Project Title: Emotion Detection from Uploaded Images

Project Overview

The "Emotion Detection from Uploaded Images" project aims to create a comprehensive system enabling users to upload an image through a Streamlit application and accurately detect and classify the emotion present in the image using Convolutional Neural Networks (CNNs). This project integrates machine learning, computer vision, and user interface design to develop a user-friendly application.

Domain Introduction:

Emotion detection through computer vision and machine learning is a rapidly growing field that involves analyzing facial expressions to identify human emotions. Leveraging convolutional neural networks (CNNs) and advanced facial recognition techniques, this technology finds significant applications in healthcare, education, and customer service by enhancing user interactions and providing deeper insights into human behavior. The integration of such systems into real-world applications promises improved user experiences and more personalized services.

Problem Statement:

Develop a Streamlit-based application that enables users to upload images and accurately detect and classify the emotions present using convolutional neural networks (CNNs).

Model Selection and Development

In this project, I employed two pre-trained Convolutional Neural Network (CNN) architectures, ResNet and VGG, and also developed a custom CNN model for emotion detection and classification. The choice of these models was driven by their proven effectiveness in image recognition tasks and their ability to handle the complexities of facial emotion detection.

Hyperparameter Tuning for ResNet:

- **Learning Rate:** Experimented with learning rates ranging from 0.001 to 0.0001.
- **Batch Size:** Tried different batch sizes (16, 32, 64) to find the optimal balance between training speed and model performance.
- **Optimization Algorithm:** Used Adam optimizer with tuned parameters for better convergence.

Hyperparameter Tuning for VGG:

- **Learning Rate:** Similar to ResNet, experimented with learning rates from 0.001 to 0.0001.

- **Batch Size:** Tested different batch sizes to optimize performance.
- **Dropout:** Implemented dropout layers to prevent overfitting.

My own model:

1. Data Preparation

- **Image Transformations:** A series of transformations are applied to the input images to prepare them for training. This includes:
    - **Resizing:** All images are resized to 48x48 pixels.
    - **Grayscale Conversion:** Images are converted to grayscale to reduce complexity and focus on structural features.
    - **Data Augmentation:** Techniques such as random horizontal flips, rotations, and color jittering are employed to enhance the diversity of the training dataset. This helps the model generalize better by exposing it to a variety of image conditions.
- **Dataset Organization:** The dataset is organized into directories for training and testing, utilizing the ImageFolder class from torchvision to load images and labels.
- **DataLoader Creation:** DataLoader objects are created for both training and testing datasets, allowing for efficient batching and shuffling during the training process.

2. Model Architecture

- **EmotionCNN Class:** The custom CNN architecture consists of three convolutional layers followed by max pooling, dropout, and fully connected layers:
    - **Convolutional Layers:**
        - conv1: First convolutional layer takes a single-channel input and outputs 32 feature maps.
        - conv2: Second layer outputs 64 feature maps.
        - conv3: Third layer outputs 128 feature maps.
    - **Pooling Layer:** Max pooling is applied after each convolutional layer to downsample the feature maps and reduce dimensionality.
    - **Fully Connected Layers:** The flattened output is passed through two fully connected layers, where the first layer has 512 neurons and the final layer outputs 7 classes corresponding to different emotions.
    - **Dropout Layer:** A dropout layer with a rate of 0.5 is used to prevent overfitting.

3. Training the Model

- **Loss Function and Optimizer:**

    - The model employs the Cross Entropy Loss function to quantify the difference between the predicted and actual labels.
    - The Adam optimizer is utilized with a learning rate of 0.001 and a weight decay of 1×10−51 \times 10^{-5}1×10−5 for regularization.
- **Learning Rate Scheduler:** A learning rate scheduler (ReduceLROnPlateau) is implemented to reduce the learning rate when the validation loss plateaus, promoting better convergence.
- **Training Loop:** The model is trained over a specified number of epochs (20 in this case), where in each epoch:
    - The model is set to training mode.
    - The training loss is computed by performing forward and backward passes through the model.
    - After training, the model evaluates its performance on the test dataset, calculating the validation loss.

4. Model Evaluation

- **Evaluation Function:** A function evaluate_model is defined to assess the model's performance on both training and testing datasets:
    - The model is set to evaluation mode, and predictions are collected.
    - Various metrics are calculated, including accuracy, precision, recall, and F1 score, which provide insights into the model's performance.

5. Model Saving

- After training and evaluation, the model's state dictionary is saved using the pickle module, allowing for future inference without needing to retrain.

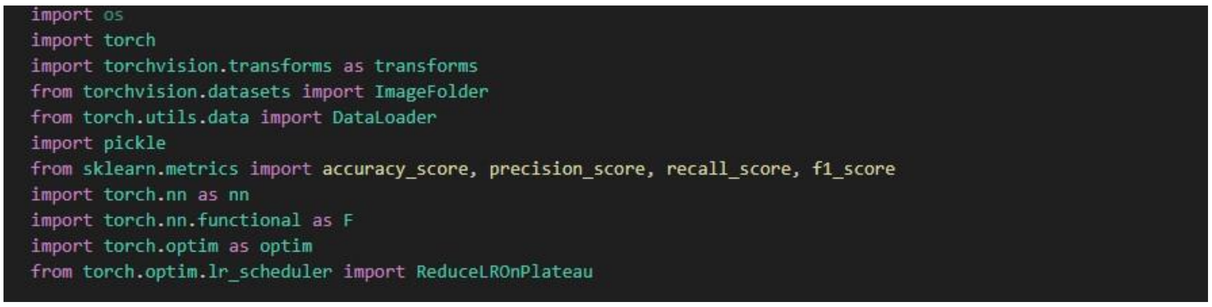
6. Results

- Finally, the metrics are printed for both training and testing datasets, giving a clear indication of the model's effectiveness in classifying emotions from images.

Objectives

- Develop and design an end-to-end solution for emotion detection from images.
- Implement a polished, user-friendly Streamlit application for image upload and emotion classification.
- Address potential applications in healthcare, education, and customer service where emotion detection is valuable.

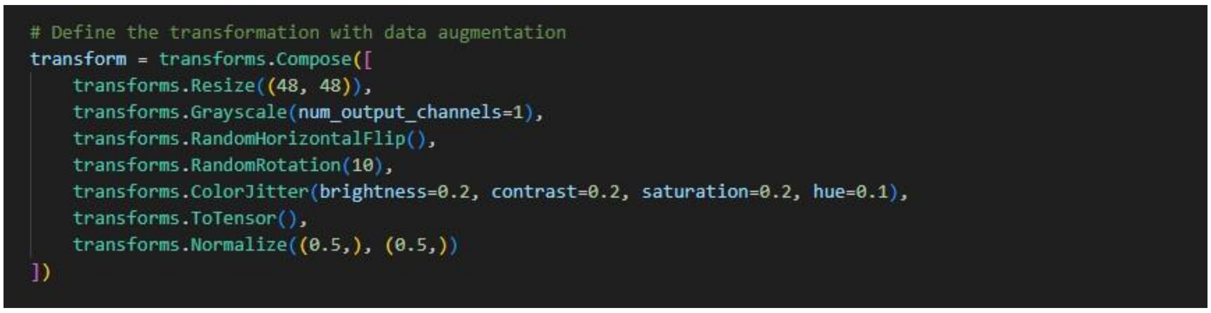
1. Import



```python
import os
import torch
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import pickle
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

- **os**: Module to interact with the operating system.
- **torch**: Main PyTorch package.
- **torchvision.transforms**: Common image transformations.
- **torchvision.datasets**: Provides a way to load datasets.
- **torch.utils.data.DataLoader**: Provides an iterable over a dataset.
- **pickle**: Used for serializing and deserializing Python objects.
- **sklearn.metrics**: Metrics to evaluate model performance.
- **torch.nn**: Provides neural network layers.
- **torch.nn.functional**: Provides functional operations.
- **torch.optim**: Provides optimization algorithms.
- **torch.optim.lr_scheduler**: Adjusts learning rate based on validation metrics.

2. Data Augmentation and Transformation

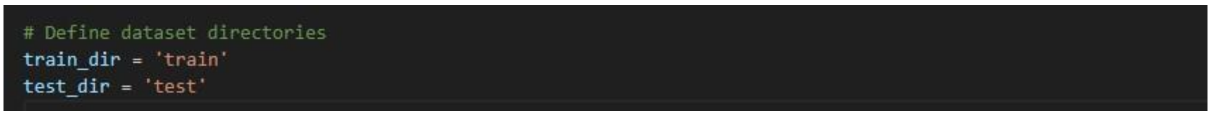


```python
# Define the transformation with data augmentation
transform = transforms.Compose([
    transforms.Resize((48, 48)),
    transforms.Grayscale(num_output_channels=1),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

- transforms.Resize: Resizes images to 48x48 pixels.
- transforms.Grayscale: Converts images to grayscale.
- transforms.RandomHorizontalFlip: Randomly flips images horizontally.
- transforms.RandomRotation: Randomly rotates images.
- transforms.ColorJitter: Randomly changes brightness, contrast, saturation, and hue.
- transforms.ToTensor: Converts images to PyTorch tensors.

- transforms.Normalize: Normalizes images to have mean 0.5 and standard deviation 0.5.

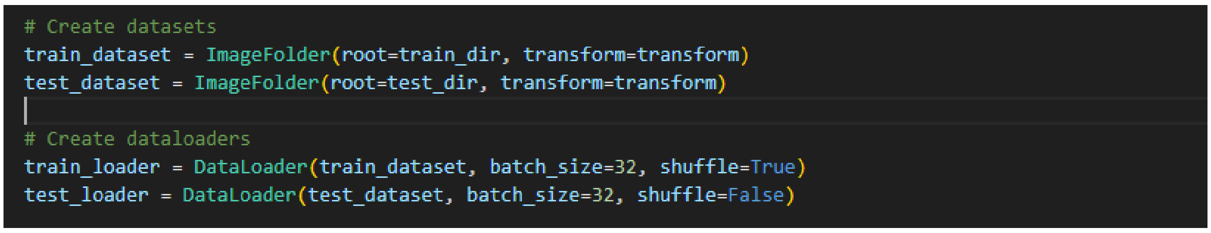
3. Dataset Directories



```python
# Define dataset directories
train_dir = 'train'
test_dir = 'test'
```

- Defines the directories where training and test images are stored.

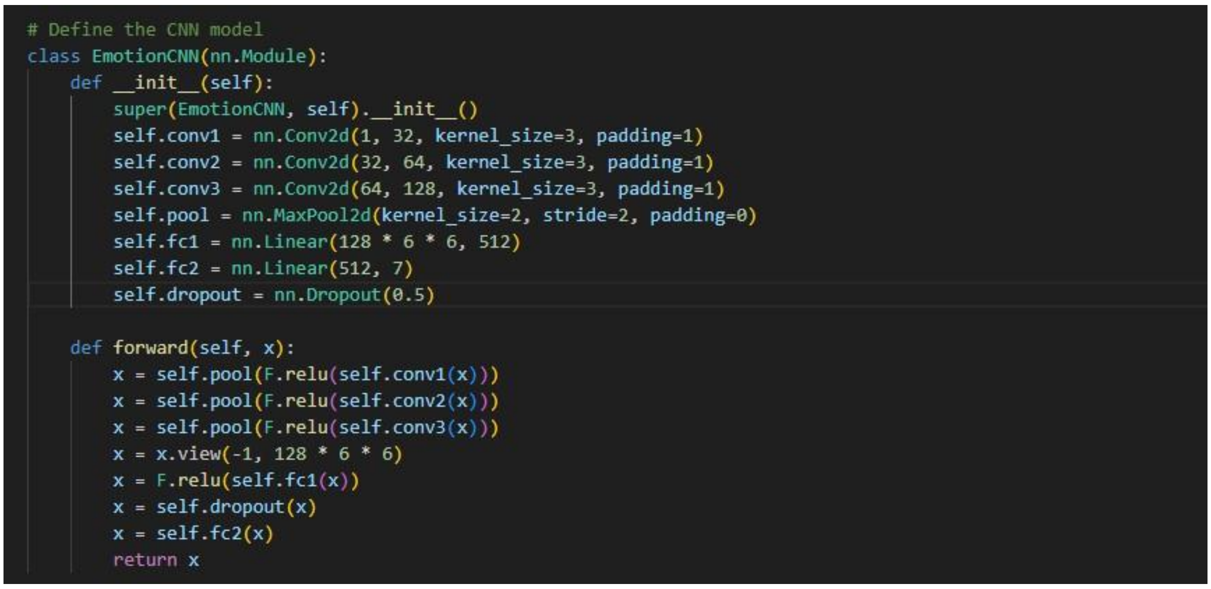
4. Create Datasets and DataLoaders



```python
# Create datasets
train_dataset = ImageFolder(root=train_dir, transform=transform)
test_dataset = ImageFolder(root=test_dir, transform=transform)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

- ImageFolder: Loads images from the specified directories and applies transformations.
- DataLoader: Creates iterable data loaders with specified batch size and shuffling.

5. Define the CNN Model



```python
# Define the CNN model
class EmotionCNN(nn.Module):
    def __init__(self):
        super(EmotionCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(128 * 6 * 6, 512)
        self.fc2 = nn.Linear(512, 7)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 128 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

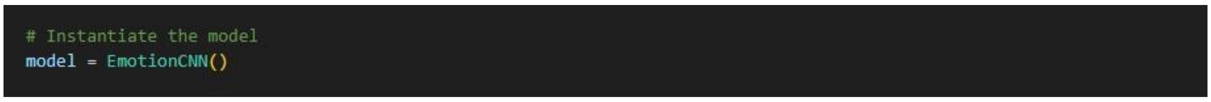
Emotion CNN: Defines the architecture of the CNN.

- conv1, conv2, conv3: Convolutional layers.

- pool: Max-pooling layer.

- fc1, fc2: Fully connected layers.
- dropout: Dropout layer for regularization.

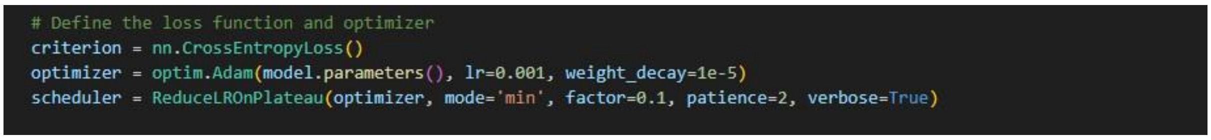
6. **Instantiate the Model**



```
# Instantiate the model
model = EmotionCNN()
```

- Creates an instance of the CNN model.

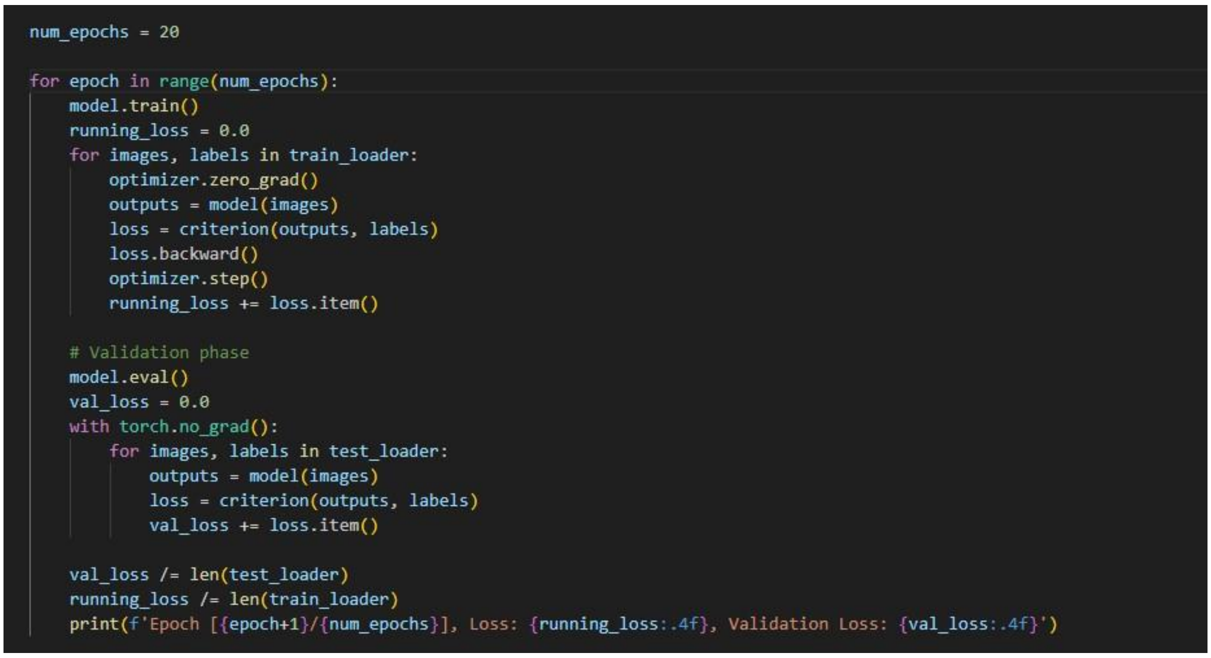
7. **Define Loss Function and Optimizer**



```
# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=2, verbose=True)
```

- criterion: Cross-entropy loss for classification.
- optimizer: Adam optimizer with learning rate and weight decay.
- scheduler: Reduces learning rate when a plateau in validation loss is detected.

8. **Training Loop**

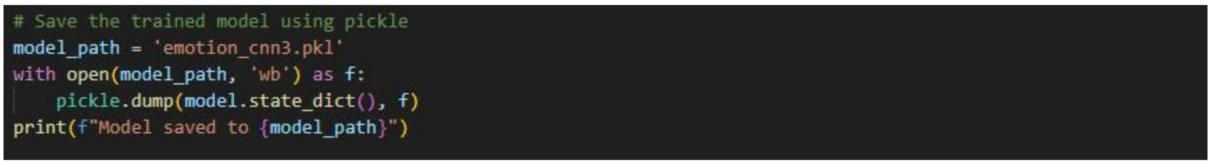


```
num_epochs = 20

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    # Validation phase
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()

    val_loss /= len(test_loader)
    running_loss /= len(train_loader)
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss:.4f}, Validation Loss: {val_loss:.4f}')
```

- model.train(): Sets the model to training mode.
- optimizer.zero_grad(): Clears previous gradients.
- outputs = model(images): Forward pass.
- loss = criterion(outputs, labels): Calculates loss.
- loss.backward(): Backward pass to calculate gradients.

- optimizer.step(): Updates model parameters.
- model.eval(): Sets the model to evaluation mode.
- with torch.no_grad(): Disables gradient calculation for validation.
- val_loss: Accumulates and averages validation loss.

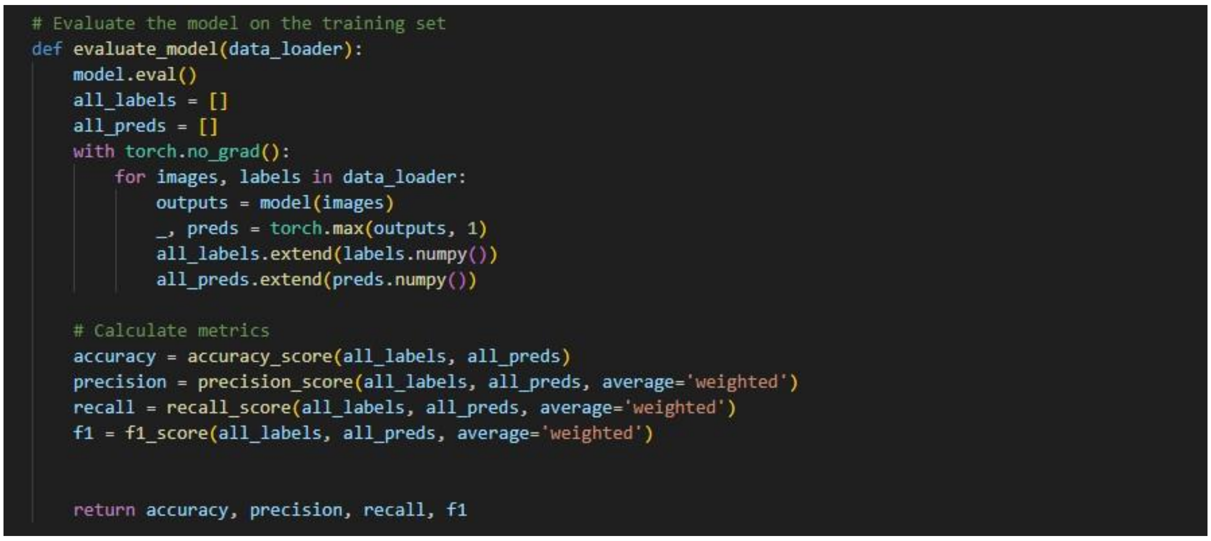
9. Save the Trained Model



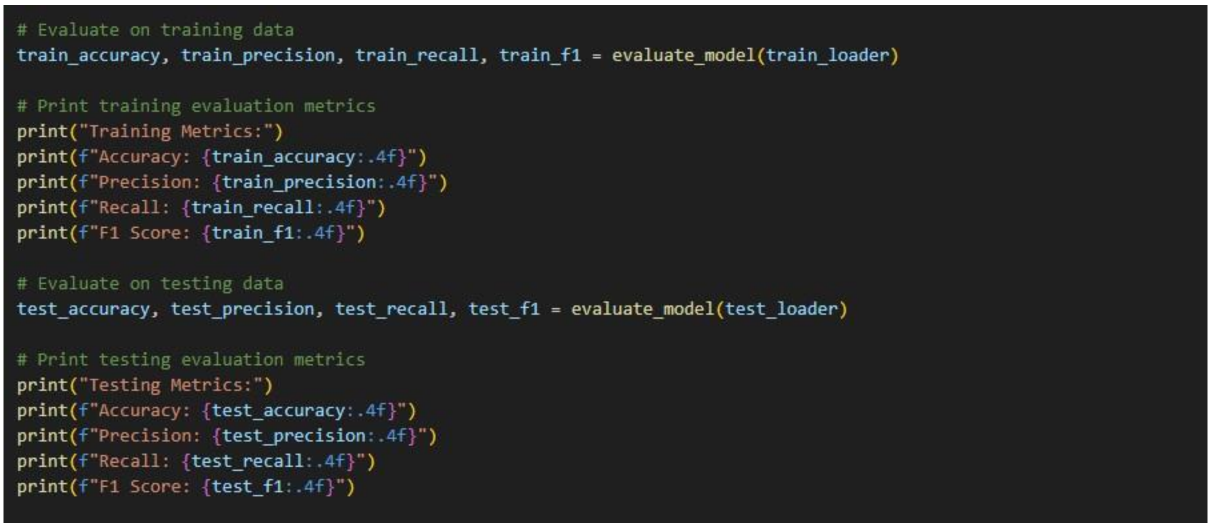
```python
# Save the trained model using pickle
model_path = 'emotion_cnn3.pkl'
with open(model_path, 'wb') as f:
    pickle.dump(model.state_dict(), f)
print(f"Model saved to {model_path}")
```

- model_path: Path to save the model.
- pickle.dump(): Saves the model state dictionary.

10. Evaluate the Model



```python
# Evaluate the model on the training set
def evaluate_model(data_loader):
    model.eval()
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in data_loader:
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.numpy())
            all_preds.extend(preds.numpy())

    # Calculate metrics
    accuracy = accuracy_score(all_labels, all_preds)
    precision = precision_score(all_labels, all_preds, average='weighted')
    recall = recall_score(all_labels, all_preds, average='weighted')
    f1 = f1_score(all_labels, all_preds, average='weighted')


    return accuracy, precision, recall, f1
```



```python
# Evaluate on training data
train_accuracy, train_precision, train_recall, train_f1 = evaluate_model(train_loader)

# Print training evaluation metrics
print("Training Metrics:")
print(f"Accuracy: {train_accuracy:.4f}")
print(f"Precision: {train_precision:.4f}")
print(f"Recall: {train_recall:.4f}")
print(f"F1 Score: {train_f1:.4f}")

# Evaluate on testing data
test_accuracy, test_precision, test_recall, test_f1 = evaluate_model(test_loader)

# Print testing evaluation metrics
print("Testing Metrics:")
print(f"Accuracy: {test_accuracy:.4f}")
print(f"Precision: {test_precision:.4f}")
print(f"Recall: {test_recall:.4f}")
print(f"F1 Score: {test_f1:.4f}")
```

- evaluate_model(): Evaluates the model on a given data loader.

- torch.max(outputs, 1): Gets predicted labels.
- accuracy_score(), precision_score(), recall_score(), f1_score(): Calculate evaluation metrics.

The evaluation results for both training and testing datasets are printed. This provides insight into the model's performance on both seen and unseen data.

Conclusion:

This implementation shows how to create a custom CNN for detecting emotions in images. It focuses on important steps like preparing the data, designing the model, training it, and measuring its performance. By using techniques like data augmentation and dropout, the model aims to perform well and be reliable in real-world situations.