bound on the learning rate
        )
    ]
)
```

Line-2: This code trains the model using the specified training generator (`train_generator`) for a total of 11 epochs, with 5 steps per epoch. It also uses the validation generator (`test_generator`) for validation during training, with 2 validation steps per epoch. The training history is stored in the `history` variable.

This code trains the model with the specified data generators and parameters for 30 epochs. Two callbacks are used:

1. `EarlyStopping`: Monitors the validation loss and stops training if there is no improvement for 5 epochs (`patience`). It restores the best weights (`restore_best_weights`) observed during training.
2. `ReduceLROnPlateau`: Monitors the validation loss and reduces the learning rate by a factor of 0.2 (`factor`) if the validation loss does not improve for 3 epochs (`patience`). The minimum learning rate is set to 0.0001 (`min_lr`).

Retrieve Training and Validation Accuracy

```
#@title Retrieve Training and Validation Accuracy
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
```

This code retrieves the training accuracy and validation accuracy from the training history (`history`). The training accuracy is stored in the `train_accuracy` variable, and the validation accuracy is stored in the `val_accuracy` variable. These values can be used for further analysis or visualization.

Plot Training and Validation Accuracy

```
#@title Plot Training and Validation Accuracy
import matplotlib.pyplot as plt

plt.plot(train_accuracy, label='Train Accuracy')
plt.plot(val_accuracy, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.show()
```

This code plots the training and validation accuracy over epochs using Matplotlib. It visualizes how the accuracy changes over each epoch during the training process. The x-axis represents the epoch number, and the y-axis represents the accuracy. The plot displays both the training accuracy (blue line) and the validation accuracy (orange line).

Evaluate the model on the test set

```
#@title Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator)

print("Test Accuracy:", test_accuracy)
print("Test Loss:", test_loss)
```

This code evaluates the trained model on the test set using the `evaluate` method. It calculates the test loss and accuracy based on the test data provided by `test_generator`. Finally, it prints out the test accuracy and test loss.