Detailed Explanation of Face Mask Detection and Classification Code

Vivekananda,240628

June 2025

1 Code Breakdown and Explanation

The Python code can be logically divided into the following sections:

1. Imports and Utility Functions
2. Dataset Preparation
3. Custom PyTorch Dataset
4. Data Preprocessing and Splitting
5. Model Definition (SimpleCNN)
6. Model Training
7. Model Evaluation
8. Model Saving
9. Inference with OpenCV for Real-time Detection and Classification

1.1 1. Imports and Utility Functions

This section imports necessary libraries and defines a helper function to parse XML annotation files.

```python
import os
import xml.etree.ElementTree as ET
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import torch
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
import torch.optim as optim
import os
def extract_annotations(xml_path):
    xml_tree = ET.parse(xml_path)
    root = xml_tree.getroot()
    img_name = root.find('filename').text
    annotations = []

    for item in root.findall('object'):
        category = item.find('name').text
        coords = item.find('bndbox')
        annotation = {
            'category': category,
            'xmin': int(coords.find('xmin').text),
            'ymin': int(coords.find('ymin').text),
            'xmax': int(coords.find('xmax').text),
            'ymax': int(coords.find('ymax').text)
        }
        annotations.append(annotation)

    return img_name, annotations
```
Listing 1: Imports and Annotation Extraction

Explanation:

- `import os`: Facilitates interaction with the operating system, such as navigating directories and constructing file paths.
- `import xml.etree.ElementTree as ET`: Used for parsing and traversing XML documents, specifically for extracting bounding box coordinates and labels from annotation files.
- `import matplotlib.pyplot as plt`: A plotting library for creating static, animated, and interactive visualizations in Python, primarily used here for displaying the final image with detections.
- `from PIL import Image`: Part of the Pillow (PIL Fork) library, essential for opening, manipulating, and saving various image file formats. It's used for loading images and cropping face regions.
- `import numpy as np`: The fundamental package for numerical computation in Python, providing support for large, multi-dimensional arrays and matrices.
- `import torch`: The core PyTorch library, which offers powerful tensor computation (like NumPy with GPU support) and deep learning functionalities.
- `import torchvision.transforms as transforms`: A module from PyTorch's `torchvision` package that provides common image transformations to prepare images for neural networks (e.g., resizing, converting to tensor).
- `import torch.nn as nn`: Contains base classes for neural network modules, allowing the definition of layers like convolutional layers and linear layers.
- `import torch.nn.functional as F`: Provides functional interfaces for neural network operations (e.g., activation functions like ReLU, pooling operations) that typically do not have learnable parameters.
- `from torch.utils.data import Dataset, DataLoader`: `Dataset` is an abstract class representing a dataset, while `DataLoader` wraps a dataset to provide an iterable over it, supporting batching, shuffling, and parallel data loading.
- `from sklearn.model`$_selectionimporttrain\_test\_split : Afunctionfromscikit-learnforsplittingdatasetsintorandomt$
  $Containsvariousoptimizationalgorithms(e.g., Adam, SGD)usedtoadjustmodelparametersduringtrainingtominimiz$
- `import cv2`: OpenCV library, imported later in the script for real-time face detection (using Haar Cascades) and drawing on images.
- `from collections import Counter`: Used to count hashable objects, particularly useful here for analyzing the distribution of class labels in the dataset.
- `def extract_annotations(xml_path):`: This function is designed to parse an XML annotation file and extract relevant information about objects within an image.
  - `xml_tree = ET.parse(xml_path)`: Parses the XML file specified by `xml_path` into an ElementTree object, making its content accessible for navigation.
  - `root = xml_tree.getroot()`: Retrieves the root element of the XML document, serving as the starting point for searching other elements.
  - `img_name = root.find('filename').text`: Finds the `<filename>` tag directly under the root and extracts its text content, which is the name of the image file.
  - `annotations = []`: Initializes an empty list to store dictionaries, where each dictionary will represent one annotated object (e.g., a face) found in the image.
  - `for item in root.findall('object'):`: Iterates through every `<object>` tag within the XML document. Each `<object>` tag describes a single object in the image.

  - `category = item.find('name').text`: Inside an `<object>` tag, it finds the `<name>` tag and extracts its text, which typically represents the class label (e.g., "with_mask").
  - `coords = item.find('bndbox')`: Finds the `<bndbox>` tag, which contains the bounding box coordinates for the current object.
  - `annotation = {...}`:
    Constructs a dictionary for the current object. It includes the `category` and integer values for `xmin`, `ymin`, `xmax`, `ymax`. `int()` conversion is necessary because XML text content is strings.
  - `annotations.append(annotation)`: Adds the populated `annotation` dictionary to the `annotations` list.
  - `return img_name, annotations`: Returns the image filename and the complete list of annotations extracted from the XML file.

1.2 2. Dataset Preparation and Exploration

This section constructs the main dataset by iterating through annotation files, extracting information, and then analyzing the label distribution.

```python
root_dir = "/kaggle/input/face-mask-detection"
ann_dir = os.path.join(root_dir, "annotations")
img_dir = os.path.join(root_dir, "images")

dataset = []
for file_name in os.listdir(ann_dir):
    ann_file = os.path.join(ann_dir, file_name)
    img_file, objects = extract_annotations(ann_file)

    for obj in objects:
        dataset.append({
            "image_path": os.path.join(img_dir, img_file),
            "label": obj["category"],
            "bbox": (obj["xmin"], obj["ymin"], obj["xmax"], obj["ymax"])
        })

print(f"Total cropped faces: {len(dataset)}")
IMG_DIR = '../input/face-mask-detection/images'
ANNOT_DIR = '../input/face-mask-detection/annotations'
image_files = [f for f in os.listdir(IMG_DIR) if f.endswith('.jpg') or f.endswith('.png'
    )]
print(len(image_files))
xml_files = [f for f in os.listdir(ANNOT_DIR) if f.endswith('.xml')]
print(len(xml_files))
from collections import Counter
labels = [item['label'] for item in dataset]
label_counts = Counter(labels)
print("Dataset Label Distribution:")
for label, count in label_counts.items():
    print(f"{label}: {count}")
class_to_index = {
    "with_mask": 0,
    "without_mask": 1,
    "mask_weared_incorrect": 2
}
```

Listing 2: Dataset Preparation and Exploration

Explanation:

- `root_dir = "/kaggle/input/face-mask-detection"`: Defines the base directory for the dataset.
- `ann_dir = os.path.join(root_dir, "annotations")`: Constructs the full path to the annotations directory.
- `img_dir = os.path.join(root_dir, "images")`: Constructs the full path to the images directory.
- `dataset = []`: Initializes an empty list that will store processed entries. Each entry will be a dictionary containing the image path, mask label, and bounding box for a single face.

- `for file_name in os.listdir(ann_dir):`: Loops through every file name present in the annotations directory.
- `ann_file = os.path.join(ann_dir, file_name)`: Creates the absolute path to the current XML annotation file.
- `img_file, objects = extract_annotations(ann_file)`: Calls the `extract_annotations` function to parse the current XML file, returning the associated image filename and a list of detected objects (faces) within it.
- `for obj in objects:`: Iterates through each individual object (face) found in the current image's annotations.
- `dataset.append({...})`: For each `obj`, a dictionary is created and appended to the `dataset` list. This dictionary contains:
  - `"image_path"`: The full path to the image file, constructed using `img_dir` and `img_file`.
  - `"label"`: The category of the mask (e.g., "with_mask") extracted from the XML.
  - `"bbox"`: A tuple containing the integer bounding box coordinates (xmin, ymin, xmax, ymax).
- `print(f"Total cropped faces: {len(dataset)}")`: Prints the total number of individual face instances that have been extracted and added to the `dataset` list.
- `IMG_DIR = '../input/face-mask-detection/images'`
  `ANNOT_DIR = '../input/face-mask-detection/annotations'`: These lines are redundant as paths were already defined, but serve as explicit references to the directories.
- `image_files = [...]`
  `print(len(image_files))`: Lists all files in the image directory ending with '.jpg' or '.png' and prints their count. This is a basic sanity check for data availability.
- `xml_files = [...]`
  `print(len(xml_files))`: Lists all files in the annotation directory ending with '.xml' and prints their count, another sanity check.
- `labels = [item['label'] for item in dataset]`: Creates a new list containing only the 'label' (mask category) for every entry in the `dataset`.
- `label_counts = Counter(labels)`: Uses the `Counter` object from the `collections` module to count the occurrences of each unique label. This helps in understanding the class distribution (e.g., how many 'with_mask', 'without_mask' instances).
- `print("Dataset Label Distribution:")`
  `for label, count in label_counts.items():`
  `print(f"{label}: {count}")`: Prints a clear overview of how many samples belong to each mask category.
- `class_to_index = {...}`:
  A dictionary mapping the string-based class labels to integer indices. Neural networks in PyTorch typically require integer labels (0, 1, 2, ...) for classification tasks, especially when using loss functions like `CrossEntropyLoss`.

1.3 3. Custom PyTorch Dataset

This section defines a custom `Dataset` class, a fundamental component in PyTorch for handling data.

```python
from torch.utils.data import Dataset
from PIL import Image

class FaceMaskDataset(Dataset):
    def __init__(self, entries, augmentations=None):
        self.entries = entries
        self.augmentations = augmentations

    def __len__(self):
        return len(self.entries)

    def __getitem__(self, index):
        record = self.entries[index]
        img = Image.open(record["image_path"]).convert("RGB")
        x1, y1, x2, y2 = record["bbox"]
        cropped_face = img.crop((x1, y1, x2, y2))

        if self.augmentations:
            cropped_face = self.augmentations(cropped_face)

        target = class_to_index[record["label"]]
        return cropped_face, target
```

Listing 3: Custom PyTorch Dataset

Explanation:

- `class FaceMaskDataset(Dataset):`: Defines a custom dataset class named `FaceMaskDataset` that inherits from `torch.utils.data.Dataset`. This inheritance is mandatory for PyTorch's `DataLoader` to work correctly.
- `__init__(self, entries, augmentations=None)`: The constructor method for the dataset.
  - `self.entries = entries`: Stores the list of data records (each containing image path, label, bbox) that was prepared earlier.
  - `self.augmentations = augmentations`: Stores an optional transformation pipeline (e.g., resizing, converting to tensor) to be applied to the images.
- `__len__(self)`: This method is required by `torch.utils.data.Dataset`.
  - `return len(self.entries)`: Returns the total number of samples (cropped faces) in the dataset, allowing `DataLoader` to know the size of the dataset.
- `__getitem__(self, index)`: This is another required method for `torch.utils.data.Dataset`. It defines how a single sample at a given `index` is loaded and processed.
  - `record = self.entries[index]`: Retrieves the dictionary corresponding to the given `index` from the stored `entries` list.
  - `img = Image.open(record["image_path"]).convert("RGB")`: Opens the image file specified by `record["image_path"]` using PIL. `.convert("RGB")` ensures the image is in a consistent 3-channel RGB format, even if the original is grayscale or has an alpha channel.
  - `x1, y1, x2, y2 = record["bbox"]`: Unpacks the bounding box coordinates (xmin, ymin, xmax, ymax) from the current record.
  - `cropped_face = img.crop((x1, y1, x2, y2))`: Uses PIL's `crop` method to extract the rectangular region of the face defined by the bounding box.
  - `if self.augmentations:  cropped_face = self.augmentations(cropped_face)`: Checks if an augmentation pipeline was provided during initialization. If so, it applies these transformations to the `cropped_face`.
  - `target = class_to_index[record["label"]]`: Converts the string label (e.g., "with_mask") into its corresponding integer index (e.g., 0) using the `class_to_index` mapping. This integer is the ground truth target for the classification model.
  - `return cropped_face, target`: Returns the processed and potentially augmented image tensor of the cropped face, and its corresponding integer class label.

1.4 4. Data Preprocessing and Splitting

This section defines image transformations, splits the dataset into training and testing sets, and creates `DataLoader` objects.

```python
from torchvision import transforms
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
preprocess = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])
train_set, test_set = train_test_split(
    dataset,
    test_size=0.2,
    stratify=[entry['label'] for entry in dataset],
    random_state=42
)
training_data = FaceMaskDataset(train_set, augmentations=preprocess)
testing_data = FaceMaskDataset(test_set, augmentations=preprocess)
train_loader = DataLoader(training_data, batch_size=32, shuffle=True)
test_loader = DataLoader(testing_data, batch_size=32, shuffle=False)
print(f"Training Samples: {len(training_data)} Testing Samples: {len(testing_data)}")
```

Listing 4: Data Preprocessing and Splitting

Explanation:

- `preprocess = transforms.Compose([...])`: Defines a composite transformation pipeline to be applied to images. `Compose` chains multiple transformations together.
  - `transforms.Resize((128, 128))`: Resizes all input images (cropped faces) to a fixed size of 128×128 pixels. This is a common requirement for neural networks, which expect fixed-size inputs.
  - `transforms.ToTensor()`: Converts the PIL Image (or NumPy array) into a PyTorch `Tensor`. It also automatically scales the pixel intensity values from the $[0, 255]$ range to the $[0.0, 1.0]$ range, and rearranges the image dimensions from HWC (Height, Width, Channels) to CWH (Channels, Width, Height), which is the standard input format for PyTorch convolutional layers.
- `train_set, test_set = train_test_split(...)`: Splits the ‘dataset‘ list (which contains dictionaries of image info) into two parts: a training set and a testing set.
  - `dataset`: The complete list of all cropped face entries.
  - `test_size=0.2`: Specifies that 20% of the data will be used for the test set, and the remaining 80% for the training set.
  - `stratify=[entry['label'] for entry in dataset]`: This is a critical argument, especially for datasets with class imbalance. It ensures that the proportion of each class label (e.g., ”with_mask”, ”without_mask”) in the training set is approximately the same as in the testing set. This helps ensure both sets are representative of the overall data distribution.
  - `random_state=42`: Sets a seed for the random number generator used in the splitting process. This makes the split reproducible, meaning you’ll get the exact same train/test split every time you run the code with this seed.
- `training_data = FaceMaskDataset(train_set, augmentations=preprocess)`: Creates an instance of the custom `FaceMaskDataset` for the training data, passing the `train_set` and the defined `preprocess` transformations.
- `testing_data = FaceMaskDataset(test_set, augmentations=preprocess)`: Creates another instance of `FaceMaskDataset` for the testing data, using the `test_set` and the same `preprocess` transformations.
- `train_loader = DataLoader(training_data, batch_size=32, shuffle=True)`:
  - `DataLoader`: Wraps the `training_data` dataset to provide an iterable over it. It efficiently loads data in batches.
  - `batch_size=32`: Each iteration from `train_loader` will yield a mini-batch of 32 image-label pairs. Training in batches is more computationally efficient and helps with model generalization.
  - `shuffle=True`: Randomly shuffles the data samples at the beginning of each training epoch. This is vital to prevent the model from learning the order of samples and to improve its ability to generalize to unseen data.

- `test_loader = DataLoader(testing_data, batch_size=32, shuffle=False)`: Creates a `DataLoader` for the testing data. `shuffle=False` because the order of test samples does not affect evaluation metrics.
- `print(f"Training Samples: {len(training_data)} Testing Samples: {len(testing_data)}")`: Prints the number of samples designated for training and testing, confirming the split.

1.5 5. Model Definition (SimpleCNN)

This section defines the architecture of the Convolutional Neural Network (CNN) used for mask classification.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.layer2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.pooling = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dense1 = nn.Linear(in_features=64 * 30 * 30, out_features=128)
        self.dense2 = nn.Linear(in_features=128, out_features=3)

    def forward(self, input_tensor):
        out = self.pooling(F.relu(self.layer1(input_tensor)))
        out = self.pooling(F.relu(self.layer2(out)))
        out = out.view(-1, 64 * 30 * 30)
        out = F.relu(self.dense1(out))
        out = self.dense2(out)
        return out
```

Listing 5: SimpleCNN Model Definition

Explanation:

- `class SimpleCNN(nn.Module):`: Defines the neural network model named `SimpleCNN`. It inherits from `nn.Module`, which is the base class for all neural network models in PyTorch. This inheritance provides core functionalities like tracking parameters and defining the forward pass.
- `__init__(self):`: The constructor of the `SimpleCNN` class, where all the layers of the neural network are defined.
  - `super(SimpleCNN, self).__init__()`: Calls the constructor of the parent class (`nn.Module`). This is crucial for proper initialization of PyTorch modules and ensures that PyTorch's internal mechanisms for managing layers and parameters are set up.
  - `self.layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)`:
    * `nn.Conv2d`: Defines a 2D convolutional layer.
    * `in_channels=3`: Specifies that the input to this layer will have 3 channels (for RGB images).
    * `out_channels=32`: This layer will produce 32 output feature maps (i.e., it applies 32 different filters).
    * `kernel_size=3`: The size of the convolutional filter is 3×3.
  - `self.layer2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)`:
    * Similar to `layer1`, but its `in_channels` is 32 (matching the `out_channels` of the previous layer). It generates 64 output feature maps.
  - `self.pooling = nn.MaxPool2d(kernel_size=2, stride=2)`:
    * `nn.MaxPool2d`: Defines a 2D max pooling layer.
    * `kernel_size=2`: The size of the pooling window is 2×2.
    * `stride=2`: The pooling window moves 2 pixels at a time, effectively downsampling the spatial dimensions (height and width) by half.
  - `self.dense1 = nn.Linear(in_features=64 * 30 * 30, out_features=128)`:

    * `nn.Linear`: Defines a fully connected (dense) layer.
    * `in_features=64 * 30 * 30`: This value represents the flattened size of the feature maps after the convolutional and pooling layers.
      Calculation trace for spatial dimensions:
        · Input image size: 128×128
        · After `layer1` (Conv2d, kernel 3×3, no padding, stride 1): Output size $= (128 - 3 + 1) = 126 \times 126$. Channels: 32.
        · After first `pooling` (MaxPool2d, kernel 2×2, stride 2): Output size $= 126/2 = 63 \times 63$. Channels: 32.
        · After `layer2` (Conv2d, kernel 3×3, no padding, stride 1): Output size $= (63 - 3 + 1) = 61 \times 61$. Channels: 64.
        · After second `pooling` (MaxPool2d, kernel 2×2, stride 2): Output size $= 61/2 = 30.5$. Since output dimensions must be integers, it's typically floored to 30×30. Channels: 64.

      So, the input to this linear layer needs to be a flat vector of $64 \times 30 \times 30$ features.
    * `out_features=128`: This layer will output 128 features to the next layer.
  - `self.dense2 = nn.Linear(in_features=128, out_features=3)`:
    * `in_features=128`: Takes the output from the previous dense layer.
    * `out_features=3`: The final output layer has 3 neurons, corresponding to the 3 distinct mask classes ("with_mask", "without_mask", "mask_weared_incorrect"). The values outputted are raw logits, which will be processed by the loss function.

- `def forward(self, input_tensor):`: This method defines the forward pass of the neural network, specifying how input data flows through the layers to produce an output.
  - `out = self.pooling(F.relu(self.layer1(input_tensor)))`:
    1. `self.layer1(input_tensor)`: The input tensor first passes through the first convolutional layer.
    2. `F.relu(...)`: The Rectified Linear Unit (ReLU) activation function is applied. ReLU introduces non-linearity, allowing the network to learn more complex relationships in the data. `F.relu` is used for functional (stateless) activations.
    3. `self.pooling(...)`: The output then goes through the max pooling layer, reducing its spatial dimensions.
  - `out = self.pooling(F.relu(self.layer2(out)))`: The same sequence of operations (convolution, ReLU, pooling) is applied for the second set of layers.
  - `out = out.view(-1, 64 * 30 * 30)`:
    * `out.view(...)`: Reshapes (flattens) the tensor. Convolutional layers output 4D tensors (`batch_size`, `channels`, `height`, `width`). For fully connected layers, the input must be a 2D tensor (`batch_size`, `features`).
    * `-1`: PyTorch automatically infers the batch size dimension.
    * `64 * 30 * 30`: The total number of features (channels multiplied by height and width) from the last pooling layer, which will be fed into the first dense layer.
  - `out = F.relu(self.dense1(out))`: The flattened output passes through the first fully connected layer (`self.dense1`), followed by another ReLU activation.
  - `out = self.dense2(out)`: The result then goes through the final fully connected layer (`self.dense2`). This layer produces the raw logits for each class. It's standard practice not to apply a Softmax activation here, as `nn.CrossEntropyLoss` implicitly applies it.
  - `return out`: The raw logits are returned as the model's output.

1.6 6. Training the Model

This section sets up the training environment (device, optimizer, loss function) and executes the training loop for the CNN model.

```python
device=torch.device("cuda" if torch.cuda.is_available() else "cpu" )
print(device)
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net = SimpleCNN().to(device)
loss_function = nn.CrossEntropyLoss()
adam_optimizer = optim.Adam(net.parameters(), lr=0.001)

total_epochs = 100
for current_epoch in range(total_epochs):
    net.train()
    epoch_total_loss = 0.0

    for img_batch, lbl_batch in train_loader:
        img_batch, lbl_batch = img_batch.to(device), lbl_batch.to(device)

        adam_optimizer.zero_grad()
        output_preds = net(img_batch)
        batch_loss = loss_function(output_preds, lbl_batch)
        batch_loss.backward()
        adam_optimizer.step()

        epoch_total_loss += batch_loss.item()

    mean_loss = epoch_total_loss / len(train_loader)
    print(f"Epoch {current_epoch+1}/{total_epochs} - Mean Loss: {mean_loss:.4f}")
```
Listing 6: Model Training Loop

Explanation:

- `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`:
  - This line determines whether a CUDA-enabled GPU is available. If `torch.cuda.is_available()` returns `True`, the `device` is set to `"cuda"` (GPU); otherwise, it's set to `"cpu"`. Training on a GPU is significantly faster for deep learning models.
- `print(device)`: Prints the selected device to the console (e.g., "cuda" or "cpu").
- `net = SimpleCNN().to(device)`:
  - `SimpleCNN()`: Instantiates an object of our defined `SimpleCNN` model.
  - `.to(device)`: Moves the entire model (all its parameters and internal buffers) from CPU memory to the specified device (GPU if available, otherwise remains on CPU). This is essential for computations to be performed on the correct hardware.
- `loss_function = nn.CrossEntropyLoss()`: Defines the loss function used to measure the discrepancy between the model's predictions and the true labels. `nn.CrossEntropyLoss` is commonly used for multi-class classification. It internally combines a Softmax activation and Negative Log Likelihood Loss. It expects raw logits from the model's output and integer class labels as targets.
- `adam_optimizer = optim.Adam(net.parameters(), lr=0.001)`:
  - Defines the optimizer, which is responsible for updating the model's weights and biases during training based on the gradients computed during backpropagation.
  - `optim.Adam`: The Adam optimizer is an adaptive learning rate optimization algorithm known for its efficiency and good performance in many deep learning tasks.
  - `net.parameters()`: Provides all the learnable parameters (weights and biases) of the `net` model to the optimizer, so it knows which values to update.
  - `lr=0.001`: Sets the initial learning rate, which controls the step size taken during parameter updates.
- `total_epochs = 100`: Sets the number of training epochs. An epoch represents one complete pass through the entire training dataset.
- `for current_epoch in range(total_epochs):`: The main training loop. This loop iterates for the specified number of epochs.

- `net.train()`: Sets the model to training mode. This is crucial because certain layers (like Dropout layers or Batch Normalization layers) behave differently during training (e.g., Dropout layers are active, Batch Normalization uses batch statistics) compared to evaluation.
- `epoch_total_loss = 0.0`: Initializes a variable to accumulate the loss over all batches within the current epoch.
- `for img_batch, lbl_batch in train_loader:`: The inner loop iterates through mini-batches of data provided by the `train_loader`. Each iteration yields a batch of images and their corresponding labels.
  - `img_batch, lbl_batch = img_batch.to(device), lbl_batch.to(device)`: Moves the image and label tensors for the current batch to the specified `device` (GPU for faster computation if available).
  - `adam_optimizer.zero_grad()`: **This is a critical step.** Before computing gradients for the current batch, any old gradients accumulated from previous iterations must be cleared. If not, gradients would accumulate, leading to incorrect updates.
  - `output_preds = net(img_batch)`: Performs the forward pass. The input batch (`img_batch`) is fed into the `net` (our CNN model), and it produces `output_preds` (the raw logits for each class).
  - `batch_loss = loss_function(output_preds, lbl_batch)`: Calculates the loss for the current batch. The `loss_function` (CrossEntropyLoss) compares the model's predicted logits (`output_preds`) with the true labels (`lbl_batch`).
  - `batch_loss.backward()`: Performs the backward pass (backpropagation). This computes the gradients of the `batch_loss` with respect to all learnable parameters of the model.
  - `adam_optimizer.step()`: Updates the model's parameters (weights and biases) using the gradients computed in the backward pass and the Adam optimization algorithm.
  - `epoch_total_loss += batch_loss.item()`: Adds the loss of the current batch to the `epoch_total_loss`. `.item()` extracts the Python number from a single-element tensor.
- `mean_loss = epoch_total_loss / len(train_loader)`: After processing all batches in an epoch, calculates the average (mean) loss for that epoch.
- `print(f"Epoch {current_epoch+1}/{total_epochs} - Mean Loss: {mean_loss:.4f}")`: Prints the current epoch number and the calculated mean loss, providing feedback on the training progress.

1.7 7. Model Evaluation

This section evaluates the trained model's performance on the unseen test dataset.

```python
net.eval()
correct_preds = 0
total_samples = 0

with torch.no_grad():
    for val_imgs, val_labels in test_loader:
        val_imgs, val_labels = val_imgs.to(device), val_labels.to(device)
        logits = net(val_imgs)
        _, predictions = torch.max(logits, dim=1)
        total_samples += val_labels.size(0)
        correct_preds += (predictions == val_labels).sum().item()

accuracy = 100 * correct_preds / total_samples
print(f"Test Accuracy: {accuracy:.2f}%")
```

Listing 7: Model Evaluation

Explanation:

- `net.eval()`: Sets the model to evaluation (or inference) mode. This is important for consistency and correctness. In evaluation mode, layers like Dropout are deactivated (no random dropping of neurons), and Batch Normalization layers use their pre-computed running means and variances instead of batch-specific statistics.
- `correct_preds = 0`
  `total_samples = 0`: Initializes two counters to keep track of the number of correctly predicted samples and the total number of samples in the test set, respectively. These will be used to calculate accuracy.

- `with torch.no_grad():`: This is a context manager that disables gradient calculation.
  - During evaluation, we do not need to compute or store gradients because we are not updating the model's weights. Disabling gradient calculation saves memory and speeds up computations.
- `for val_imgs, val_labels in test_loader:`: Iterates through batches of data from the `test_loader`.
  - `val_imgs, val_labels = val_imgs.to(device), val_labels.to(device)`: Moves the validation images and labels for the current batch to the same device (GPU or CPU) where the model resides.
  - `logits = net(val_imgs)`: Performs a forward pass on the validation images through the trained `net` model to obtain the raw class logits.
  - `_, predictions = torch.max(logits, dim=1)`:
    * `torch.max(logits, dim=1)`: This function returns a tuple: (maximum values, indices of maximum values).
    * We are interested in the `indices` (the second element), which represent the predicted class for each sample in the batch. `dim=1` specifies that the maximum should be taken along the class dimension (i.e., for each sample, find the class with the highest logit). The `_` is used to ignore the maximum values themselves.
  - `total_samples += val_labels.size(0)`: Adds the number of samples in the current batch (`val_labels.size(0)` gives the batch size) to the `total_samples` counter.
  - `correct_preds += (predictions == val_labels).sum().item()`:
    * `(predictions == val_labels)`: Creates a boolean tensor where each element is `True` if the prediction matches the true label for that sample, and `False` otherwise.
    * `.sum()`: Sums the `True` values (which are treated as 1) to get the total number of correct predictions within the current batch.
    * `.item()`: Extracts the Python numerical value from the single-element tensor result of `.sum()`.
- `accuracy = 100 * correct_preds / total_samples`: Calculates the overall test accuracy as a percentage.
- `print(f"Test Accuracy: {accuracy:.2f}%")`: Prints the final calculated test accuracy, formatted to two decimal places.

1.8 8. Model Saving

This section demonstrates how to save the trained model's weights for future use.

```
torch.save(net.state_dict(), "mask_classifier.pth")
print("Model saved as mask_classifier.pth")
```

Listing 8: Model Saving

Explanation:

- `torch.save(net.state_dict(), "mask_classifier.pth")`:
  - `torch.save()`: A PyTorch function used to serialize and save an object to disk.
  - `net.state_dict()`: This method returns a Python dictionary containing all the learnable parameters (weights and biases) of the `net` model. It's standard practice to save only the `state_dict()` rather than the entire model object, as this results in smaller file sizes and allows for more flexible loading (e.g., loading weights into a different model architecture as long as the layer names match).
  - `"mask_classifier.pth"`: The filename where the model's state dictionary will be saved. The `.pth` or `.pt` extensions are commonly used for PyTorch model files.
- `print("Model saved as mask_classifier.pth")`: Confirms to the user that the model has been successfully saved.

1.9 9. Inference with OpenCV for Real-time Detection and Classification

This final section demonstrates how to load the trained model and use it in conjunction with OpenCV for real-time face detection and mask classification on a new image.

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import torch
from torchvision import transforms

mask_model = SimpleCNN()
mask_model.load_state_dict(torch.load("mask_classifier.pth", map_location=torch.device('cpu')))
mask_model.eval()
mask_model.to(device)


class_names = {0: "with_mask", 1: "without_mask", 2: "mask_wore_incorrect"}
class_colors = {
    "with_mask": (0, 255, 0),
    "without_mask": (255, 0, 0),
    "mask_wore_incorrect": (0, 165, 255)
}


image_path = "/kaggle/input/face-mask-detection/images/maksssksksss104.png"
raw_image = cv2.imread(image_path)

if raw_image is None:
    print("Check the path ")
else:
    processed_img = raw_image.copy()
    grayscale_img = cv2.cvtColor(processed_img, cv2.COLOR_BGR2GRAY)

    face_detector = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
    detected_faces = face_detector.detectMultiScale(grayscale_img, scaleFactor=1.1, minNeighbors=5)

    print(f"Number of faces found: {len(detected_faces)}")

    face_transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.ToTensor()
    ])


    for (x, y, w, h) in detected_faces:
        face_crop = processed_img[y:y+h, x:x+w]
        rgb_crop = cv2.cvtColor(face_crop, cv2.COLOR_BGR2RGB)
        face_img = Image.fromarray(rgb_crop)
        input_tensor = face_transform(face_img).unsqueeze(0).to(device)

        with torch.no_grad():
            logits = mask_model(input_tensor)
            _, predicted_class = torch.max(logits, dim=1)
            class_label = class_names[predicted_class.item()]
            box_color = class_colors[class_label]


        cv2.rectangle(processed_img, (x, y), (x+w, y+h), box_color, 2)
        cv2.putText(processed_img, class_label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, box_color, 2)


    final_result = cv2.cvtColor(processed_img, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(10, 8))
    plt.imshow(final_result)
    plt.axis('off')
    plt.title("Mask Classification Result")
    plt.show()
```

Listing 9: Inference with OpenCV

Explanation:

- `import cv2`
  `import numpy as np`
  `import matplotlib.pyplot as plt`
  `from PIL import Image`
  `import torch`
  `from torchvision import transforms`: These lines ensure all necessary libraries for inference and visualization are imported. Some are repeated for clarity of section dependency.

- `mask_model = SimpleCNN()`: An instance of the `SimpleCNN` class is created. This is crucial because `torch.load()` only loads the saved weights (`state_dict`), not the model's architecture (the class definition). So, the model architecture must be instantiated first.

- `mask_model.load_state_dict(torch.load("mask_classifier.pth", map_location=torch.device('cpu')))`:
  - `torch.load("mask_classifier.pth", ...)`: Loads the saved state dictionary from the specified file.
  - `map_location=torch.device('cpu')`: This argument is important. If the model was trained on a GPU and you're loading it on a CPU-only machine (or vice-versa), this ensures that all model parameters are correctly mapped to the desired device, preventing potential errors.
  - `mask_model.load_state_dict(...)`: Loads the parameters (weights and biases) from the loaded state dictionary into the `mask_model` instance.

- `mask_model.eval()`: Sets the loaded model to evaluation mode. This is essential for consistent predictions, as it disables layers like Dropout and sets Batch Normalization to use its learned running statistics.

- `mask_model.to(device)`: Moves the loaded model to the appropriate device (GPU or CPU) for inference, similar to during training.

- `class_names = {0: "with_mask", ...}`
  `class_colors = {"with_mask": (0, 255, 0), ...}`:
  - These dictionaries map the integer class predictions back to human-readable string labels and assign specific colors (in BGR format, standard for OpenCV) for drawing bounding boxes and text on the image.

- `image_path = "/kaggle/input/face-mask-detection/images/maksssksksss104.png"`: Specifies the path to the image on which inference will be performed.

- `raw_image = cv2.imread(image_path)`: Reads the image from the specified path using OpenCV. OpenCV's `imread` function by default reads images in BGR (Blue, Green, Red) color channel order, unlike PIL or Matplotlib which use RGB.

- `if raw_image is None: print("Check the path ")`: Checks if the image was successfully loaded. If `cv2.imread` fails (e.g., due to an incorrect path or corrupted file), it returns `None`.

- `else: processed_img = raw_image.copy()`: If the image is loaded, a copy is made to draw on, preserving the original.

- `grayscale_img = cv2.cvtColor(processed_img, cv2.COLOR_BGR2GRAY)`: Converts the BGR image to grayscale. Haar Cascades for face detection generally perform better and are faster on grayscale images.

- `face_detector = cv2.CascadeClassifier(cv2.data.haascades + "haarcascade_frontalface_default.xml")`:
  - Initializes OpenCV's pre-trained Haar Cascade classifier. This is a traditional machine learning method for object detection (specifically frontal faces here) that relies on cascades of simple features.

  - `cv2.data.haascades`: Provides access to the path where OpenCV's pre-trained cascade XML files are stored.

- `detected_faces = face_detector.detectMultiScale(grayscale_img, scaleFactor=1.1, minNeighbors=5)`:
  - Performs face detection on the `grayscale_img`.
  - `scaleFactor=1.1`: Defines how much the image size is reduced at each image scale. A smaller value (e.g., 1.05) makes the detection more thorough but slower.
  - `minNeighbors=5`: Specifies how many neighbors (overlapping detections) a candidate rectangle must have to be considered a true detection. Higher values result in fewer but higher-quality detections.
  - The function returns a list of rectangles, where each rectangle is represented by `(x, y, w, h)` (top-left corner coordinates, width, height) of a detected face.

- `print(f"Number of faces found: {len(detected_faces)}")`: Prints the count of faces detected by the Haar Cascade classifier.

- `face_transform = transforms.Compose([...])`: Defines a transformation pipeline specifically for the individual face crops before they are fed into the CNN. This is identical to the `preprocess` used during training.

- `for (x, y, w, h) in detected_faces:`: Iterates through each detected face.
  - `face_crop = processed_img[y:y+h, x:x+w]`: Crops the specific region of the image containing the detected face using NumPy array slicing.
  - `rgb_crop = cv2.cvtColor(face_crop, cv2.COLOR_BGR2RGB)`: Converts the BGR color format of the `face_crop` (from OpenCV) to RGB, which is the expected format for PIL and PyTorch's `transforms`.
  - `face_img = Image.fromarray(rgb_crop)`: Converts the NumPy array `rgb_crop` into a PIL Image object, which can then be processed by `face_transform`.
  - `input_tensor = face_transform(face_img).unsqueeze(0).to(device)`:
    * `face_transform(face_img)`: Applies the defined transformations (resizing to 128×128, converting to PyTorch tensor) to the cropped face.
    * `.unsqueeze(0)`: Adds an extra dimension at the beginning of the tensor. PyTorch models expect input in batches, so a single image tensor of shape `(C, H, W)` needs to become `(1, C, H, W)` (a batch of one image).
    * `.to(device)`: Moves the prepared input tensor to the same device as the model.
  - `with torch.no_grad():`: Disables gradient calculation during inference for efficiency.
  - `logits = mask_model(input_tensor)`: Feeds the preprocessed face tensor into the trained `mask_model` to get the raw class logits.
  - `_, predicted_class = torch.max(logits, dim=1)`: Extracts the index of the class with the highest logit, which represents the model's prediction.
  - `class_label = class_names[predicted_class.item()]`: Converts the integer predicted class (e.g., 0, 1, 2) back into its human-readable string label (e.g., "with_mask").
  - `box_color = class_colors[class_label]`: Retrieves the corresponding color (BGR tuple) for drawing the bounding box and text, based on the predicted class label.
  - `cv2.rectangle(processed_img, (x, y), (x+w, y+h), box_color, 2)`: Draws a rectangle around the detected face on the `processed_img`.
    * `(x, y)`: Top-left corner of the rectangle.
    * `(x+w, y+h)`: Bottom-right corner of the rectangle.
    * `box_color`: The color of the rectangle border (in BGR).
    * 2: The thickness of the rectangle border in pixels.
  - `cv2.putText(processed_img, class_label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, box_color, 2)`: Draws the predicted class label as text on the image.
    * `class_label`: The text to be displayed.

  * `(x, y - 10)`: The starting coordinates for the text (placed slightly above the bounding box).
  * `cv2.FONT_HERSHEY_SIMPLEX`: The font type.
  * `0.8`: The font scale.
  * `box_color`: The color of the text (in BGR).
  * `2`: The thickness of the text characters.

- `final_result = cv2.cvtColor(processed_img, cv2.COLOR_BGR2RGB)`: Converts the `processed_img` from OpenCV's BGR format to Matplotlib's preferred RGB format before displaying.

-
  ```
  plt.figure(figsize=(10, 8))
  plt.imshow(final_result)
  plt.axis('off')
  plt.title("Mask Classification Result")
  plt.show()
  ```
  : These lines use Matplotlib to display the final image with all detected faces, their bounding boxes, and the mask classification predictions. `plt.axis('off')` removes the axes and ticks for a cleaner image display.