数据集说明

本项目涉及两个主要数据集：Face2 和 Face3。

1.Face2 数据集

来源于 Kaggle，链接为 Deepfake and Real Images Dataset。该数据集包含两类图像：一类为真实人脸图像（Real），通过实际拍摄获得；另一类为伪造人脸图像（Fake），由深度伪造技术生成。

2.Face3 数据集

由伪造数据（Fake）和真实数据（Real）两部分组成，分别来源于 PGGAN 和 CelebA 数据集。

Fake 数据由 PGGAN 模型生成，具体生成过程包括从低分辨率（如 4×4）开始，通过渐进式增长的训练方法逐步增加生成器和判别器的网络层，最终生成高分辨率（如 1024×1024）的伪造人脸图像。生成过程中，PGGAN 使用 WGAN-GP 损失函数，结合小批量标准差（Minibatch Std）和正则化策略（如 Pixel Norm 和 Equalized Learning Rate），以提升生成图像的多样性和稳定性。生成的数据分为训练集（Train）、验证集（Valid）和测试集（Test），可通过以下链接获取相关资源：PGGAN 数据。

Real 数据来源于 CelebA 数据集，文件链接为 CelebA 数据（文件名为 `img_align_celeba_png.7z`）。CelebA 是一个大规模人脸数据集，包含 202,599 张人脸图像，涉及 10,177 个身份，并为每张图片提供 5 个地标位置和 40 个二进制属性注释。该数据集具有较大的姿势变化和复杂的背景，经过调整后，提取了图像中的人脸区域，并剔除了大部分人脸以外的信息。根据项目需求，Real 数据集进一步划分为三部分：测试集（Test）包含 CelebA 中最后 10,000 张图片（索引范围：192,600-202,599）；验证集（Valid）包含测试集以外的倒数 20,000 张图片（索引范围：172,600-192,599）；训练集（Train）包含剩余的所有图片（索引范围：1-172,599）。

环境配置

Python版本：3.8.20

PyTorch版本：2.4.1

预备软件：VScode（插件：Python、PythonDebugger）、Anaconda

1. 打开AnacondaPrompt，输入：

```
conda create -n pytorch_env python=3.8.20
```

创建一个名称为pytorch_env的环境。

2. 在AnacondaPrompt中输入：

```
conda install pytorch torchvision torchaudio -c pytorch
```

安装PyTorch。

3. 打开VS Code：

按下 `Ctrl + Shift + P` 打开命令面板，输入并选择 Python: SelectInterpreter 。 选择刚刚创建的pytorch_env环境的Python解释器。

4. 测试是否安装成功：

在VS Code中输入：

```
import torch
```

```
print(torch.__version__)
```

输出：2.4.1

配置成功。

构建与使用 ModelTools

---

1. 硬件设备检测

本模块的第一步是检测硬件环境，明确是否支持 GPU 加速，并获取设备的详细参数。这对于确定深度学习模型训练和推理的运行环境至关重要。

1.1 获取 CPU 信息

该部分通过 `cpuinfo` 库获取 CPU 的品牌、架构、频率和线程数等信息，并格式化输出。

```
info = cpuinfo.get_cpu_info()
print(f"CPU version: {info['brand_raw']}")
print("{:<8}\t{:<5}\t{:<10}\t{:<8}".format("Architecture", "Bits",
"Frequency", "Thread count"))
print("{:<13}\t{:<4}\t{:<13}\t{:<8}".format(info['arch'], info['bits'],
info['hz_actual_friendly'], info['count']))
```

1.2 检测 GPU 可用性

通过 `torch.cuda.is_available()` 检测是否有可用 GPU。如果有，则获取 GPU 名称、显存大小、CUDA 支持版本等详细信息。

```
if torch.cuda.is_available():
    properties = torch.cuda.get_device_properties(torch.cuda.current_device())
    print(f"GPU version: {properties.name}")
    print(f"Total memory: {properties.total_memory / 1024 / 1024} MB")
```

1.3 返回设备对象

最终返回可用的设备对象 (`torch.device`)，用于后续模型操作。

```
return torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
```

---

2. 数据加载与预处理

深度学习模型的训练依赖于高质量的数据集加载和预处理。`datasets_load` 函数负责从磁盘读取数据集并应用转换，生成用于训练、验证、测试的数据加载器。

2.1 定义数据集结构

通过输入的根目录、数据集子目录、数据类型和标签，初始化数据存储结构。

```
datasets = {t: [] for t in data_type}
```

2.2 加载数据集

遍历指定路径，将每个数据类型的子数据集加载到内存中，支持自定义预处理方式。

```
for i in datasets_dir:
    for j in data_type:
        path = os.path.join(root_dir, i, j)
        for k in classes:
            datasets[j].append(MyDataset(path, k,
transform=data_transforms[j]))
```

2.3 数据加载器的创建

将所有子数据集合并为一个 `ConcatDataset` 对象，并生成对应的数据加载器，用于批量处理数据。

```
image_datasets = {x: ConcatDataset(datasets[x]) for x in data_type}
dataloaders = {x: DataLoader(image_datasets[x], batch_size=batch,
shuffle=True) for x in data_type}
```

2.4 返回加载器和数据集大小

函数最终返回包含数据加载器和数据集大小的字典，供模型训练和评估使用。

```
return dataloaders, dataset_size
```

---

3. 模型加载与创建

模块支持加载预训练模型、从零创建新模型，并能动态调整模型结构以适应任务需求。

3.1 动态加载模型

根据用户指定的模型名称，通过 `globals()` 动态查找模型类，并实例化。

```
model = globals()[model_name]()
```

3.2 加载预训练参数

通过 `torch.load` 加载之前保存的模型权重、优化器状态和训练历史记录。

```
state = torch.load(model_path, map_location=device)
model.load_state_dict(state['state_dict'])
optimizer_state_dict = state['optimizer']
```

3.3 模型结构调整

对于特定模型（如 `Vgg16`），调整其最后的分类层以适应二分类任务。

```
if model_name == 'Vgg16':
    model = torchvision.models.vgg16(weights=torchvision.models.VGG16_Weights.DEFAULT)
    model.classifier[-1] = nn.Linear(in_features=4096, out_features=2)
```

---

4. 模型性能评估

模型性能评估是深度学习开发的重要环节，包含计算模型的计算量（FLOPs）、参数量、推理时间和吞吐量。

4.1 计算 FLOPs 和参数量

使用 `thop` 库计算模型的 FLOPs 和参数量，以量化模型的复杂度。

```
inputs = torch.randn(batch_size, 3, 224, 224, dtype=torch.float).to(device)
flops, params = thop.profile(model, inputs=(inputs,))
print(f"FLOPs: {flops / 1e9}G, Params: {params / 1e6}M")
```

4.2 测试推理速度

多次运行模型前向传播，测量平均推理时间（毫秒）。

```
timings = []
for _ in range(repetitions):
    start = time.time()
    _ = model(inputs)
    end = time.time()
    timings.append(end - start)
mean_time = np.mean(timings)
print(f"Mean Inference Time: {mean_time:.3f}ms")
```

4.3 计算吞吐量

根据总推理时间和批量大小计算模型吞吐量（每秒处理的样本数）。

```
throughput = (repetitions * batch_size) / total_time
print(f"Throughput: {throughput:.2f} samples/second")
```

---

5. 混淆矩阵可视化

混淆矩阵是一种重要工具，用于评估模型分类性能，展示预测值与真实值的匹配情况。

5.1 生成混淆矩阵

使用 `sklearn.metrics.confusion_matrix` 生成混淆矩阵，并计算准确率、召回率等指标。

```
self.matrix = metrics.confusion_matrix(self.y_true, self.y_pred)
accuracy = 100 * (tp + tn) / total
```

5.2 绘制混淆矩阵

通过 `matplotlib` 将混淆矩阵可视化，支持显示原始值或百分比。

```
plt.imshow(self.matrix, cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
```

5.3 归一化矩阵

支持按行归一化显示比例值，以便于直观比较分类效果。

```
matrix = matrix.astype('float') / matrix.sum(axis=1)[:, np.newaxis]
```

训练模块实现与说明

---

1. 模型训练步骤

1.1 环境配置

在训练开始前，对硬件和算法加速进行配置：

- 启用 CuDNN 自动优化以加速 GPU 训练。
- 配置非确定性模式，允许更高的运行效率。

```
torch.backends.cudnn.benchmark = True
torch.backends.cudnn.deterministic = False
```

1.2 数据加载与增强

加载训练集和验证集，并对训练数据进行增强以提高模型的泛化能力：

- 通过多种增强方式（如随机旋转、水平翻转、色彩变换等）生成多样化的训练样本。
- 验证数据仅进行基础的大小调整。

```
data_transforms = {
    'Train': transforms.Compose([...]),  # 包括旋转、翻转、颜色调整等操作
    'Valid': transforms.Compose([...])   # 仅调整尺寸
}
loaders, dataset_size = datasets_load(root_dir, datasets_dir, data_type,
classes, batch, data_transforms)
```

1.3 模型加载与创建

根据 `pre_epochs` 参数，判断是从头创建模型还是加载预训练模型：

- **从头创建**：通过 `model_create` 函数创建指定模型。
- **加载预训练模型**：通过 `model_load` 函数加载之前训练的模型权重和优化器状态。

```
if pre_epochs is None:
    model = model_create(model_name, device)
else:
    model, best_acc, optimizer_state_dict, history = model_load(...)
```

1.4 优化器配置

根据用户选择的优化器名称，初始化优化器：

- **Adam 优化器**：适合大多数模型，能够自适应调整学习率。
- **SGD 优化器**：通过固定学习率进行随机梯度下降。

```
if optimizer_name == 'Adam':
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
else:
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

---

2. 训练与验证

2.1 损失函数选择

根据模型类型选择损失函数：

- 如果最后一层包含 `LogSoftmax`，使用 `NLLLoss`。
- 否则使用 `CrossEntropyLoss`。

```
if model_name == 'Cnn':
    criterion = nn.NLLLoss().to(device)
else:
    criterion = nn.CrossEntropyLoss().to(device)
```

2.2 训练与验证流程

循环进行训练和验证，分阶段执行以下步骤：

- **训练阶段**：模型设置为 `train` 模式，启用梯度计算；进行前向传播、损失计算、反向传播和优化。
- **验证阶段**：模型设置为 `eval` 模式，禁用梯度计算；仅进行前向传播和损失计算。

```
for phase in data_type:
    if phase == 'Train':
        model.train()
    else:
        model.eval()
    for images, labels in loaders[phase]:
        optimizer.zero_grad()  # 清空梯度
        outputs = model(images)
        result_loss = criterion(outputs, labels)
        if phase == 'Train':
            result_loss.backward()
            optimizer.step()
```

2.3 动态学习率调整

使用余弦退火学习率调度器动态调整学习率，以提高收敛效果。

```
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=20)
scheduler.step()
```

---

3. 模型保存

在每轮训练结束时保存模型：

- 记录最佳验证集准确率时，保存模型权重及优化器状态。

- 模型以 `模型名称_优化器_轮次数.pt` 命名。

```
state = {
    'state_dict': model.state_dict(),
    'best_acc': best_acc,
    'optimizer': optimizer.state_dict(),
    'train_acc_history': train_acc_history,
    'valid_acc_history': valid_acc_history,
    ...
}
model_save(model_name, state, optimizer_name, n_epochs + pre_epochs)
```

---

4. 输出结果

在训练完成后，输出以下关键信息：

- **总耗时**：包括训练和验证所有轮次的总时长。
- **最佳准确率**：在验证集上达到的最高准确率。

```
total_time = end_time - start_time
print(f'Total time: {minutes:.0f} minutes {seconds:.1f} seconds.')
print(f'Best accuracy: {best_acc:.4f}')
```

---

5. 主函数调用

用户通过 `__main__` 主函数调用 `train`，并指定模型名称、优化器、学习率等参数进行训练。例如：

- 使用 `efficientnet_b0` 模型。
- 选择 `Adam` 优化器，学习率为 0.01。
- 在 GPU 上训练 3 轮。

```
if __name__ == "__main__":
    train(d='cuda', model_name='efficientnet_b0', optimizer_name='Adam',
learning_rate=0.01, n_epochs=3)
```

测试模块实现与说明

1. 模型测试步骤

1.1 数据加载

在测试开始前，加载测试数据集并进行预处理。

- **数据集配置**：定义测试集目录和标签名称。
- **数据预处理**：对测试数据进行基础操作（如调整尺寸、张量转换）。

```
root_dir = "data"
datasets_dir = ["Face2", "Face3"]
data_type = ["Test"]
classes = ["Fake", "Real"]
data_transforms = {'Test': transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])}
loaders, dataset_size = datasets_load(root_dir, datasets_dir, data_type,
classes, batch, data_transforms)
```

```
10 test_loader = loaders["Test"]
```

1.2 加载模型

通过指定设备名称和预训练模型的轮次数，加载模型的权重和优化器状态：

- **设备检测**：使用 `get_device` 确认测试运行的设备。
- **模型加载**：调用 `model_load` 函数加载已保存的模型及相关状态。

```
1 device = get_device(d)
2 model, best_acc, optimizer_state_dict, history = model_load(model_name,
  device, d, optimizer_name, pre_epochs)
```

---

2. 测试过程

2.1 前向推理

测试模型在每个测试样本上的预测结果，记录以下内容：

- `y_true`：真实标签。
- `y_pred`：模型预测标签。
- `correct` 和 `**total**`：计算分类准确率。

```
1 with torch.no_grad():
2     for test_images, test_labels in test_loader:
3         outputs = model(test_images.to(device))
4         _, predicted = torch.max(outputs, 1)
5         y_pred.extend(predicted.cpu().numpy())
```

```
        y_true.extend(test_labels.cpu().numpy())
        correct += (predicted == test_labels.to(device)).sum().item()
```

2.2 进度条更新

在每个批次结束后，调用 `update_progress_bar` 函数显示测试进度：

- 显示当前批次编号。
- 显示每批次的处理时间。

```
update_progress_bar(phase, batch_start_time, test_loader_len, batch_num,
batch_end_time)
```

---

3. 测试结果分析

3.1 混淆矩阵

调用 `ConfusionMatrix` 类生成混淆矩阵并计算分类性能指标（如准确率、召回率等）。支持可视化矩阵，以直观展示模型分类结果。

```
cm = ConfusionMatrix(y_true, y_pred, classes, normalize)
cm.plot_confusion_matrix()
```

3.2 性能评估

通过 `evaluation` 函数评估模型的计算量和推理速度，包括：

- **FLOPs**：模型每秒的浮点运算次数。
- **推理时间**：每次前向传播的平均时间。
- **吞吐量**：每秒可处理的样本数量。

```
evaluation(model, device)
```

3.3 输出结果

最终输出以下信息：

- **测试时间**：测试过程的总耗时（以分钟和秒为单位）。
- **测试准确率**：模型在测试集上的分类准确率。

```
print(f'Total time: {minutes:.0f} minutes {seconds:.1f} seconds.')
print(f'Best accuracy: {100 * correct / total:.2f}%')
```

---

4. 主函数调用

用户通过 `__main__` 主函数调用 `test`，并指定相关参数进行测试。例如：

- 测试模型 `efficientnet_b0`。
- 使用 `Adam` 优化器，加载已训练的 3 个轮次权重。
- 在 GPU 上进行测试。

```
if __name__ == "__main__":
    test(d='cuda', pre_epochs=3, model_name='efficientnet_b0',
optimizer_name='Adam', normalize=True)
```