DOCUMENTATION

CONVOLUTIONAL NUERAL NETWORK

A Convolutional Neural Network (CNN) is a specialized type of deep learning model mainly used for image and visual data processing. It works by automatically learning features such as edges, shapes, textures, and objects from images through multiple layers. The core layers of a CNN include convolution layers, which apply filters to extract important features; activation functions like ReLU, which introduce non-linearity; pooling layers, which reduce spatial size and computation while retaining key information; and fully connected layers, which perform final classification or prediction. Unlike traditional neural networks, CNNs use local connectivity and shared weights, making them highly efficient for handling high-dimensional image data. CNNs are widely used in applications such as image classification, face recognition, medical image analysis, object detection, and autonomous vehicles due to their accuracy and ability to learn complex visual patterns automatically.

Ignoring Warning Messages in Python

```
import warnings
warnings.filterwarnings("ignore")
```

This code is used to suppress warning messages in Python so that they do not appear in the output while running a program. The first line imports the built-in `warnings` module, which is responsible for handling warning messages generated by Python or external libraries. The second line, `warnings.filterwarnings("ignore")`, tells Python to ignore all warnings globally, meaning any warning (such as deprecation or runtime warnings) will not be displayed. This is commonly used in data science and machine learning projects to keep the output clean and focused on results, especially when warnings are repetitive or not critical to the task being performed.

Importing Required Libraries

```
import os
import cv2
import numpy as np
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tensorflow.keras.models import Sequential,load_model
from tensorflow.keras.layers import Conv2D,MaxPool2D,Flatten,Dense
```

This code imports the essential Python libraries needed for building and evaluating a Convolutional Neural Network (CNN) model. The `os` module is used to handle file and directory operations, while `cv2` (OpenCV) helps in reading, resizing, and processing image data. `numpy` is used for numerical operations and array handling, and `seaborn` is mainly used for data visualization, especially for plotting the confusion matrix. From `sklearn`, `LabelEncoder` is used to convert categorical class labels into numeric form, `train_test_split` helps divide the dataset into training and testing sets, and `confusion_matrix` is used to evaluate the model's performance. From `tensorflow.keras`, the `Sequential` and `load_model` modules are used to build and

load deep learning models, while layers like `Conv2D`, `MaxPool2D`, `Flatten`, and `Dense` are used to create the CNN architecture for feature extraction and classification.

Loading and Preprocessing Image Data for a CNN Model

```
data=r"C:\Users\asus\Downloads\Deeplearningdataset\forest_fire\Testing"
```

```
x1,y1=[],[]
for i in os.listdir(data):
    new_dir=os.path.join(data,i)
    if os.path.isdir(new_dir):
        for j in os.listdir(new_dir):
            image_path=os.path.join(new_dir,j)
            if image_path.endswith((".png",".jpg",".jpeg")):
                image_data=cv2.imread(image_path)
                array_resize=cv2.resize(image_data,(100,100))
                array_reshape=array_resize.reshape(100,100,3)
                array_norm=array_reshape/255
                x1.append(array_norm)
                y1.append(i)
x1=np.array(x1)
y1=np.array(y1)
print(x1.shape)
print(y1.shape)
```

```
(68, 100, 100, 3)
(68,)
```

This code is used to load and prepare image data from a folder structure for use in a Convolutional Neural Network (CNN). First, the dataset path is stored in the variable `data`, which points to the testing images directory. Two empty lists, `x1` and `y1`, are created to store image data and their corresponding labels. The code then loops through each subfolder in the main directory, where each subfolder name represents a class label. Inside each subfolder, all image files with `.png`, `.jpg`, or `.jpeg` extensions are read using OpenCV. Each image is resized to `100 × 100` pixels to maintain a consistent input size, reshaped to include three color channels (RGB), and normalized by dividing pixel values by 255 to scale them between 0 and 1. The processed images

are added to x1, and their labels are added to y1. Finally, both lists are converted into NumPy arrays, and their shapes are printed, showing that 68 images of size 100 × 100 × 3 were loaded along with 68 corresponding labels.

Loading and Preprocessing Training and Validation Image Data

```
data1=r"C:\Users\asus\Downloads\Deeplearningdataset\forest_fire\Training and Validation"
```

```
x2,y2=[],[]
for i in os.listdir(data1):
    new_dir=os.path.join(data1,i)
    if os.path.isdir(new_dir):
        for j in os.listdir(new_dir):
            image_path=os.path.join(new_dir,j)
            if image_path.endswith((".png",".jpg",".jpeg")):
                image_data=cv2.imread(image_path)
                array_resize=cv2.resize(image_data,(100,100))
                array_reshape=array_resize.reshape(100,100,3)
                array_norm=array_reshape/255
                x2.append(array_norm)
                y2.append(i)
x2=np.array(x2)
y2=np.array(y2)
print(x2.shape)
print(y2.shape)
```

```
(1912, 100, 100, 3)
(1912,)
```

This code is used to load and preprocess images from the Training and Validation dataset for a Convolutional Neural Network (CNN). The dataset path is stored in data1, which points to the folder containing class-wise image subfolders. Two empty lists, x2 and y2, are created to store processed image data and their corresponding labels. The code loops through each class folder, reads all image files with valid extensions using OpenCV, and resizes each image to 100 × 100 pixels to maintain uniform input size. The images are reshaped to include three color channels (RGB) and normalized by dividing pixel values by 255 to scale them between 0 and 1. Each processed image is added to x2, while the class name is added to y2 as the label. Finally, the lists are converted into NumPy arrays and their shapes are printed, showing that 1912 images of

size 100 × 100 × 3 were successfully loaded along with 1912 corresponding labels, making the data ready for CNN training and validation.

Concatenating NumPy Arrays for Features and Labels

```
x=np.concatenate((x1,x2))
y=np.concatenate((y1,y2))
y
```

```
array(['fire', 'fire', 'fire', ..., 'nofire', 'nofire', 'nofire'],
      dtype='<U6')
```

In the provided code, two pairs of NumPy arrays, x1 & x2 and y1 & y2, are concatenated using `np.concatenate()`. The first line, `x = np.concatenate((x1, x2))`, combines the feature arrays x1 and x2 into a single array x. Similarly, `y = np.concatenate((y1, y2))` merges the label arrays y1 and y2 into one array y. The resulting y array contains string labels such as `'fire'` and `'nofire'`, representing categories for each data point in x. The `dtype='<U6'` indicates that the array elements are Unicode strings of maximum length 6. This approach is commonly used to combine datasets from multiple sources before feeding them into machine learning models for training or evaluation.

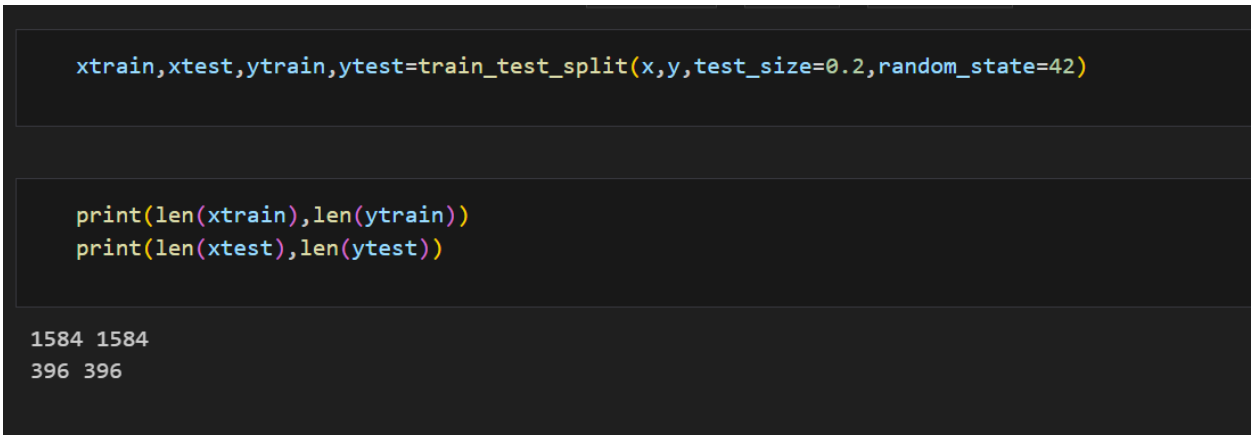
Encoding Categorical Labels with `LabelEncoder`

```
le=LabelEncoder()
y=le.fit_transform(y)
y
```

```
array([0, 0, 0, ..., 1, 1, 1])
```

The code shown uses `LabelEncoder` from the `sklearn.preprocessing` module to convert categorical labels into numeric form. First, an instance of `LabelEncoder` is created using `le = LabelEncoder()`. Then, the `fit_transform` method is applied to the target variable y, which learns the unique categories in y and converts each category into a corresponding integer. The transformed y now contains numeric values (like 0, 1, 2, etc.) instead of the original categorical labels, making it suitable for machine learning algorithms that require numerical input. The final line y simply outputs this encoded array, showing the mapping of the original categories into integers. This process is essential when working with algorithms that cannot directly handle string labels.

Train–Test Split



This code uses `train_test_split()` from `sklearn` to divide the dataset into training data and testing data so that a machine learning model can be trained and then evaluated fairly. Here, `x` represents the input features and `y` represents the target labels. The parameter `test_size=0.2` means 20% of the total data is reserved for testing, while the remaining 80% is used for training. The `random_state=42` ensures that the data is split in the same way every time the code runs, which helps in reproducibility. After splitting, `xtrain` and `ytrain` contain the training inputs and labels, while `xtest` and `ytest` contain the testing inputs and labels. The `print(len(xtrain), len(ytrain))` and `print(len(xtest), len(ytest))` lines confirm that the number of input samples and corresponding labels match in both sets, showing 1584 samples for training and 396 samples for testing, which correctly follows the 80–20 split rule.

CNN Model Architecture

```
model=Sequential()
model.add(Conv2D(32,(3,3),activation="relu",input_shape=(100,100,3)))
model.add(MaxPool2D(2,2))
model.add(Conv2D(64,(3,3),activation="relu"))
model.add(MaxPool2D(2,2))
model.add(Conv2D(128,(3,3),activation="relu"))
model.add(MaxPool2D(2,2))
model.add(Flatten())
model.add(Dense(128,activation="relu"))
model.add(Dense(2,activation="softmax"))
```

This code builds a Convolutional Neural Network (CNN) using Keras' `Sequential` model, which means layers are added one after another in a linear order. The model starts with a Conv2D layer of 32 filters using a 3×3 kernel and the ReLU activation function, designed to extract basic features like edges and textures from input images of size 100×100 with 3 color channels (RGB). A MaxPooling2D layer (2×2) follows to reduce spatial dimensions and computation while keeping important features. This pattern is repeated with 64 and then 128 filters, allowing the network to learn more complex and deeper features as the data flows through the layers. After feature extraction, the Flatten layer converts the 2D feature maps into a 1D vector so it can be fed into fully connected layers. The Dense layer with 128 neurons and ReLU activation learns high-level patterns, and finally, the Dense layer with 2 neurons and softmax activation outputs class probabilities, making this model suitable for a binary image classification problem.

Model Summary

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 98, 98, 32)	896
max_pooling2d (MaxPooling2D)	(None, 49, 49, 32)	0
conv2d_1 (Conv2D)	(None, 47, 47, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 23, 23, 64)	0
conv2d_2 (Conv2D)	(None, 21, 21, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 128)	0
flatten (Flatten)	(None, 12800)	0
dense (Dense)	(None, 128)	1,638,528
dense_1 (Dense)	(None, 2)	258

```
Total params: 1,732,034 (6.61 MB)

Trainable params: 1,732,034 (6.61 MB)

Non-trainable params: 0 (0.00 B)
```

This summary shows the overall parameter count of the CNN and how they are used during training. The Total parameters (1,732,034) represent all weights and biases

present across every layer in the model, which together require about 6.61 MB of memory. Since this model is built entirely from trainable layers like Conv2D and Dense, all parameters are listed as Trainable parameters (1,732,034), meaning they are updated during backpropagation to minimize the loss. The Non-trainable parameters (0) indicate that there are no frozen layers, such as pre-trained layers or fixed weights, in this network. Most of the parameters come from the Dense(128) layer because it is fully connected to the flattened input of 12,800 features, while convolution and pooling layers contribute far fewer parameters. Overall, this breakdown confirms that the model is fully learnable and its complexity is mainly driven by the dense layer at the end.

Compiling the Model

```
model.compile(optimizer="adam",loss="sparse_categorical_crossentropy",metrics=["accuracy"])
```

The `model.compile()` function is used to configure the CNN model before training by defining how it should learn and how its performance will be evaluated. In this code, the Adam optimizer is selected, which efficiently updates model weights by combining the benefits of momentum and adaptive learning rates, leading to faster and more stable convergence. The loss function `sparse_categorical_crossentropy` measures how well the model's predicted class probabilities match the true labels when the target labels are provided as integer values (for example, 0 or 1 instead of one-hot encoded vectors). The `metrics=["accuracy"]` option tells the model to calculate accuracy during training and validation, making it easy to understand how often the model predicts the correct class. Overall, this compilation step prepares the model with the necessary rules to learn from data and track its performance effectively.

Model Fitting

```
model.fit(xtrain,ytrain,batch_size=10,epochs=22,validation_data=(xtest,ytest))
```

```
Epoch 1/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 31s 148ms/step - accuracy: 0.9135 - loss: 0.2481 - val_accuracy: 0.9268 - val_loss: 0.2055
Epoch 2/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 25s 154ms/step - accuracy: 0.9451 - loss: 0.1848 - val_accuracy: 0.9545 - val_loss: 0.1684
Epoch 3/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 31s 194ms/step - accuracy: 0.9476 - loss: 0.1443 - val_accuracy: 0.9470 - val_loss: 0.1482
Epoch 4/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 32s 198ms/step - accuracy: 0.9621 - loss: 0.1219 - val_accuracy: 0.9545 - val_loss: 0.1346
Epoch 5/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 27s 171ms/step - accuracy: 0.9571 - loss: 0.1234 - val_accuracy: 0.9596 - val_loss: 0.1084
Epoch 6/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 29s 178ms/step - accuracy: 0.9640 - loss: 0.1015 - val_accuracy: 0.8889 - val_loss: 0.3095
Epoch 7/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 40s 167ms/step - accuracy: 0.9634 - loss: 0.0952 - val_accuracy: 0.9596 - val_loss: 0.1457
Epoch 8/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 29s 181ms/step - accuracy: 0.9710 - loss: 0.0878 - val_accuracy: 0.9697 - val_loss: 0.1070
Epoch 9/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 26s 165ms/step - accuracy: 0.9754 - loss: 0.0670 - val_accuracy: 0.9747 - val_loss: 0.0969
Epoch 10/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 27s 166ms/step - accuracy: 0.9766 - loss: 0.0695 - val_accuracy: 0.9697 - val_loss: 0.0910
Epoch 11/22
159/159 ━━━━━━━━━━━━━━━━━━━━ 26s 163ms/step - accuracy: 0.9811 - loss: 0.0542 - val_accuracy: 0.9646 - val_loss: 0.1080
```

The `model.fit()` function is used to train the CNN using the training data (`xtrain`, `ytrain`) while evaluating its performance on validation data (`xtest`, `ytest`). In this example, the model is trained for 22 epochs, meaning the entire training dataset is passed through the network 22 times, with a batch size of 10, so the data is processed in small groups to balance learning stability and memory usage. Each epoch displays the number of steps completed, the time taken, and key metrics such as training accuracy and loss, which show how well the model is learning from the training data. Additionally, validation accuracy and validation loss indicate how well the model generalizes to unseen data. As the epochs progress, the decreasing loss values and increasing accuracy demonstrate that the model is improving, while fluctuations in validation metrics help identify learning stability and potential overfitting.

Image Preprocessing for Model Prediction

```
test_data1=r"C:\Users\asus\Downloads\Deeplearningdataset\forest_fire\Testing\fire\abc186.jpg"
```

```
image_data=cv2.imread(test_data1)
array_resize=cv2.resize(image_data,(100,100))
array_reshape=array_resize.reshape(1,100,100,3)
array_norm=array_reshape/255
print(array_norm.shape)
```

```
(1, 100, 100, 3)
```

This code prepares a single image so it can be used as input for the trained CNN model during prediction. First, the variable `test_data1` stores the file path of the image that needs to be tested. The image is read using `cv2.imread()`, which loads it as a NumPy array. Since the CNN model was trained using images of size 100×100, the image is resized to the same dimensions with `cv2.resize()` to maintain consistency. The resized image is then reshaped into `(1, 100, 100, 3)`, where 1 represents the batch size and 3 represents the RGB color channels, making it compatible with the model's expected input format. After reshaping, the pixel values are normalized by dividing by 255, scaling them to a range between `0` and `1` for better model performance. Finally, printing the shape confirms that the image is correctly processed and ready for prediction.

Model Prediction and Class Label Selection

```
    a=model.predict(array_norm)
    a
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 314ms/step

array([[1.000000e+00, 2.848066e-08]], dtype=float32)
```

```
    label=["fire","nofire"]
```

```
    b=np.argmax(a)
    label[b]
```

```
'fire'
```

In this code, the preprocessed image (`array_norm`) is passed into the trained CNN model using `model.predict(array_norm)`, which returns the model's prediction probabilities for each class. The output `[[1.000000e+00, 2.848066e-08]]` shows that the model predicts two values—one for fire and one for no fire—where the first value is almost 1 (very high confidence) and the second is extremely close to 0. This means the model is almost certain the image belongs to the fire class. Next, the list `label = ["fire", "nofire"]` defines the class names in the same order as the model's output neurons. The function `np.argmax(a)` finds the index of the highest probability value in the prediction array, which is `0`. Finally, `label[b]` uses this index to fetch the corresponding class name, resulting in the output "fire", indicating that the model has classified the input image as containing fire.

Batch Prediction on Test Dataset Using the Model

```
ypred=model.predict(xtest)
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step
```

In this code, `ypred = model.predict(xtest)` is used to generate predictions for the entire test dataset at once. Here, `xtest` contains multiple test images that were already preprocessed (resized, reshaped, and normalized) in the same way as the training data. When this line runs, the progress bar `13/13` indicates that the test data is processed in 13 batches, and the time shown (`1s 54ms/step`) represents how long the model takes to process each batch. The output `ypred` is a NumPy array that stores the predicted probability values for each test image and each class, which can later be converted into class labels using functions like `np.argmax()`. This step is mainly used for model evaluation, such as calculating accuracy, confusion matrix, or classification reports on unseen data.

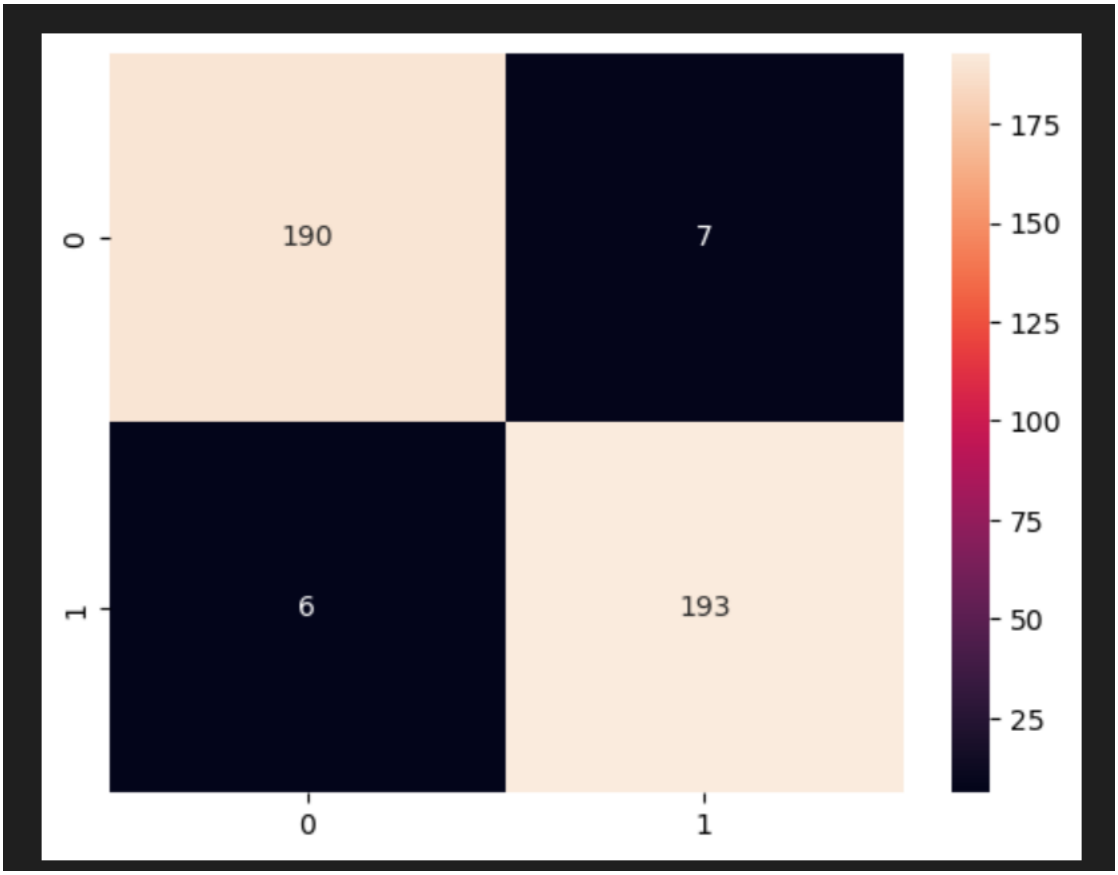
Confusion Matrix Creation and Model Performance Visualization

```
ypredd=[]
for i in ypred:
    ypredd.append(np.argmax(i))
```

```
cm=confusion_matrix(ytest,ypredd)
cm
```

```
array([[190,   7],
       [  6, 193]])
```

```
sns.heatmap(cm,fmt="d",annot=True)
```



In this code, the predicted probabilities stored in ypred are first converted into class labels using a loop and the `np.argmax()` function, where each prediction vector is transformed into a single class index and saved in the list ypredd. This step is necessary because the confusion matrix requires actual class labels, not probability values. Next, `confusion_matrix(ytest, ypredd)` compares the model's predicted labels with the true labels from `ytest` and produces a 2×2 matrix that summarizes the classification results. The output matrix `[[190, 7], [6, 193]]` shows that the model correctly classified 190 images of class 0 and 193 images of class 1, while it misclassified 7 images of class 0 as class 1 and 6 images of class 1 as class 0. Finally, `sns.heatmap()` visually represents this confusion matrix using color intensity and annotated values, making it easier to interpret the model's accuracy and error distribution across both classes.