

应用程序开发过程（前期）

在本次应用程序开发过程中，前期工作主要集中在 **Jetson Nano B01 开发板** 的组装与环境配置上，以确保开发平台能够稳定运行为后续开发提供必要支持。

1.开发板的组装与硬件连接

我们首先完成了 **Jetson Nano B01 开发板** 的组装工作，主要包括设备线路连接、机箱的组装以及主要硬件设备（如 **USB 摄像头**）的安装。安装过程中，我们确保所有硬件设备的连接稳定，特别是 **USB 摄像头** 的安装和与开发板的连接，以便能够后续进行实时视频流的处理。完成硬件连接后，我们对 **SD 卡** 进行了格式化、分区操作，并将操作系统镜像文件烧录到 **SD 卡** 中，确保 **Jetson Nano** 上的操作系统和系统工具能够正常运行。此步骤保证了开发板的基本启动和运行环境的搭建。

2.系统安装与硬件设备验证

镜像烧录完毕后，我们进行了基础的系统安装，确保操作系统能够顺利启动并运行。接着，我们通过命令行查看硬件设备信息，包括处理器、内存、摄像头等设备是否正确识别。我们特别注意检查 **Jetson Nano B01** 提供的计算机视觉库是否成功导入，确保 **OpenCV** 和其他基础库能够正常使用，为后续的图像处理与深度学习任务提供支持。

3.配置开发环境与工具

Jetson Nano B01 开发板 本身并未提供集成开发环境，虽然它提供了编译器，但在控制台开发效率较低。因此，我们选择了 **PyCharm**

作为集成开发环境（IDE）进行开发。在安装 PyCharm 前，我们首先需要安装 JDK 11，这是因为在没有安装 JDK 的情况下，通过命令行执行 `./pycharm.sh` 无法启动 PyCharm。我们在 Jetson Nano 上安装了 JDK 11，随后成功安装并配置了 PyCharm 2020 版本，确保其能够顺利运行并进行项目开发。

4.确定开发任务与功能配置

在完成系统基础配置和开发环境搭建后，我们明确了开发目标，项目的核心任务是实现基于 Jetson Nano B01 开发板的表情识别嵌入式开发。主要功能包括面部识别、表情识别、疲劳提醒以及报告生成等。为了实现这些功能，我们还需要额外安装一些库和工具，这些库包括用于 前端设计 的 PyQt5，以及用于人脸识别的 Facenet_PyTorch 库。我们成功安装并配置了这些库，为后续的系统开发和调试提供了坚实的基础。

经过前期的硬件组装、系统安装、开发环境配置以及库文件的安装，我们已经为下一阶段的表情识别系统开发做好了充分准备，系统可以开始进行代码编写与功能实现。

应用程序开发过程（中期）

1.人脸识别算法的选定

在应用程序的第一阶段，我们选择了 `facenet_pytorch` 库作为人脸识别的算法工具。该库基于深度学习模型，能够高效地检测和识别用户的面部区域，并且已经预先训练好了人脸识别模型。我们通过测试代码调用了 `facenet_pytorch` 库的预训练模型，初步验证了其人脸识别的基本功能。测试结果显示，该模型能够成功检测到用户的面部区域，尽管如此，我们在实际运行过程中发现了一些显著的性能问题，影响了用户体验。

首先，程序启动时间过长。在运行程序时，摄像头虽然已经被打开，但程序的窗口却未能及时弹出，这导致用户在等待程序加载时感到不便。其次，即便程序窗口成功弹出，运行过程中界面表现出严重的卡顿现象，尤其是在摄像头实时采集图像时，帧率下降明显，导致画面失帧严重，这不仅影响了程序的响应速度，也给用户带来了不好的使用体验。我们意识到，虽然 `facenet_pytorch` 的人脸检测功能强大，但由于其背后的深度学习模型需要较大的计算资源，导致在本地硬件上运行时计算负担过重，尤其是结合摄像头采集实时图像的场景中，性能瓶颈更为明显。

针对这一问题，我们决定优化人脸检测的部分，考虑到 `facenet_pytorch` 对计算资源的消耗较大，我们尝试更换为 Haar 特征级联人脸检测 算法（`haarcascade_frontalface_default.xml` 配置文件）。这种方法相比深度学习模型计算量较小，并且已被广泛应用于计算机

视觉领域，能在实时处理上提供较好的性能。

然而，更换算法后，问题并未完全得到解决。在我们对 Haar 级联算法进行测试时，虽然它在性能上有所提升，但仍然存在一定的卡顿现象，特别是在摄像头图像传输的过程中。经过仔细检查，我们发现瓶颈并非完全出现在算法本身，而是出在摄像头的配置和参数设置上。我们分析并修改了 OpenCV 中调用摄像头部分的配置参数，包括分辨率和帧率的设置，使其更适应 Jetson Nano B01 的硬件环境。

调整后，我们重新测试了摄像头的捕捉效果，结果表明，摄像头的画面捕捉流畅，帧率得到了显著提升，卡顿问题得到有效缓解。经过优化后，程序的响应速度得到了提升，用户体验也大大改善，界面表现更加流畅。此时，我们确保了程序在使用人脸识别算法时，能够稳定运行，并为后续集成表情识别模型提供了更好的基础。

通过这次优化，我们验证了 Haar 特征级联 检测与摄像头参数调整的有效性，成功提升了系统的整体性能，为实现高效的表情识别打下了坚实的基础。接下来，我们将进一步测试和优化表情识别部分，确保系统在 Jetson Nano B01 平台上能够顺畅运行。

```
from facenet_pytorch import MTCNN
import torch
import cv2

# 检查 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# 使用 MTCNN 检测人脸
mtcnn = MTCNN(keep_all=True, device=device)

# 打开视频流
```

```

video_capture = cv2.VideoCapture(0)

while True:
    ret, frame = video_capture.read()
    if not ret:
        break

    # 转换为 RGB 格式（MTCNN 需要）
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # 检测人脸
    boxes, _ = mtcnn.detect(rgb_frame)

    # 绘制检测框
    if boxes is not None:
        for box in boxes:
            x1, y1, x2, y2 = [int(coord) for coord in box]
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

    # 显示视频
    cv2.imshow('Video', frame)

    # 按 'q' 退出
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

video_capture.release()
cv2.destroyAllWindows()

```

2.表情识别模型模型选定

在本项目中，表情识别部分采用了 卷积神经网络（CNN）模型。CNN 是一种深度学习模型，专门用于从图像中自动提取特征并进行分类。在面部表情识别任务中，CNN 能够通过多层卷积、池化和全连接层，自动学习和识别面部表情的关键特征。

模型结构与原理：

卷积层（Convolutional Layer）： 卷积层通过多个卷积核对输入图像进行特征提取。每个卷积核会通过滑动窗口的方式在图像上进行卷

积运算，提取图像中的边缘、纹理等基本特征。在本模型中，我们使用了三次卷积池化操作，逐步提取图像的低级到高级特征，帮助模型识别出眼睛、嘴巴等面部特征。

池化层（Pooling Layer）： 池化层的作用是对卷积后的特征图进行下采样，减少特征的空间维度，降低计算量，并增强模型的鲁棒性。通过最大池化（Max Pooling），模型能够保留图像中的重要信息，减少噪声，并防止过拟合。

全连接层（Fully Connected Layer）： 在卷积层和池化层处理过的特征图之后，数据会被展平并传入全连接层进行分类。全连接层的任务是将提取到的特征映射到表情类别上，通过一系列非线性变换输出 8 个表情类别的预测结果。

训练与准确率：

我们使用的数据集包含 55338 张图片，涵盖了多种不同的面部表情（如开心、悲伤、愤怒等）。经过训练，本模型在该数据集上的准确度接近 90%。这种较高的准确率使得我们能够有效地识别出不同的表情类别，满足项目中实时识别的需求。

```
class FaceCNN(nn.Module):
    # 初始化网络结构
    def __init__(self):
        super(FaceCNN, self).__init__()

        # 第一次卷积、池化
        self.conv1 = nn.Sequential(
            # 输入通道数 in_channels，输出通道数(即卷积核的通道数)out_channels，卷积核大小 kernel_size，步长 stride，对称填 0 行列数 padding
            # input:(batch_size, 1, 48, 48), output:(batch_size, 64, 48, 48), (48-3+2*1)/1+1 = 48
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1),
            # 卷积层
```

```

        nn.BatchNorm2d(num_features=64), # 归一化
        nn.RReLU(inplace=True), # 激活函数
        # output:(batch_size, 64, 24, 24)
        nn.MaxPool2d(kernel_size=2, stride=2), # 最大值池化
    )

# 第二次卷积、池化
self.conv2 = nn.Sequential(
    # input:(batch_size, 64, 24, 24), output:(batch_size, 128, 24, 24), (24-3+2*1)/1+1 =
24
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1,
padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.RReLU(inplace=True),
    # output:(batch_size, 128, 12, 12)
    nn.MaxPool2d(kernel_size=2, stride=2),
)

# 第三次卷积、池化
self.conv3 = nn.Sequential(
    # input:(batch_size, 128, 12, 12), output:(batch_size, 256, 12, 12), (12-3+2*1)/1+1 =
12
    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1,
padding=1),
    nn.BatchNorm2d(num_features=256),
    nn.RReLU(inplace=True),
    # output:(batch_size, 256, 6, 6)
    nn.MaxPool2d(kernel_size=2, stride=2),
)

# 参数初始化
self.conv1.apply(gaussian_weights_init)
self.conv2.apply(gaussian_weights_init)
self.conv3.apply(gaussian_weights_init)

# 全连接层
self.fc = nn.Sequential(
    nn.Dropout(p=0.2),
    nn.Linear(in_features=256*6*6, out_features=4096),
    nn.RReLU(inplace=True),
    nn.Dropout(p=0.5),
    nn.Linear(in_features=4096, out_features=1024),
    nn.RReLU(inplace=True),
    nn.Linear(in_features=1024, out_features=256),

```

```

        nn.ReLU(inplace=True),
        nn.Linear(in_features=256, out_features=8),      #7 分类
    )

# 前向传播
def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    # 数据扁平化
    x = x.view(x.shape[0], -1)
    y = self.fc(x)
    return y

```

3.疲惫提醒算法：

这部分是建立在能够识别疲惫的基础上，但是如果直接检测到疲惫就提醒用户，这显然不合理，所以我们采用了一种类似滑动窗口方法，统计一段时间内窗口种的疲惫状态个数。如果疲惫状态个数所占的比率达到阈值（这里阈值不能太大也不能太小，过小失去检测意义，过大难以检测疲惫状态），可以认为用户处于疲惫状态，并发出提醒。

核心思想是不断统计队列种疲惫状态的比例。具体算法如下：

```

if emotion_mode == 7: # 疲惫状态
    # print(ui.switchButton_tired_opener.state)
    cnt1 += 1
    arr[cnt] = 1
    cnt += 1
    print("\n rate:", float(cnt1 / cnt))
    print("cnt  cnt1", cnt, cnt1)
    if cnt == 20: # 队列满了进行判断，判断疲惫状态所占的比例是否达到阈值
        if float(cnt1 / cnt) > 0.75 and ui.switchButton_tired_opener.state: # 前提打开疲惫提醒功能
            tip1()
            cnt = 0 # 达到阈值重新计数
            cnt1 = 0
            continue
    v = arr[0]
    for i in range(0, 19): # 循环移进，队首出队

```



```

        arr[i] = arr[i + 1]
    if (v == 1): # 如果移出的状态是疲惫状态，那么疲惫状态个数减一
        cnt1 -= 1
    cnt -= 1
else: # 非疲惫状态计数
    arr[cnt] = 0
    cnt += 1
    print("cnt  cnt1", cnt, cnt1)
    print("\n rate:", float(cnt1 / cnt))
    if cnt == 20:
        if float(cnt1 / cnt) > 0.75 and ui.switchButton_tired_opener.state:
            tip1()
            cnt = 0
            cnt1 = 0
            continue
    v = arr[0]
    for i in range(0, 19):
        arr[i] = arr[i + 1]
    if (v == 1):
        cnt1 -= 1
    cnt -= 1

```

4. PyQt 界面开发：

在应用程序的开发过程中，用户界面是不可忽视的一部分。为了提供一个直观、易用的界面，我们选择了 **PyQt5** 作为图形界面开发框架。**PyQt5** 是一个跨平台的图形界面框架，能够帮助我们快速构建具有丰富交互功能的桌面应用程序。以下是界面开发的具体步骤和实现细节：

界面布局设计：

为了确保用户能够高效地使用系统，我们设计了多个界面模块，包括实时视频流显示、情绪识别结果展示、系统设置以及报告生成等功能。界面的布局分为以下几个主要部分：

(1)视频显示区域：这是界面的核心部分，实时展示来自摄像头的

视频流。用户可以看到自己的面部图像，并且程序会在图像中实时显示出检测到的人脸区域。

(2)情绪识别结果显示：在视频显示区域的旁边，我们设计了一个区域用于展示当前检测到的面部表情。该区域会动态更新，显示出用户的当前情绪（例如：生气、开心、疲惫等）。

(3)系统控制面板：包括几个按钮和滑动条，用于用户调整设置，例如启用或禁用提醒功能、设置提醒时间等。用户可以根据自己的需求设置系统提醒。

(4)报告生成按钮：此按钮点击后，系统会生成并显示用户的情绪报告，包括在当前会话中识别到的情绪类型和每种情绪的持续时间。

在设计过程中，我们采用了现代化的界面布局和颜色搭配，使得系统更加美观、易用。同时，我们还在界面中加入了不同的状态提示，如“正在识别”、“识别完成”等，以使用户清楚了解系统的当前状态。

主题色和控件设计：我们选用了简洁、清晰的配色方案，避免过于复杂的设计，确保用户能够专注于系统的核心功能。

提示信息：在用户操作时，系统会适时弹出提示信息（如“请调整摄像头位置”），确保用户能够顺利使用系统。

在开发过程中，我们对各个功能模块进行了多次测试，确保各项功能的稳定性与流畅性。测试结果表明，视频流显示、情绪识别结果更新、提醒功能等均能稳定运行，用户交互体验也得到了优化。

通过 PyQt5 开发的界面，我们实现了一个既直观又实用的用户界面。用户可以方便地查看实时的情绪识别结果，调整系统设置，并

生成个性化的情绪报告。这为后续表情识别和提醒功能的实现提供了坚实的基础，也为用户提供了一个流畅且友好的操作体验。

应用程序开发过程（后期）

在完成核心部分算法的开发之后，接下来进行程序的测试工作，主要包括模型的准确率测试、功能执行测试以及整体系统性能的验证。测试内容主要涵盖以下几个方面：

1. 模型准确率测试

本系统的核心目标之一是通过卷积神经网络（CNN）进行表情识别，识别 8 种常见情绪表情：生气、厌恶、沮丧、害怕、开心、正常、惊喜和疲惫。在测试过程中，我们首先验证了系统对这 8 种表情的识别准确性。具体来说，我们通过对模型进行训练和验证，确保其能够准确地对每种表情进行分类。

测试方法：我们使用了包含 55338 张图片的数据集，其中包括各种不同情绪状态的用户面部图像。经过训练后的模型，在测试集上达到了 90% 以上的准确率，能够识别并分类生气、厌恶、沮丧、害怕、开心、正常、惊喜和疲惫这 8 种表情。

测试结果：测试结果表明，系统能够准确识别这 8 种表情，并且准确度达到了设计预期，满足了用户需求。

2. 提醒功能测试

本系统包含了两种重要的提醒功能：休息提醒和疲惫提醒，这两项功能旨在帮助用户合理安排工作时间，减少长时间使用电脑带来的身心问题。为了验证这些功能的正常运行，我们进行了如下测试：

（1）休息提醒功能测试：

测试方法：首先，用户需要在系统设置界面中开启休息提醒功能，并设置提醒时长（例如，工作 60 分钟后进行提醒）。系统会根据用户的设置，记录并计算当前的工作时长，确保在设定时间达到时，提醒用户休息。

测试结果：系统能够根据用户设定的时长准确地进行提醒（如设置 60 分钟后提醒，系统能够在规定时间到达后及时弹出提醒），证明该功能可以正常执行。

（2）疲惫提醒功能测试：

测试方法：在进行疲惫提醒功能测试时，我们首先确保启用了疲惫状态检测功能。当用户在使用过程中表现出疲惫的面部表情时，系统会根据设定的疲惫状态阈值，自动检测并记录疲惫状态的频率。

测试结果：当疲惫状态在滑动窗口内的比例超过设定的阈值时，系统能够及时提醒用户“疲惫，请休息”。这一功能也得到了验证，并且能够在实际使用中准确执行。

3. 情绪报告生成功能测试

情绪报告是系统的一项附加功能，旨在帮助用户了解自己在使用电脑过程中的情绪变化。系统通过记录每次打开摄像头后的情绪状态，统计每种表情的持续时间，生成详细的情绪报告。

测试方法：开启摄像头并运行系统，记录用户在一定时间内表现出的情绪状态（如生气、开心、疲惫等），系统将自动记录各情绪的出现时间和持续时长。

测试结果：情绪报告功能能够准确地统计用户情绪状态的持续时

间，并根据不同的情绪类别生成报告。用户能够通过报告直观地看到自己在一段时间内的情绪变化趋势，并能够根据报告调整自己的使用习惯或工作安排。

4. 功能测试总结

通过以上测试，我们验证了系统的核心功能和附加功能的正常运行，确保了每项功能都能按预期执行。具体结果如下：

（1）表情识别准确率：系统能够准确地识别 8 种表情，并且准确度达到了 90% 以上，符合设计预期。

（2）休息提醒和疲惫提醒功能：系统能够按照用户设置的时间准确提醒用户休息，并且能有效检测疲惫状态并及时提醒用户。

（3）情绪报告生成：系统能够记录用户情绪的持续时间，并生成准确的情绪报告，帮助用户了解自己的情绪变化。

至此，所有核心功能和附加功能的测试均已完成，并且系统表现稳定，达到了设计目标。