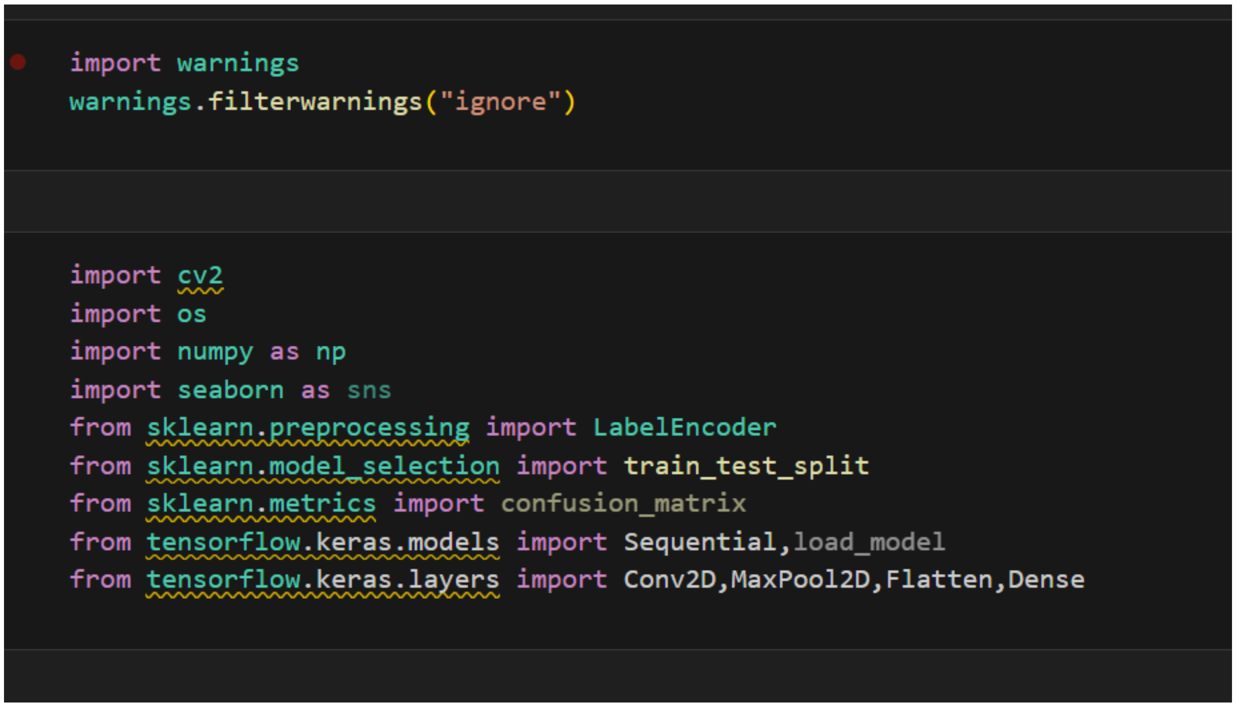
DOCUMENTATION

CONVOLUTIONAL NUERAL NETWORK

A Convolutional Neural Network (CNN) is a type of deep learning model mainly used for image and visual data processing. It works by automatically learning important features from images such as edges, shapes, and textures instead of relying on manual feature extraction. A CNN consists of layers like convolution layers (to extract features using filters), pooling layers (to reduce image size and computation), and fully connected layers (to perform final classification). Activation functions like ReLU add non-linearity, while Softmax is commonly used in the output layer for classification. CNNs are widely used in applications such as image recognition, face detection, medical image analysis, and self-driving cars because they are highly accurate and efficient for handling image data.

Importing Required Libraries

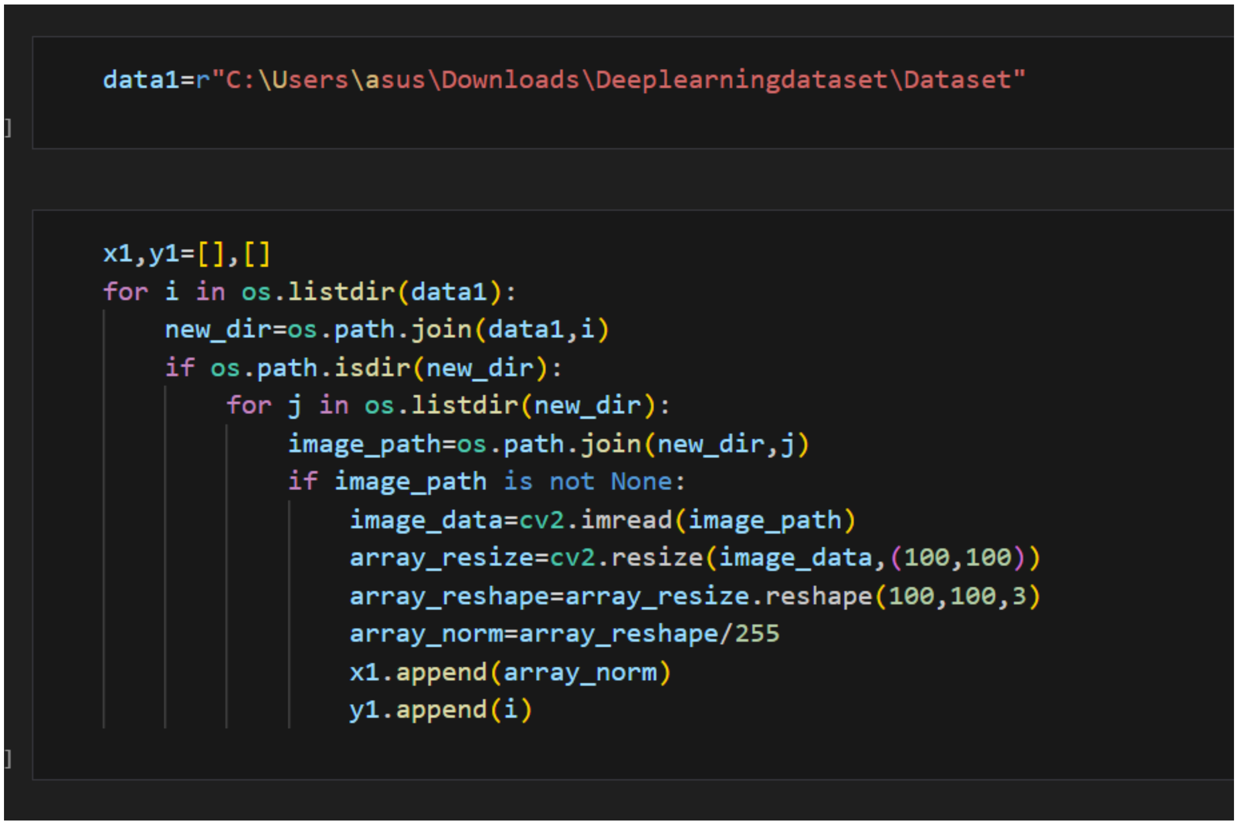


```
import warnings
warnings.filterwarnings("ignore")
```

```
import cv2
import os
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 block imports all the necessary Python libraries required for building and evaluating a Convolutional Neural Network (CNN) model. The `warnings` module is used to suppress unnecessary warning messages to keep the output clean. cv2 (OpenCV) helps in reading and processing images, while os is used to interact with the file system, such as accessing folders and files. numpy is essential for numerical operations and handling image data in array form, and seaborn is mainly used for data visualization. From `sklearn`, `LabelEncoder` is used to convert categorical labels into numerical form, `train_test_split` helps divide the dataset into training and testing sets, and `confusion_matrix` is used to evaluate model performance. From `tensorflow.keras`, `Sequential` is used to build the CNN model layer by layer, `load_model` helps load a saved trained model, and layers like Conv2D, `MaxPool2D`, `Flatten`, and Dense are used to extract image features, reduce dimensions, convert data into a vector, and perform final classification respectively.

Image Dataset Loading and Preprocessing

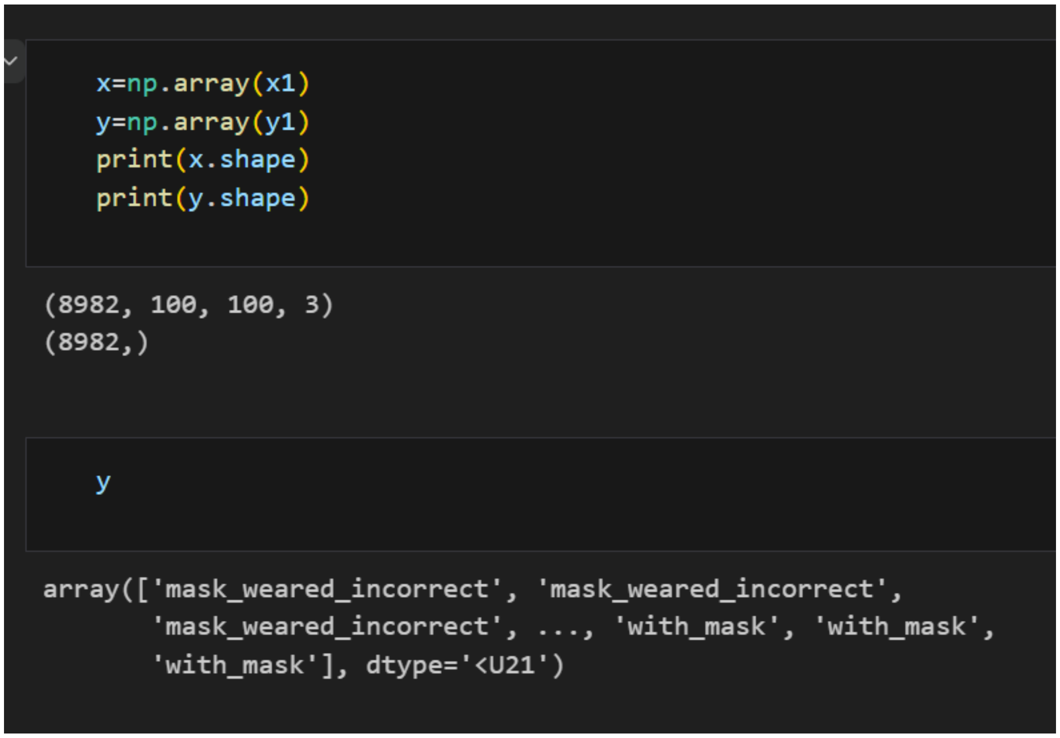


```
data1=r"C:\Users\asus\Downloads\Deeplearningdataset\Dataset"
```

```
x1,y1=[],[]
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 is not None:
                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)
```

This code is used to load image data from a dataset folder and preprocess it for training a CNN model. First, the variable `data1` stores the path of the main dataset directory, where each subfolder represents a class label. Two empty lists, x1 and y1, are created to store image data and their corresponding labels. The outer loop goes through each subfolder (class) in the dataset directory, and `os.path.join` is used to build the full path. The `os.path.isdir` check ensures that only directories are processed. Inside each class folder, the inner loop reads every image file using `cv2.imread`. Each image is resized to a fixed size of 100×100 pixels to maintain uniform input dimensions. The resized image is reshaped to include three color channels (RGB), normalized by dividing pixel values by 255 to scale them between 0 and 1, and then appended to x1. The folder name `i` is appended to y1 as the class label. Overall, this process converts raw images into normalized numerical arrays along with their labels, making them ready for training a CNN model.

Converting Data to NumPy Arrays and Understanding Shapes



```
x=np.array(x1)
y=np.array(y1)
print(x.shape)
print(y.shape)
```

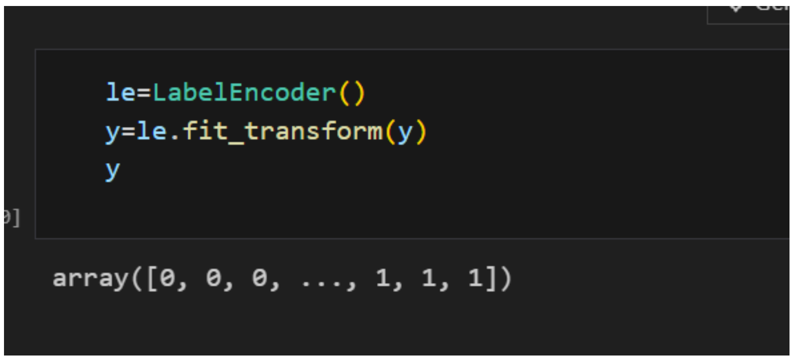
```
(8982, 100, 100, 3)
(8982,)
```

```
y
```

```
array(['mask_weared_incorrect', 'mask_weared_incorrect',
       'mask_weared_incorrect', ..., 'with_mask', 'with_mask',
       'with_mask'], dtype='<U21')
```

This code converts the image data and labels stored in Python lists into NumPy arrays, which are required for efficient computation and model training. The statement `x = np.array(x1)` converts the list of preprocessed images into a NumPy array, while `y = np.array(y1)` converts the list of class labels into a NumPy array. Printing `x.shape` gives `(8982, 100, 100, 3)`, which means there are 8,982 images in the dataset, each of size 100×100 pixels with 3 color channels (RGB). Printing `y.shape` gives `(8982,)`, indicating that there is one label for each image. The output of y shows class names like `'mask_weared_incorrect'` and `'with_mask'`, confirming that the labels are currently in string format. This step ensures that the dataset is properly structured and aligned, with each image correctly mapped to its corresponding class label, ready for further preprocessing such as label encoding and model training.

Encoding Class Labels Using LabelEncoder

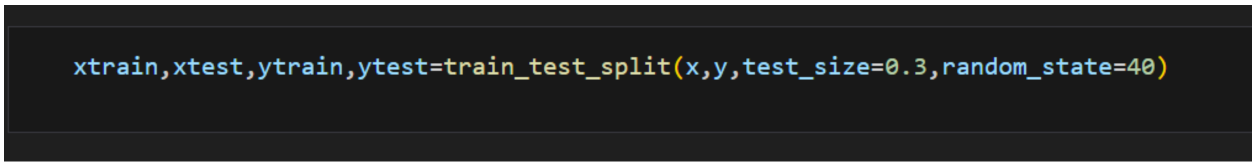


```
le=LabelEncoder()
y=le.fit_transform(y)
y
```

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

This code converts the categorical class labels stored in y into numerical values so that they can be used by a machine learning or deep learning model. First, `LabelEncoder()` creates an encoder object from the `sklearn.preprocessing` module. The method `le.fit_transform(y)` learns all the unique class names in y (such as `with_mask` and `mask_weared_incorrect`) and assigns each one a unique integer value, typically starting from 0. The transformed labels replace the original string labels, which is why the output now appears as an array of 0s and 1s. This step is essential because neural networks cannot process text labels directly and require numerical values for training and prediction.

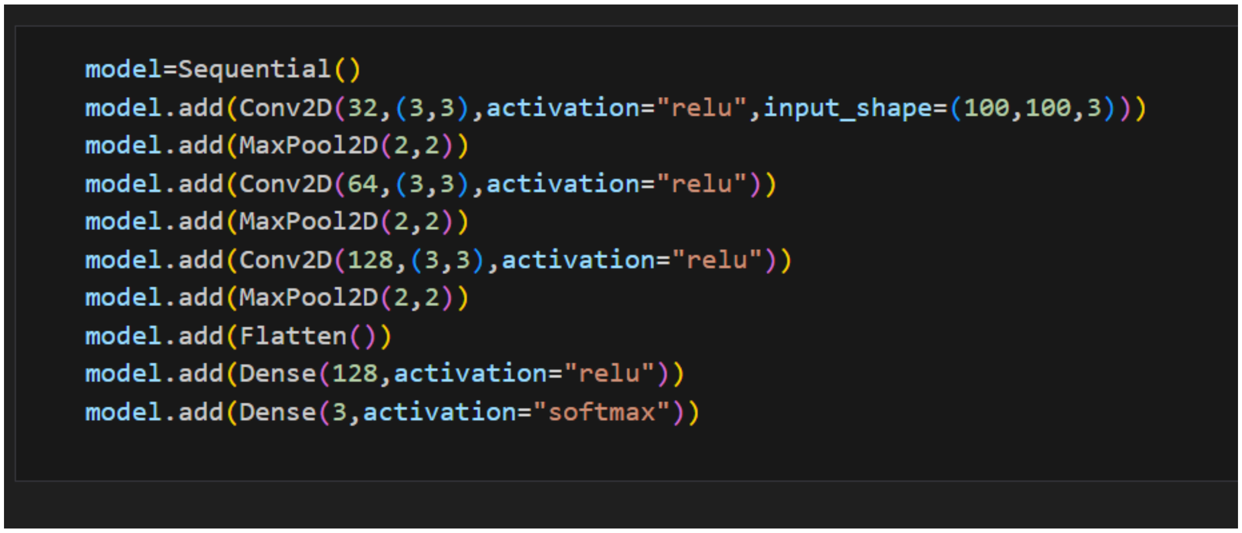
Splitting Data into Training and Testing Sets



```
xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.3,random_state=40)
```

This line of code divides the dataset into training and testing subsets using the `train_test_split` function from `sklearn.model_selection`. Here, x contains the image data and y contains the corresponding encoded labels. The parameter `test_size=0.3` means that 30% of the total data is reserved for testing the model, while the remaining 70% is used for training. The `random_state=40` ensures that the data is split in the same way every time the code is run, which helps in getting consistent and reproducible results. As a result, `xtrain` and `ytrain` are used to train the CNN model, while `xtest` and `ytest` are used to evaluate its performance on unseen data.

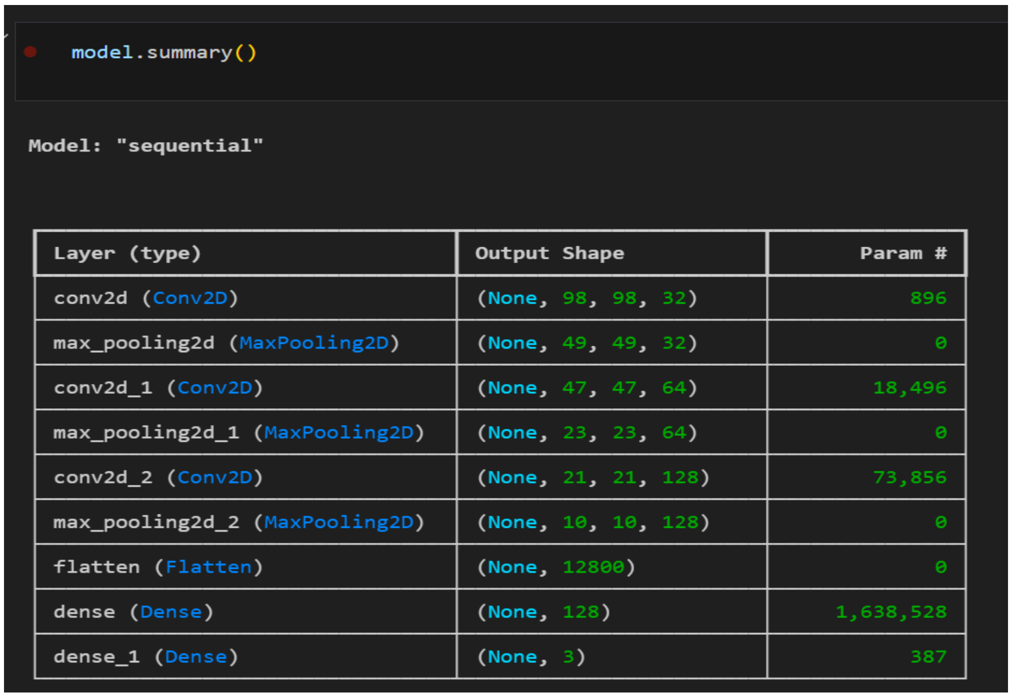
Building the CNN Model Architecture



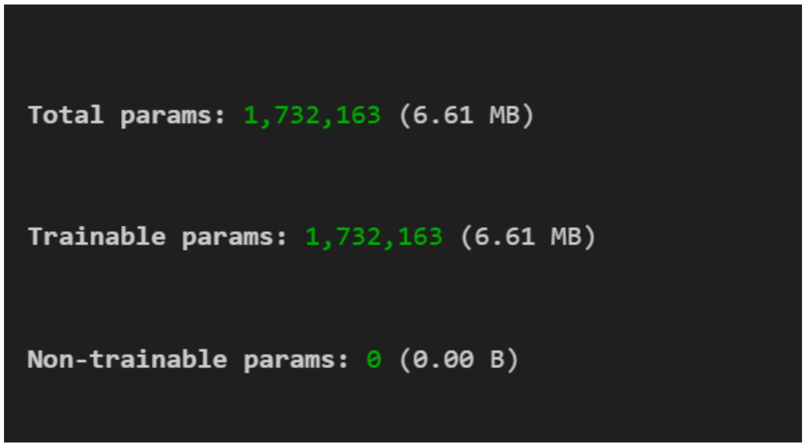
```
model=Sequential()
model.add(Conv2D(32,(3,3),activation="relu",input_shape=(100,100,3)))
model.add(MaxPool2D(2,2))
model.add(Conv2D(64,(3,3),activation="relu"))
model.add(MaxPool2D(2,2))
model.add(Conv2D(128,(3,3),activation="relu"))
model.add(MaxPool2D(2,2))
model.add(Flatten())
model.add(Dense(128,activation="relu"))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

This code defines the architecture of a Convolutional Neural Network (CNN) using the Keras `Sequential` model, which allows layers to be added one after another. The first Conv2D layer uses 32 filters of size 3×3 with the ReLU activation function and an input shape of 100×100×3, meaning the model expects color images of that size. This is followed by a `MaxPool2D` layer with a 2×2 pool size to reduce spatial dimensions and computation. The second and third convolution layers use 64 and 128 filters respectively, each with 3×3 kernels and ReLU activation, followed by max pooling layers to progressively extract more complex features while reducing image size. The `Flatten` layer then converts the 2D feature maps into a 1D vector so it can be fed into fully connected layers. A dense layer with 128 neurons and ReLU activation learns high-level patterns, and the final dense layer with 3 neurons and Softmax activation outputs class probabilities for three different classes. This layered structure helps the model learn from simple features to complex patterns for accurate image classification.

Understanding the Model Summary Output



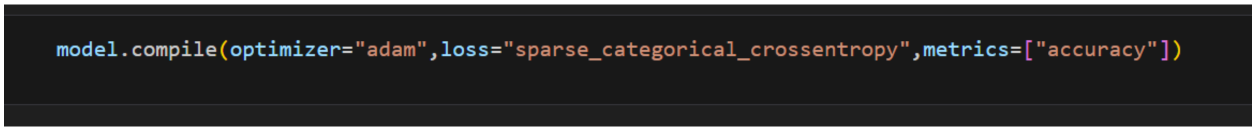
```
model.summary()

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, 3)              │          387 │
└───────────────────────────────┴────────────────────────┴──────────────┘
```



```
Total params: 1,732,163 (6.61 MB)

Trainable params: 1,732,163 (6.61 MB)

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

The `model.summary()` command displays a detailed overview of the CNN architecture, showing each layer, its output shape, and the number of parameters it contains. The output shape starts with None, which represents the batch size that can vary during training. As the data passes through the convolution and max pooling layers, the spatial dimensions (height and width) gradually decrease while the depth (number of filters) increases, indicating feature extraction at different levels. The convolution layers have trainable parameters because they learn filter weights, while max pooling and flatten layers have zero parameters since they only perform operations. After flattening, the data becomes a 1D vector of size 12,800, which leads to a large number of parameters in the first dense layer (1,638,528) because every neuron connects to all inputs. The final dense layer outputs 3 values corresponding to the three classes. The total number of parameters is 1,732,163, all of which are trainable, meaning the model learns all weights during training with no fixed (non-trainable) parameters.

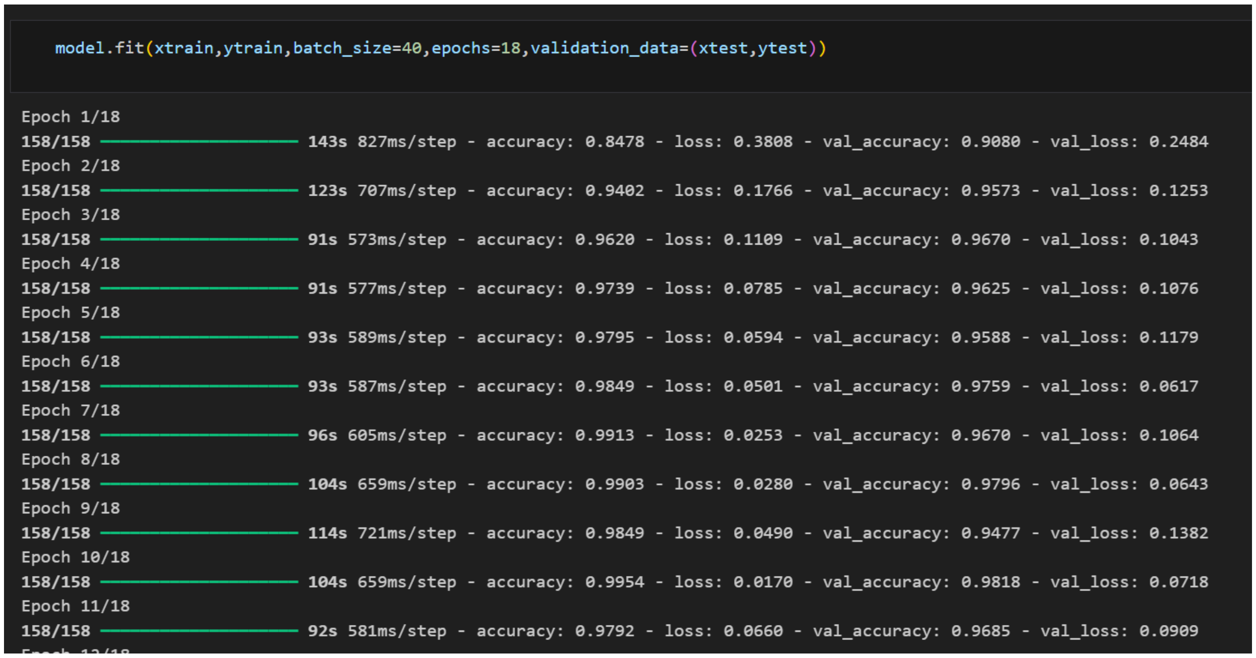
Compiling the Model



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

This line of code prepares the neural network for training. The `optimizer="adam"` decides how the model updates its weights to reduce errors, and Adam is popular because it is fast and efficient. The `loss="sparse_categorical_crossentropy"` tells the model how to calculate error, and it is used when the output has multiple classes and the labels are given as numbers (like 0, 1, 2...). The `metrics=["accuracy"]` means the model will calculate accuracy so we can see how well it is predicting correct classes during training and testing.

Training the Model and Understanding Epoch Results

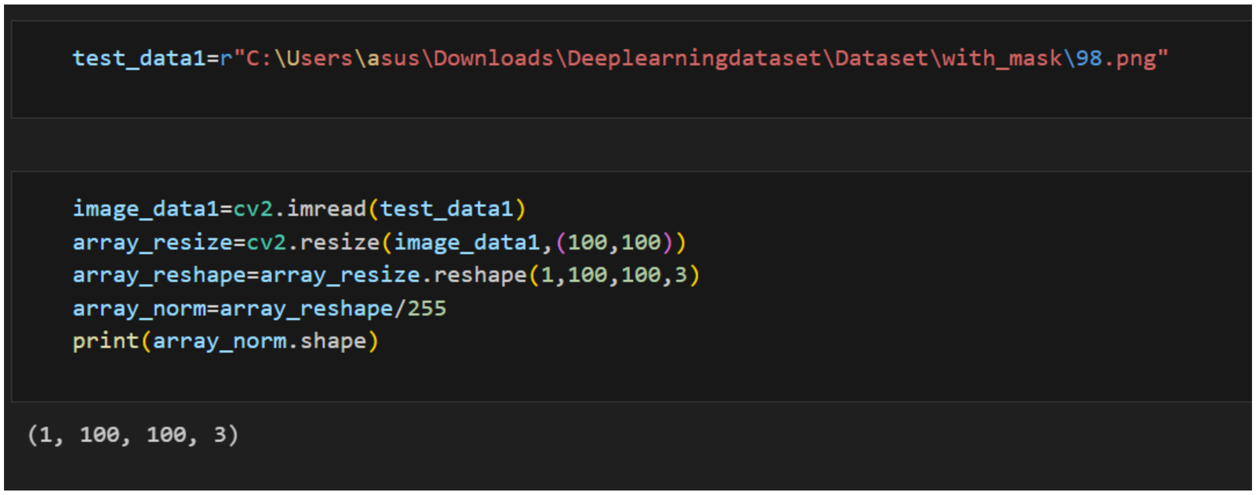


```
model.fit(xtrain,ytrain,batch_size=40,epochs=18,validation_data=(xtest,ytest))
```

```
Epoch 1/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 143s 827ms/step - accuracy: 0.8478 - loss: 0.3808 - val_accuracy: 0.9080 - val_loss: 0.2484
Epoch 2/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 123s 707ms/step - accuracy: 0.9402 - loss: 0.1766 - val_accuracy: 0.9573 - val_loss: 0.1253
Epoch 3/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 91s 573ms/step - accuracy: 0.9620 - loss: 0.1109 - val_accuracy: 0.9670 - val_loss: 0.1043
Epoch 4/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 91s 577ms/step - accuracy: 0.9739 - loss: 0.0785 - val_accuracy: 0.9625 - val_loss: 0.1076
Epoch 5/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 93s 589ms/step - accuracy: 0.9795 - loss: 0.0594 - val_accuracy: 0.9588 - val_loss: 0.1179
Epoch 6/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 93s 587ms/step - accuracy: 0.9849 - loss: 0.0501 - val_accuracy: 0.9759 - val_loss: 0.0617
Epoch 7/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 96s 605ms/step - accuracy: 0.9913 - loss: 0.0253 - val_accuracy: 0.9670 - val_loss: 0.1064
Epoch 8/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 104s 659ms/step - accuracy: 0.9903 - loss: 0.0280 - val_accuracy: 0.9796 - val_loss: 0.0643
Epoch 9/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 114s 721ms/step - accuracy: 0.9849 - loss: 0.0490 - val_accuracy: 0.9477 - val_loss: 0.1382
Epoch 10/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 104s 659ms/step - accuracy: 0.9954 - loss: 0.0170 - val_accuracy: 0.9818 - val_loss: 0.0718
Epoch 11/18
158/158 ━━━━━━━━━━━━━━━━━━━━ 92s 581ms/step - accuracy: 0.9792 - loss: 0.0660 - val_accuracy: 0.9685 - val_loss: 0.0909
```

This output shows the process of training the neural network using the `model.fit()` function. The model is trained on the training data (`xtrain`, `ytrain`) with a batch size of 40 for 18 epochs, meaning the entire dataset is passed through the network 18 times. After each epoch, the model reports training accuracy and loss, along with validation accuracy and validation loss calculated using the test data (`xtest`, `ytest`). As the epochs progress, training accuracy increases and training loss decreases, showing that the model is learning effectively. The validation accuracy also improves and remains high, indicating good generalization to unseen data, while small fluctuations in validation loss suggest minor variations but no severe overfitting.

Preprocessing a Single Image for Model Prediction



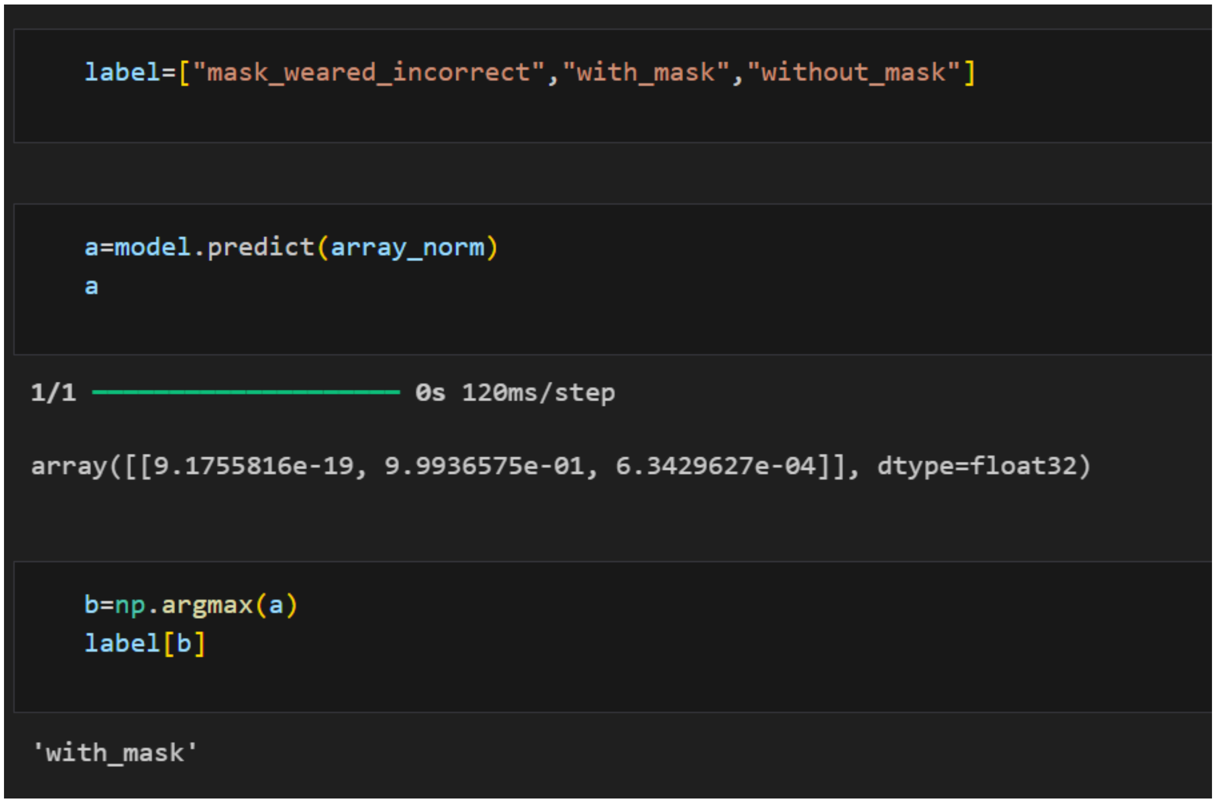
```
test_data1=r"C:\Users\asus\Downloads\Deeplearningdataset\Dataset\with_mask\98.png"
```

```
image_data1=cv2.imread(test_data1)
array_resize=cv2.resize(image_data1,(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 correctly given as input to a deep learning model. First, the image file path is stored in `test_data1`, and `cv2.imread()` reads the image from that location. The image is then resized to `100×100` pixels using `cv2.resize()` to match the input size expected by the model. Next, the image is reshaped to `(1, 100, 100, 3)`, where 1 represents one image, `100×100` is the image height and width, and 3 corresponds to the RGB color channels. After that, the pixel values are normalized by dividing by 255 so they fall between `0` and `1`, which helps the model make better predictions. Finally, the shape of the processed image is printed to confirm it is in the correct format for prediction.

Predicting the Class of an Input Image



```
label=["mask_weared_incorrect","with_mask","without_mask"]
```

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

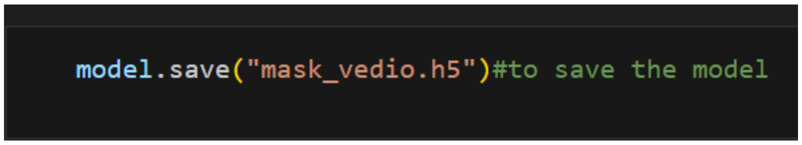
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 120ms/step

array([[9.1755816e-19, 9.9936575e-01, 6.3429627e-04]], dtype=float32)
```

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

```
'with_mask'
```

This code is used to predict the class of a single image using the trained model. The `label` list defines the class names in the same order as the model's output neurons: mask_weared_incorrect, with_mask, and without_mask. The `model.predict(array_norm)` function returns prediction probabilities for each class, showing how confident the model is about the image belonging to each category. In the output array, the highest probability value corresponds to the with_mask class. The `np.argmax(a)` function finds the index of this highest probability, and `label[b]` converts that index into the actual class name. As a result, the model correctly predicts that the person in the image is wearing a mask.

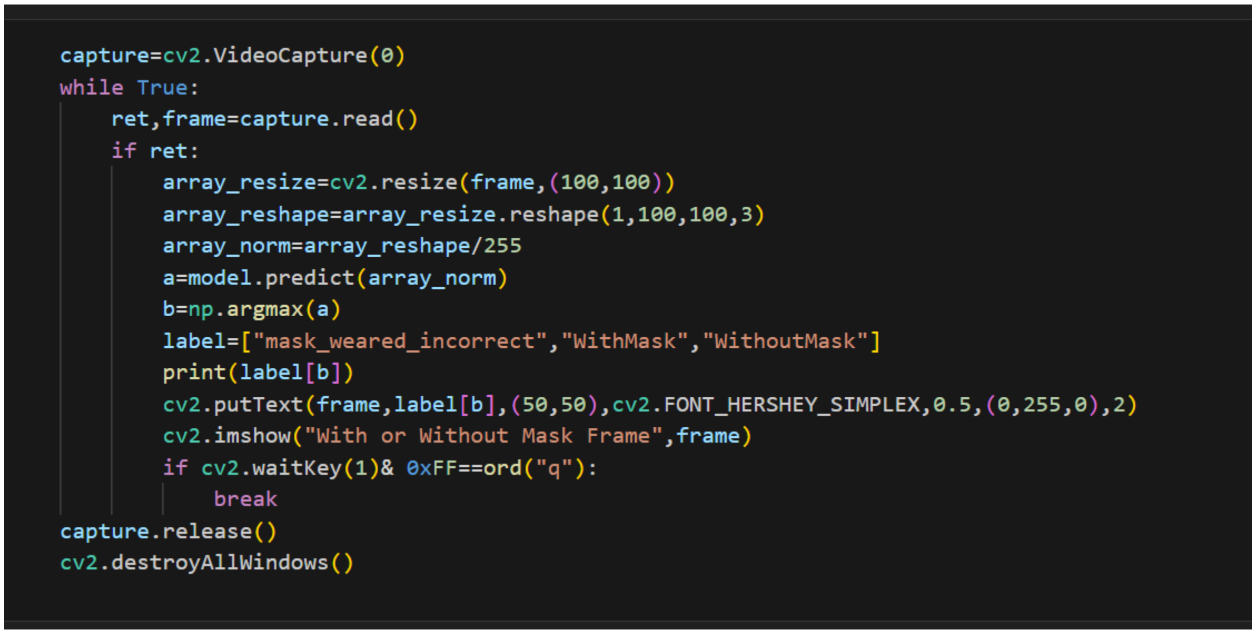
Saving the Trained Deep Learning Model



```
model.save("mask_vedio.h5")#to save the model
```

This line of code saves the trained neural network model to a file named `mask_vedio.h5` using the `model.save()` function. The `.h5` file stores the complete model, including its architecture, learned weights, and training configuration, so it can be reused later without retraining. This is useful for deploying the model in real-world applications or loading it again for further testing or predictions using `load_model()`.

Real-Time Face Mask Detection Using Webcam



```
capture=cv2.VideoCapture(0)
while True:
    ret,frame=capture.read()
    if ret:
        array_resize=cv2.resize(frame,(100,100))
        array_reshape=array_resize.reshape(1,100,100,3)
        array_norm=array_reshape/255
        a=model.predict(array_norm)
        b=np.argmax(a)
        label=["mask_weared_incorrect","WithMask","WithoutMask"]
        print(label[b])
        cv2.putText(frame,label[b],(50,50),cv2.FONT_HERSHEY_SIMPLEX,0.5,(0,255,0),2)
        cv2.imshow("With or Without Mask Frame",frame)
        if cv2.waitKey(1)& 0xFF==ord("q"):
            break
capture.release()
cv2.destroyAllWindows()
```

This code captures live video from the webcam and performs real-time face mask detection using the trained deep learning model. The `cv2.VideoCapture(0)` starts the webcam, and the `while  True` loop continuously reads video frames. Each captured frame is resized to `100×100`, reshaped, and normalized to match the model's input format before making a prediction. The model predicts whether the person is wearing a mask correctly, incorrectly, or not wearing a mask, and the class with the highest probability is selected using `np.argmax()`. The predicted label is then displayed on the video frame using `cv2.putText()`. The live video window is shown using `cv2.imshow()`, and the loop stops when the user presses the "q" key. Finally, the webcam is released and all OpenCV windows are closed properly.