Ex-6: Building a Convolutional Neural Network (CNN) for Dog and Cat Image Classification

Objective:

To design, train, and evaluate a Convolutional Neural Network (CNN) for classifying images of dogs and cats using TensorFlow and Keras.

Requirements:

- Python (>=3.7)
- TensorFlow and Keras
- NumPy and Matplotlib
- OpenCV or PIL (for image processing)
- Google Colab / Jupyter Notebook / Local Machine with GPU support
- Dataset: Dogs vs. Cats dataset (from Kaggle or any standard source)

Step-by-Step Explanation

Step 1: Import Required Libraries

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
import os
```

- tensorflow and keras are used to build and train the CNN.
- layers module helps in defining different layers in the CNN model.
- numpy is used for numerical computations.
- matplotlib.pyplot is used for visualizing training progress.
- os is used for file and directory management.

Step 2: Load and Preprocess the Dataset

- Download the Dogs vs. Cats dataset from Kaggle.
- Extract it into separate folders: train/dogs/, train/cats/, validation/dogs/, validation/cats/.

- Use ImageDataGenerator to preprocess images and apply data augmentation.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_dir = "dataset/train"
validation_dir = "dataset/validation"
datagen = ImageDataGenerator(
    rescale=1.0/255.0,  # Normalize pixel values between 0 and 1
    validation_split=0.2,  # 20% of data for validation
    rotation_range=20,  # Rotate images up to 20 degrees
    width_shift_range=0.2,  # Shift images horizontally
    height_shift_range=0.2,  # Shift images vertically
    shear_range=0.2,  # Shear transformations
    zoom_range=0.2,  # Zoom transformations
    horizontal_flip=True  # Flip images horizontally
)
train_generator = datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary',
    subset='training'  # Training subset
)
validation_generator = datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary',
    subset='validation'  # Validation subset
)
```

- ImageDataGenerator applies transformations to augment training data and improve generalization.
- rescale=1.0/255.0 normalizes pixel values.
- flow_from_directory loads images from directories and applies preprocessing.

Step 3: Build the CNN Model

```
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(150,150,3)),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(128, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])
```

- The model has **three convolutional layers** with ReLU activation.
- MaxPooling2D(2,2) reduces dimensionality and speeds up computation.
- Flatten() converts 3D feature maps into a 1D vector.
- Dense(512, activation='relu') is a fully connected layer for learning patterns.
- Dropout(0.5) prevents overfitting.
- Dense(1, activation='sigmoid') outputs a probability score for binary classification.

Step 4: Compile and Train the Model

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
history = model.fit(
```

```
    train_generator,
    epochs=20,
    validation_data=validation_generator
)
```

- The model uses the **Adam optimizer** for efficient learning.
- binary_crossentropy is used for binary classification.
- The model is trained for **20 epochs**.

Step 5: Evaluate the Model

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

- This plots the training and validation accuracy to analyze the model's performance.

Step 6: Test the Model on New Images

```
from tensorflow.keras.preprocessing import image
import numpy as np
def predict_image(img_path):
    img = image.load_img(img_path, target_size=(150, 150))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0) / 255.0
    prediction = model.predict(img_array)
    print("Dog" if prediction[0] > 0.5 else "Cat")
```

```
predict_image("test/dog.jpg")

predict_image("test/cat.jpg")
```

- image.load_img() loads an image and resizes it.
- image.img_to_array() converts the image into a NumPy array.
- np.expand_dims() reshapes the array to match the model input.
- The model predicts the class (Dog or Cat).

Step 7: Save and Load the Model

```
model.save("dog_cat_classifier.h5")
```

Load the model

```
loaded_model = keras.models.load_model("dog_cat_classifier.h5")
```

- Saves the trained model to an .h5 file for future use.
- The saved model can be reloaded using load_model().

Step 8: Transfer Learning for Improved Accuracy

```
base_model = tf.keras.applications.MobileNetV2(input_shape=(150,150,3),
include_top=False, weights='imagenet')

base_model.trainable = False  # Freeze base model layers

model = keras.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])

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

model.fit(train_generator, epochs=10, validation_data=validation_generator)
```

- Uses **MobileNetV2**, a pre-trained model, for better accuracy.
- base_model.trainable = False ensures the pre-trained layers are not modified.

Convolutional Neural Network (CNN) - A Detailed Explanation with Examples

---

1. Introduction to CNN

A **Convolutional Neural Network (CNN)** is a deep learning algorithm that is particularly effective for image classification, object detection, and other computer vision tasks. Unlike traditional neural networks, CNNs automatically learn spatial features from images by applying convolutional layers.

CNNs are inspired by the human **visual cortex**, where neurons respond to specific regions of an image.

---

2. Why CNNs Instead of Traditional Neural Networks?

Traditional **Feedforward Neural Networks (FNNs)** treat images as 1D vectors, ignoring spatial relationships. This leads to:

- A high number of parameters.
- Inefficient feature extraction.
- Poor performance on image-related tasks.

CNNs solve these issues by using convolutional layers to automatically detect important features like edges, textures, and objects.

---

3. CNN Architecture

A typical CNN consists of several layers:

(a) Input Layer

- Accepts input images (e.g., 150×150 RGB image).
- Preprocessed (normalized, resized, etc.) before feeding into the model.

(b) Convolutional Layer

- Applies **filters (kernels)** to extract features like edges, textures, and shapes.
- Uses a mathematical operation called **convolution**.
- Each filter detects a specific pattern in the image.

👉 **Example:** For a **3×3 filter**, convolution is performed as:

New Pixel Value=(Filter∗Region of Image)+Bias

(c) Activation Function (ReLU)

- **ReLU (Rectified Linear Unit)** is used to introduce non-linearity.
- It replaces negative values with zero:

$$f(x)=\max(0,x)$$

👉 **Why ReLU?**

- Prevents the vanishing gradient problem.
- Faster convergence during training.

(d) Pooling Layer

- Reduces the size of feature maps to make the network efficient.
- Common pooling types:
  - **Max Pooling:** Selects the maximum value from a region.
  - **Average Pooling:** Computes the average of values.

👉 **Example:** If a **2×2 max pooling filter** is applied:

1	**3**	**2**	**4**
5	6	8	2
9	7	3	5
1	2	4	8

After applying **Max Pooling (2×2)**, the new feature map is:

6	**8**
9	8

(e) Fully Connected Layer (FC Layer)

- Flattens the pooled feature maps into a **1D vector**.
- Passes the vector into a **fully connected neural network** for final classification.

(f) Output Layer

- Uses **Softmax** (multi-class) or **Sigmoid** (binary classification) activation.
- Predicts final class probabilities.

---

4. Example: CNN for Handwritten Digit Classification (MNIST Dataset)

Step 1: Import Required Libraries

pythoncode

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
```

Step 2: Load MNIST Dataset

pythoncode

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values (0-255) → (0-1)
x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape data for CNN (Adding channel dimension)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

Step 3: Build CNN Model

pythoncode

```
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),

    layers.Flatten(),
```

```
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Step 4: Compile and Train Model

pythoncode

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

Step 5: Evaluate Model

pythoncode

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test Accuracy:", test_acc)
```

---

5. Real-World Applications of CNNs

CNNs are widely used in:

1. **Image Classification:** Cat vs. Dog classification, MNIST digits.
2. **Object Detection:** Self-driving cars (detecting pedestrians, traffic signs).
3. **Medical Image Analysis:** Detecting tumors in MRI scans.
4. **Facial Recognition:** Used in security systems.
5. **Autonomous Robots:** Scene understanding and navigation.

---

6. Conclusion

- CNNs are powerful for extracting spatial features from images.
- The combination of **convolution, pooling, and fully connected layers** makes CNNs efficient.
- CNNs **outperform traditional machine learning models** for image-related tasks.

Here are a few areas cover in detail:

1. **Mathematical Explanation of Convolution** – How filters extract features.
2. **Feature Map Visualization** – Understanding how CNNs "see" images.
3. **Backpropagation in CNNs** – How CNNs learn through weight updates.
4. **Transfer Learning with CNNs** – Using pre-trained models for better accuracy.
5. **Building a CNN from Scratch** – Full implementation with a real dataset (Dogs vs. Cats, CIFAR-10, etc.).
6. **Optimizing CNNs** – Hyperparameter tuning, dropout, batch normalization, etc.

Each of the key CNN concepts in detail with examples. Let's go step by step.

---

1. Mathematical Explanation of Convolution

What is Convolution?

Convolution is the core operation in a Convolutional Neural Network (CNN). It applies a small filter (kernel) to an image to extract important features like edges, textures, and patterns.

How Does Convolution Work?

- A **filter (kernel)** slides over an image.
- The **dot product** of the filter and the corresponding image region is calculated.
- This produces a **feature map** (also called an activation map).

Example

Let's take a **3×3 filter** applied to a **5×5 image**:

Image (5×5 pixel matrix)

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 0 \\ 7 & 8 & 9 & 2 & 1 \\ 1 & 3 & 2 & 0 & 2 \\ 5 & 6 & 1 & 2 & 3 \end{bmatrix}$$

Filter (3×3 kernel)

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Performing Convolution

- Place the **3×3 filter** on the **top-left corner** of the image.
- Compute the dot product (sum of element-wise multiplications).
- Slide the filter across the image to generate the **feature map**.

🔹 **Key takeaway**: Convolution extracts **edges, corners, and textures** that help in feature learning.

---

2. Feature Map Visualization

CNNs learn different **features** at different layers. The **early layers** detect basic edges, while the **deeper layers** recognize complex patterns.

Example: Visualizing Filters in CNN

pythoncode

```
import tensorflow as tf
import matplotlib.pyplot as plt

model = tf.keras.applications.VGG16(weights='imagenet', include_top=False)
filters, biases = model.layers[1].get_weights()

# Display first 6 filters
fig, axes = plt.subplots(1, 6, figsize=(15, 5))
for i in range(6):
    axes[i].imshow(filters[:, :, :, i], cmap='gray')
    axes[i].axis('off')
plt.show()
```

👉 **This code visualizes the first layer's filters** in a pre-trained VGG16 model.

---

3. Backpropagation in CNNs

How CNNs Learn?

CNNs **update filter weights** using **backpropagation**:

1. Compute **loss** (e.g., cross-entropy).
2. Compute **gradients** of loss w.r.t. weights.
3. Use **gradient descent (e.g., Adam optimizer)** to adjust weights.

Key Differences from Traditional Neural Networks

✅ CNNs **share weights** (fewer parameters).
✅ CNNs **preserve spatial structure** (use filters, not fully connected layers).

---

4. Transfer Learning with CNNs

Instead of training from scratch, we **reuse pre-trained models** (like VGG16, ResNet, or MobileNet).

Example: Transfer Learning with MobileNetV2

pythoncode

```
import tensorflow as tf

from tensorflow.keras import layers, models

# Load pre-trained MobileNetV2

base_model = tf.keras.applications.MobileNetV2(input_shape=(150,150,3),
include_top=False, weights='imagenet')

base_model.trainable = False  # Freeze the base model layers

# Add custom layers

model = models.Sequential([

    base_model,

    layers.GlobalAveragePooling2D(),

    layers.Dense(128, activation='relu'),

    layers.Dropout(0.5),

    layers.Dense(1, activation='sigmoid')

])
```

```
# Compile and train
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

- **Why Transfer Learning?**
  - Faster training (uses pre-learned filters).
  - Better accuracy with fewer images.

---

5. Building a CNN from Scratch

Dataset: Cats vs. Dogs Classification

pythoncode

```
import tensorflow as tf
from tensorflow.keras import layers, models
# Define CNN architecture
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(150,150,3)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(128, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])
# Compile Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train Model
history = model.fit(train_generator, epochs=20, validation_data=validation_generator)
```

- **This CNN processes images and classifies them as Dogs or Cats.**

---

6. Optimizing CNNs

Key Optimization Techniques

✅ **Batch Normalization** – Speeds up training, improves stability.
✅ **Dropout Regularization** – Prevents overfitting.
✅ **Data Augmentation** – Increases dataset size artificially.

Example: Data Augmentation

pythoncode

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)


train_generator = datagen.flow_from_directory('dataset/train', target_size=(150,150), batch_size=32, class_mode='binary')
```

- **Why use data augmentation?**
  - **Reduces overfitting.**
  - **Improves generalization.**
